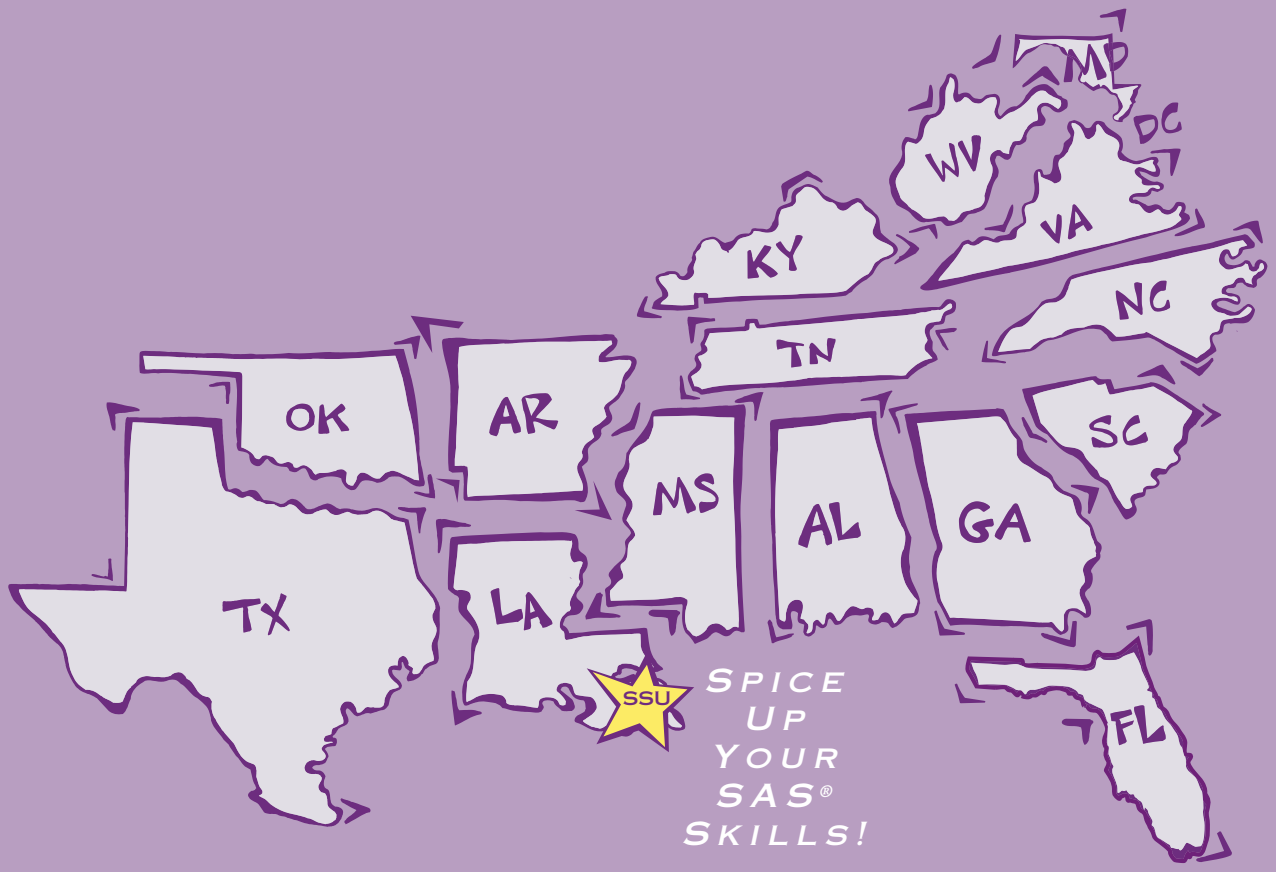
The Joint Conference of the South East and South Central SAS® Users Groups



Proceedings

Conference Co-Chairs

Deborah B. Buck
S. David Riba

August 19-22, 2001

Hotel Inter-Continental
New Orleans, Louisiana

SSU2001
Conference Proceedings

These Conference Proceedings are a permanent record of the 2001 Conference of the Southern SAS Users Group (SSU), a combined conference of the SouthEast SAS Users Group, Inc. (SESUG) and the South Central SAS Users Group, Inc. (SCSUG) Neither SSU, SESUG, SCSUG, or SAS can take responsibility for the accuracy or originality of the contents included herein.

The correct bibliographic citation of these Proceedings is as follows:

SSU 2001, *Proceedings of the Southern SAS Users Group Conference*
New Orleans, LA , 2001 948 pages

The SSU 2001 Conference Co-chairs, Deborah Babcock Buck and S. David Riba, developed these Proceedings. F. Joseph Kelley, our Proceedings Coordinator, prepared the final version, including the Table of Contents, and the Keyword and Author Indices, for publication. SAS Institute graciously printed, assembled, and delivered the final product as a permanent record of the 2001 Conference of the Southern SAS Users Group (SSU)

TO ORDER COPIES of the printed *Proceedings* or the companion CD (if available), CONTACT:

S. David Riba
JADE Tech, Inc.
P O Box 4517
Clearwater, FL 33758
Phone: (727) 726-6099
Email: dave@jadetek.com

About the Artwork and Cover

The cover was created by Kimberly Riddell of JCouch Design. It is based on input from Debbie and Dave but Jessie took the original concept and transformed it using her graphics artist's magic. Thanks, Jessie!

Printed in the USA

FOREWORD

In your possession you have the most enduring tangible product of SSU 2001: the Proceedings of the Conference for Southern SAS Users. Inside, you will find 140 papers by over 100 authors. A lot of SAS expertise, blood, sweat and tears went into the writing, editing, formatting, compiling, printing and shipping of this book to make it available to you when you picked up your registration materials.
We hope you get much use from these Proceedings.

Over four years ago, informal discussions began among members of the SouthEast SAS Users Group (SESUG) and the South-Central SAS Users Group (SCSUG) on the possibility of a one-time joint regional conference to be held in New Orleans, Louisiana. Why New Orleans? New Orleans is in the eastern-most portion of the South-Central region, but adjoins the western edge of the SouthEast region – and it's a fun city! Both groups felt that a New Orleans conference, utilizing the combined talents of SESUG and SCSUG, would result in an exceptional SAS conference in a world-renowned city. Approximately three years ago a Steering Committee was formed consisting of three members of the SESUG Executive Committee (EC) – Frank Dilorio, Andrew T. Kuligowski and S. David Riba – and three members of the SCSUG EC – Deborah Babcock Buck, Clarence Wm. Jackson and Thomas Winn – to develop a plan for this unique conference which would combine the strengths of both regional groups. Shortly after the official plan was approved by both the SESUG and SCSUG ECs, the Steering Committee was finalized with Randy C. Finch replacing Frank (who had to step aside due to work commitments), the Conference Co-chairs were selected, and the Hotel Inter-Continental was chosen as the conference site . Two years ago, a contest was held to decide on a name for this joint conference of two regional SAS User Groups, and "SSU 2001 – the Conference for Southern SAS Users" was officially named. The conference logo was designed, incorporating traditional New Orleans colors, the official Conference name and a cornet issuing forth some of the best known SAS keywords. From the inception until now, the six person Steering Committee has been active behind the scenes in all aspects of the conference.

As Conference Co-Chairs, we were responsible for all aspects of SSU 2001, from its initial development through the Closing Session. However, we did not do this by ourselves. We had a team of very dedicated, very hard working individuals who helped turn our ideas into the reality that became SSU 2001.

The selection of Section Chairs was a challenging task which involved the input of the entire Steering Committee. Our goal was to include representatives from both regions in each section, and to team experienced Section Chairs with those new to conference planning. Satisfying this goal was not always easily accomplished, but we tried hard to make it happen. This year, we introduced a new concept that has never been tried at a regional SAS conference before, a section called Introduction to SAS. For this new section, a team of experienced Section Chairs, who were also long-time SAS users, was selected to design and organize a one-day series of presentations which would guide new SAS users through a step-by-step approach to using SAS.

The Presentation Section Chairs had very important tasks, including recruiting top-notch presenters, reviewing and selecting papers, keeping presenters informed of deadlines (and enforcing these deadlines when necessary), and making sure the all-important papers which appear in this Proceedings arrived in time for publication. In addition to the Presentation Section Chairs, who are generally associated with the most visible portion of a conference, our Administrative Section Chairs oversaw the "behind-the-scenes" operations such as AV, food service, guest programs, sponsorship, publications, proceedings and registration. We deeply appreciate the outstanding job done by all of the SSU Section Chairs. Their names are listed elsewhere in this Proceedings.

A very, very special thanks goes to SAS for its many and varied contributions to the success of SSU 2001. Without their help, the SESUG and SCSUG regional conferences would either not exist or would cost the attendees considerably more than they do now. This year for SSU, SAS provided even more assistance because of the larger 2 ½ day combined conference, and the unique nature of this conference. Some of the support provided by SAS included printing and mailing the Call for Participation brochures, Registration Booklets, and the SAS User Group Conference brochures at their own expense. They also printed and shipped the Proceedings to the conference (this book). SAS also provided the services of our graphic artist, Kimberly Riddell, whose assistance was invaluable in designing our logo and the covers for these Proceedings. They also offered tremendous support at the conference itself by providing the Keynote Speaker, various section presenters, a Demo Room, prizes at the closing session, and much more.

FOREWORD

We would particularly like to thank Michael Smith, our SAS liaison, who has been a real pleasure to work with and has gone the extra mile for us a number of times. Our thanks also to Lisa Brugh, who was our SAS liaison earlier in this process and was also delightful to work with. We are thrilled that our longtime friend, Rick Langston, Manager, Core Systems Dept., Base Systems Research Division at SAS, agreed to be our Keynote Speaker, as well as host our Iron SAS gameshow and present several papers. Rick brings exactly the right mix of SAS expertise and humor to this conference. Finally, our thanks to all of the staff at SAS who have contributed to the success of SSU.

Our thanks to Marcella Moresco, Convention Services Manager, and the entire staff at the Hotel Inter-Continental for their patience and support over the many months that it took to bring this conference to fruition. Their willingness to work with us and to accommodate our requests has been one of the critical factors in the success of this conference.

We are grateful to those corporations who chose to participate in our "Corporate Sponsorship" program. They provided considerable, and appreciated, financial assistance to SSU. Their support enabled us to provide a number of additional benefits to the SSU attendees.

Our personal thanks go to both Andy Kuligowski and Joe Kelley for serving as our unofficial "sounding boards" throughout this process. Their "ear", and expert counsel, over the many months it took to plan and organize this conference helped both of us tremendously. While both Andy and Joe need to be recognized for their official roles in this conference, their unofficial roles were equally important and worth noting.

Debbie would like to thank her family and friends for all their invaluable support. She also appreciates Kayla and Stormy's patience (even if it was somewhat begrudgingly given) when she worked on SSU matters rather than spending time with them. Sincere thanks to Dave for his many creative and innovative ideas that helped make this such a unique conference and for his boundless energy in attaining our goals.

Dave, too, would like to thank his family and friends for their support and patience over the many months he has devoted to SSU. He would particularly like to thank Abby for her patience, sage counsel, and expert suggestions. Finally, he would be remiss in not thanking Debbie for putting up with the myriad "crazy idea du jour" emails and suggestions, and for providing the perfect balance necessary for success.

As SSU 2001 becomes a fondly remembered past conference, plans are underway for SESUG 2002 and SCSUG 2002. Let's start making our plans to attend either (or both) of the conferences. SESUG 2002 will be hosted by Heidi Markovitz and Dave Maddox in Savannah, GA. Your hosts for SCSUG 2002 will be Clarence Wm. Jackson and Dr. Neil Fleming in Richardson, TX. We hope you've had a "hot" time in New Orleans at SSU 2001 with plenty of opportunity to "spice up your SAS skills", and we hope to see you next year!

Deborah Babcock Buck
Presentations Chair

S. David Riba
Administrative Chair

CONFERENCE LEADERS

Conference Co-Chairs

Deborah Babcock Buck
D. B. & P. Associates
Houston, TX

S. David Riba
JADE Tech, Inc.
Clearwater, FL

Section Chairs

Data Warehousing	Tom Mannigel, Insyst,Inc. Bob Woods, Bank of America Corporation
Emerging Technologies	Jack Shoemaker, Statprobe Technologies Clara Waterman, Maxim Group
Hands-On Workshops & Volunteer Coordinator	Heidi Markovitz, Simply Systems
Internet, Intranet & the Web	Caroline Bahler, Meridian Software, Inc. Jimmy DeFoor, Brierley and Partners
Introduction to SAS	Imelda Go, Lexington County School District One Andrew T. Kuligowski, Nielsen Media Research Thomas J. Winn, Jr., Texas State Auditor's Office
Posters	Philip d'Almada, EDS Eric Brinsfield, Meridian Software, Inc.
Training & Weekend Workshops	John Bentley, First Union National Bank
SAS Solutions & Vertical Products	Matt Becker, PharmaNet, Inc. E. Barry Moser, Louisiana State University Kasi Peek, Blue Cross Blue Shield of Tennessee
Serendipity	Derek Nguyen, DataLogic Consulting, Inc. Ian Whitlock, Westat
Statistics & Data Analysis	Maribeth Johnson, Medical College of Georgia Lita Rosario, Marquee Associates
Student Coordinator	Sally Muller, University of North Carolina
Tutorials	Keith Cranford, Marquee Associates Carla Mast, Transmedia Network, Inc Joy M. Smith, NCSU, Dept of Statistics
AV & Food Service	Andrew T. Kuligowski, Nielsen Media Research
City Coordinator & Guest Program	Hester Johnson, Hibernia National Bank Robert Sigle, Entergy Services Corp.
Corporate Sponsorship	Randy Finch, Tennessee Valley Authority Clarence Wm. Jackson, City of Dallas
Publications & Proceedings	Dan Bruns, Tennessee Valley Authority Francis J. Kelley, University of Georgia
Registration	Elizabeth Hamilton, University of North Carolina Becky Peek, Independent Contractor Ann Stephan, Texas State Comptroller's Office

ORGANIZATIONS

SOUTH-CENTRAL SAS USERS GROUP EXECUTIVE COMMITTEE

Deborah Buck
Jimmy DeFoor
Clarence W. Jackson, Vice President
Mark MacMullen
Tom Mannigel
Rich Priem
Ann Stephan
Thomas J. Winn, Jr., President

SOUTHEAST SAS USERS GROUP EXECUTIVE COUNCIL

Greg Barnes Nelson
Dan Bruns, Vice President
Philip d'Almada
Frank C. DiIorio
Randy Finch, Treasurer
Maribeth Johnson
F. Joseph Kelley, President
Andrew T. Kuligowski
David Maddox
Heidi Markovitz
George Matthews
S. David Riba

Emeritus Members:

Melissa Garreans
Grace Lossman
Andrew Parks
Deborah Skinner

Tamara Fischell (dec.)

SSU 2001 STEERING COMMITTEE

Deborah Babcock Buck
Clarence William Jackson
Thomas J. Winn, Jr.
Randy Finch
Andrew T. Kuligowski
S. David Riba

SSU 2001
THE CONFERENCE FOR SOUTHERN SAS USERS

TABLE OF CONTENTS

EMERGING TECHNOLOGIES

TABLE OF CONTENTS

INTRODUCTION TO SAS

POSTERS

TABLE OF CONTENTS

SAS SOLUTIONS AND VERTICAL PRODUCTS

STATISTICS AND DATA ANALYSIS

TUTORIALS

DATA WAREHOUSING

SECTION CHAIRS

Tom Mannigel
Insyst,Inc.

Bob Woods
Bank of America Corporation

DATA WAREHOUSING

Producing Multipurpose Metadata for Data Quality, Trending, and a Data Dictionary

John E Bentley , First Union National Bank, Charlotte, NC

Abstract: A data warehouse provides a single version of the truth. The "truth", unfortunately, is often difficult to understand and is only as accurate as the data it is based on. The truth can change quickly and it's important to know just as soon as that happens. Metadata is critical for helping users understand their data, and specific elements are important for assessing data quality and useful for trend analysis. SAS Software provides the tools for generating, tracking, comparing, and reporting metadata. This paper presents an application that generates metadata in the form of one-way frequencies (for categorical fields) and descriptive statistics (for numeric fields) from newly loaded data warehouse tables, compares the current metadata to metadata produced in the previous load, and then reports major changes and anomalies. It also produces a series of html files containing frequencies and descriptive statistics linked to the data warehouse's data dictionary. SAS products used include BASE, Macro Language, AF/FRAME, SAS/CONNECT and Multi-Process Connect, and the Output Delivery System. The application was developed under UNIX, but can be ported to any operating system. All levels of SAS users will find something useful in the presentation.

Simplified Software Project Management for the Rest of Us Or a Twelve Step Program for the Chronically Overworked Programmer, Project Leader or Manager

Tom Mannigel , Insyst,Inc.

Abstract

Many SAS systems such as Data Warehouses are build by one programmer or a single team who are looking for all the help they can get. Unfortunately the so called "Rapid Development Techniques" designed to help quicken software projects are for the most part for huge teams spending millions of dollars or software that's sold by the millions. For that reason these approaches are very complex and require a lot of resources. Something-smaller projects don't have. There's a big difference in the support and resources for a project that the president has bet the company's success on and a "whatever's needed" custom SAS system for a small group of very impatient end users. It's like trying to build a custom home using the same approach you'd use to build a skyscraper. Unhappily you're probable going to waste and not save time. There must be a simpler way. There is. "Simplified Software Project Management For the Rest of US", describes a 12 step approach I've developed over the last couple of decades to help speed software projects for the rest of us who don't have the luxury of infinite resources and time.

Introduction

The goal of this approach is to save time and increase your value. There may appear to be addition unneeded steps. But let me assure you every step is important and design to save time in the long run.

The twelve steps are with approximately percentage of the total project:

1. Do a preliminary design and estimate 2.5%
2. What's it's worth? 3%
3. What are the risks? 2%
4. Do a very detailed design and schedule. 15%
5. Build the system. 45%
6. Have weekly progress reports. 5%
7. Do systematic testing. 5%
8. Do transaction level testing. 10%
9. Do user testing. 5%
10. Document it quickly. 5%
11. Have a Party. 0%
12. Was it worth it? 2.5%

Now let's go over each step in detail:

Step 1. Do a preliminary design and estimate.

What you do.

This is a step that must be done for all projects now matter how large or small. For this step create a quick generalize view of what the system would look like at a high level. Something like: a reporting system that reads x number of files and creates y number of reports. Based on the size and number of reports you make a quess as to how long it should take to do the project. Remember that this is a quick and dirty estimate. A more detail estimate will come later.

How long should you spend.

This step should take from five minutes to a few days based on the size of the project. For this and all the following examples, a small project is 1 programmer for six weeks and projects at the high end is a team of seven for 6 months.

How this step saves time.

This step eliminates requests that are not worth doing without a lot of effort on your part. Note if you eliminate on average 1 out of 40 projects at this step you've save time. However be honesty in your evaluation because you don't want to exclude a high value project because you grossly over estimate it.

Step 2. What's it's worth?

What you do.

This is a step that is usually only done on large project but, as I will show it needs to be done on all projects. Given your design from the previous step what is the value of the new system directly in dollars and indirectly in intangible benefits. Take the time to ask some simple question like: What are the benefits of new system?
Will be increase sales or cut cost?
Will it save time, make someone job easier or less boring?
Will it help to get a product to market sooner or just get information quicker?
Having taken a look at its look at its benefits then ask the question does the benefits **far** out weigh the cost. I've emphasized far because we are in a resource scarce

business and if this project does not "obviously" create massive benefits then you need to move to something that will. Our goal here is to maximize your value and one obvious why is to make sure that your involve in only the best highest value projects.

How long should you spend.
This step should take from a few hours to a couple of weeks for the range of projects we are considering here. Sometimes this step can be a project in itself. But is important to note that you don't have to know down to the last dollar the value of the new system because we are looking to do only high value system with an obvious benefit. If your value estimates are off a little, we still should be able to identify high value projects.

How this step saves time.
Even though this is probably an addition step for most people. It is one of the most important. It will save time by eliminate low or no value projects very early. The last thing you want to do is build a system that has no value. At this point you've spent 5% of the project time if you eliminate 1 out of 20 your time ahead.

Step 3. What are the risks?

What you do.
At this point in the project you need to determine the chances of success. There are three factors that determine the probability that a project will succeed they are:

1. The complexity of the technology involved
2. The expertise of the people doing the work
3. The support the project has.

Give each of these categories a number from 0 10 based on the following:

Category 1➔ 0 is leading edge technology that
Has never been tried
10 for twenty year old proven
Technology.
Category 2➔ 0 for novices
10 for experts.
Category 3➔ 0 for no support
10 for the a managers support for the small project or the president of the company for the large project.

Add the three numbers together.
The projects with a these total have the following chance of success.

25 to 30	Excellent
18.to 24	Good
13 to 17	Fair
below 12	Poor.

Given these ratings wait the potential return versus the risks. High risk should have a corresponding high return. If not you may need to beef up the category that is low either a less technologically difficult design, more expertise or more support.

How long should you spend.
The amount of time for this step should not be very much if you use the simplified approach I've outlined here.

How this step saves time.
The function of this step is twofold. One this gives the client at an earlier point in the project a sense of the risks involved. And secondly to eliminate or correct at the projects start possible factors that can cause failure later on
which is major time and value waster.

Step 4. Do A Very Detail Design and Schedule.

What you do.
This is the step where real work begins. Sometime this step is skip and the programming step begins using the preliminary step design to go by. My experience is that this is were a lot project get into big trouble. First of all because it is a rough estimate and the design is not tied down clearly, the estimate can be off by a factor of 10 which usually makes for a very unhappy client. I'm recommend that you do the detail design at this point in the project because it's the best time to do it. At some point your going to have to nail down every detail to deliver the system why not do it at a point that it does you the most good and at a point where changes have the least impact.

One question is how detailed a design do you do. I suggest that you have enough detail that a programmer can do their job without needing any more information. Doing the design at a programmer level or data flow is too inefficient. It's like programming the system twice. Normally I build process flow diagrams and have all the supporting information such as data dictionaries, file definitions and report layouts that a programmer would needs to their job. I also create an estimate and schedule of how long each step should take.

How long should you spend
This is a major step for the project and should take from 10% to 20% of the total project time.

How this step saves time.
This is a very important step and should not be skip. It saves time by determining the exact cost and resources required build the system. If the ballpark estimate is really off then the client needs to know now so they can cancel the project if its not worth it or resoucres are not available. Most projects get into trouble because of bad estimates. The better the design the better the estimate the better the project will go. Also I've found that the clearer

the design the less time you'll spend on programming step which the largest step of the project.

Step 5. Build the System.

What you do.
For this step I like a "code to the max" approach. If you've done the previous steps properly this step which requires the most work should go smoothly and rapidly. If it's not going smoothly then you need to revisit the design, expertise or your support.

How long should you spend
This step is where most of the work is done and should take from 40% to 60% of the project. To little time here and value of the project will suffer. Too much time and you're not at max effectiveness.

How this step saves time.
This is the step that everyone must do. Again if you've done the work leading to this step you should truly be able to build a system rapidly that you know is high value and not a waste of time because its never used.

Step 6. Have weekly progress reports.

What you do.
Based on the progress to make weekly reports to all the people that involve in the project. Include in the report should be a weekly summary of progress which includes an estimate of the number of days the project is either ahead or behind based on the estimate in step 4. I also include the following files:

1. Open issues file.
2. Unexpected problems file.
3. A changed to original design.
4. A suggested enhancement file.

How long should you spend
This again is a very important step that is usually skip or done occasionally. You should spend from 4 hours/week to 2 days a week.

How this step saves time.
The value of this step is that there are no surprises at the end of the project. You're building critibility during the whole project and if the project is over budget you've got justification for it. The time save is that the project is not canned at the last minute because it over runs and you don't get enough time to fix it.

Step 7. Do systematic testing.

What you do.
This is the first of three testing steps. I've divided testing into three steps to emphasis its importance.
In this step the programmer checks for obvious problems. I call system bugs those that effect the whole system. There usually are very obvious and can be found by asking: Do the results make sense? Or by doing a ballpark estimate of what results should be.

How long should you spend
This step should spend from 2 of days to a couple of weeks.

How this step saves time.
Since this is a rather quick first crack at testing you should get a sense of how well the program is structure and programmed.
If you find a lot of problems you can easily go back and do some restructuring. The intent here is to get the system bug free before it goes into production. Fixing bugs after the system is in production takes four to ten times more time then at this point.

Step 8. Do transaction level testing.

What you do.
For the second level of testing you manually trace several transactions or the lowest level of data through the system to make sure every step is correct. This step will check every detail of the system.

How long should you spend
This step should spend from 1 week to a man month.

How this step saves time.
No system is complete until it is 100% bug free if you face that at this point and do the testing now you save a ton of time later. You going to spend at least 15% of the time fixing bugs you need to do it where you the most effective and now is the time.

Step 9. Do End-User testing.

What you do.
The last level of testing has two purposes, I 've found that users are the most effective testers. They are the fastest bug finders available. For this step have an end-users take a look at the results and see if they can detect any problems.

How long should you spend
This step should spend from 2 days to a week.

How this step saves time.
This step saves time because it starts the transition process and end-users are the best tester going. They can find bugs like no other and in less time than no other can. But be careful not to over use them.

Step 10. Document it quickly.

What you do.
This step is only important from a standpoint of supporting and enhancing the system once it done. I recommend the following be done. All programs should be self-documented and be done as you build the program. Included also should be the final process flow diagrams. User documentation should be minimal if any at all.

How long should you spend
This step should spend from 2 or three days to a week.

How this step saves time.
Everyone wants lots of documentation but no one uses it. So don't spend a lot of time doing it. You should provide enough documentation that a new knowledgeable programmer can make changes to the system. If the system requires a lot of user documentation's the system is in trouble.

Step 11. Have a party.

What you do.
This is a very consequential step that I learned from a Six 6 company. They always have a party at the end of every project for a lot of substantial reasons. One is it fortifies the project as being a big success. It also gives everyone something to look forward to. To have a party you have to have an end date and get user acceptance. The end date is important because it forces the project to a conclusion. Some projects go on forever, wasting a lot of time. And some projects never really get accepted by the end-users creating lots of problems. I say have a party.

How long should you spend
One Lunch hour.

How this step saves time.
It is very important from a success and value standpoint that the system be used by the end-user. This step encourages the end-users to use the system and also not to let it drag on.

Step 12. What is the final value of the system?

What you do.
After you've had the party and the systems been in use for a while determine the true value of the system.

How long should you spend
A few days.

How this step saves time .
To be honest this is a step that is rarely done but I believe is the most meaningful.
This is the step where you learn what you did right and what you did wrong and what it was worth. So that the next time you can do a better faster job with more value.

Tom Mannigel is President of Insyst, Inc. a Houston based SAS Institute Quality Partner and has been a SAS system developer since 1981. His company has built over hundred different systems with a 97% success rate. He can be contact at tmannigel@aol.com.

Paper P103

A Scorecard approach to improving Data Quality

Rob Phelps, Phil Nousak, PricewaterhouseCoopers LLP, Chapel Hill, NC

ABSTRACT

An ever-increasing number of strategic and tactical business decisions are being made from analyzing data gathered in Data Warehouses and Data Marts. Bad decisions, poorly performing predictive models and monetary losses result when data quality is not monitored. What you think you know about your organization, your customers, or your suppliers may be distorted by undependable data.

Data quality cannot be improved independently of the process producing the data or the context in which the data is to be used. Technology-only approaches are not sufficient to provide sustained data quality improvements. The road to data quality improvement involves several factors. These include a technical understanding of the data and data gathering processes. It also includes the establishment of a reporting process to monitor changes in data quality as well as people with well-defined roles, responsibilities, and authority to develop a culture that supports data quality improvement.

This paper will describe a scorecard-based approach to identify, measure and monitor data quality problems. It will cover the people and processes needed to sustain such an effort, as well as an implementation using SAS software to build the technical infrastructure. Although the focus for the paper is on data quality assessment in a data warehouse, much of the approach can also be implemented outside of a formal system.

INTRODUCTION

DATA QUALITY BUSINESS ISSUES

The PricewaterhouseCoopers LLP Global Risk Management Solutions Data Management Survey 2001 sampled a broad mix of major 'Top 500' corporations, middle-market businesses, and companies primarily engaged in e-business.

Results from this survey show that over 75% of Chief Information Officers, IT directors or equivalent executives at 600 companies across the US, Australia and UK reported having experienced significant problems because of defective data.

The survey also shows that poor data quality causes hard dollar loss, failed securities deliveries, missed corporate actions, or erroneous trading decisions. Data quality issues often result in the following:

- Extra costs to prepare reconciliations
- A delay or scrapping of a new system implementation
- A failure to bill or collect receivables
- Inability to deliver orders or lost sales because of incorrect stock records
- Failure to meet a significant contractual requirement or service level performance

The following are some of the areas related to Human Resource (HR) systems that are adversely affected by poor data quality.

- ERP Conversion
- Outsourced HR programs such as Pension Plan Administration
- Employee Workforce Planning
- Globalization and /or Integration of Business Operations
- Mergers, Acquisitions, Divestitures and Reorganizations
- Government Reporting
- Labor Negotiations

As e-business becomes more pervasive, the rising exposure to poor data quality increases the risk of incurring greater internal costs as well as costs to on-line commercial relationships. Investors are becoming increasingly sensitive to data problems as a sign of a deep malaise at the core of any organization. As reporting methods expand across non-financial areas in support of strategic balanced scorecard management models, the reliability of all kinds of data will come under growing scrutiny.

DATA QUALITY BUSINESS STRATEGY

Companies must take a strategic view of insuring quality data. This includes a process for monitoring and correction of data quality issues supported by sound and demonstrable data metrics.

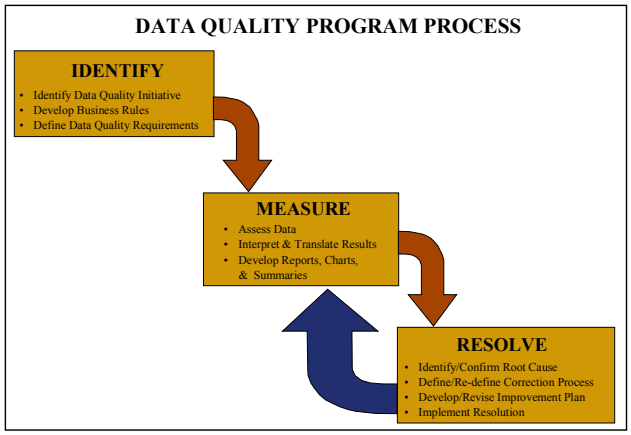
This paper represents a holistic approach to the ongoing issues related to organizational data management. It begins with an overview of a fundamental process as the basis for the establishment of a program for improving data quality over time. Next, an approach for the establishment of a Data Quality Program (DQP) is presented with a discussion of the types of errors found in any collection of information and a primer describing the process for quantifying systematic errors. A description of the metrics and reports needed to systematize the DQP follow, as well as a section describing the roles and responsibilities for people to support a data quality effort. The final section includes a technical framework with consideration for implementation using SAS software.

OVERVIEW OF A DATA QUALITY PROGRAM

The Data Quality Program (DQP) is a single point of reference for addressing all issues affecting data quality in an organization or business unit. It provides a forum representing all points of view within the Information

Systems community in defining, identifying, measuring, analyzing, and resolving data quality issues. The DQP provides a foundation for making decisions and provides direction throughout the quality management process. Figure 1 provides an overview of the data quality program process.

FIGURE 1



IDENTIFICATION

The first step is to identify data quality improvement opportunities and define business rules and data quality requirements. It is important to re-visit the identification phase, as often as new data quality measurement needs are determined.

In the Identify phase, the DQP analysis files are built from extracts of source system data or from the Operational Data Store (ODS) in a data warehouse. These data, including tables, records and elements, are extracted in their "native" form and are made available to a reporting tool before any transformation process has occurred. Business rules are used to monitor and report data quality problems. In the DQP, these rules are called filters. Examples of business rules that can be verified with filters include:

- A salary change date that cannot be earlier than the hire date
- A salary that cannot be less than the minimum wage
- An employee who must be 15 years old before his/her date of hire
- Social Security numbers that are not numeric
- A last shipped date that is less that the last ordered date

Suspect records that do not meet these data quality requirements (e.g., consistent, valid and complete) are then identified. The records are presented in detailed reports and used to provide the information to point to the root cause of the DQ issues.

MEASURE

This step involves the application of data quality metrics to data attributes or records from data loaded into a data warehouse or other system. These metrics are translated into business terminology. The measures are communicated by the creation of detailed reports, summaries, trend analyses, and scorecards that portray data quality levels and present recommendations.

RESOLVE

Understanding of the root cause of these quality problems are needed before resolution can occur. After identifying and measuring quality issues the next step is to formulate a correction process with business case justifications. This involves the development of a task schedule and assigning responsibility to execute the correction process.

The correction steps provide the ability to confirm (or modify) the root cause, re-define the correction process, and incorporate modifications into an improvement plan. It will be necessary to repeat the measurement and resolution phases for the data quality improvement process to ensure quality maintenance.

The resolution part of a Data Quality Program depends on the coordination of the individuals responsible for the data quality review. Described in a latter section are the roles and responsibilities for staffing such a program.

In the resolution phase, the DQP Business Analyst collects the DQ detail reports and distributes them to the source data owners as appropriate. The automation of this process happens in the latter stages of deployment.

After reviewing the reports, the source data owner can:

- Correct the data in the source system, or
- Recommend modifications or additions to the filter list, or
- Confirm that the suspect data is acceptable and document its occurrence

.

AN APPROACH TO MANAGING DATA QUALITY

TYPE OF ERRORS

Problems or errors in data can occur either randomly or systematically. Systematic errors often occur as a result of a misapplication or misinterpretation of business rules. Systematic errors can be dealt with in a number of ways, including modifying the data collection process, introducing a systematic correction method, or simply reporting the inconsistencies. Random errors, in contrast, require direct review of input records and are not resolvable by systematic means.

CHECKING FOR ERRORS

One way to check for problems in data is to use filters to identify suspect records. The records are suspect for identification and classification purposes. Later steps are taken to either correct the information identified by filters or to document any discrepancy.

Filters represent simple conditions that identify a single issue with the data analyzed. The types of filters range from the simplest checks of the domain or range of a single variable, through comparisons of one or more fields within or between records in a table, to a complex review of multiple fields across multiple tables and source systems.

The following outline presents types of filters that may be prepared for a database. The outline presents the filters in order of increasing complexity, from simpler within-record checks to the more complex between-table and between-system checks. The key to building a sound DQP is to begin with the simpler filters within a system and build to the more complex filters among systems.

Within variable

- value in domain (e.g., categorical variable matches reference list; numeric variable in range)
- test for missing value when appropriate (e.g., table key, required variable)
- valid date

Between variables

- cross-check in domain (e.g., salary increase matches compensation transaction code)
- test for logical relationships

Between records

- unique key when appropriate
- proper sequence (e.g., employment activity only before termination)
- missing records (e.g., apparent pay change record not in table)
- proper transaction variable assignment (e.g., comparison of selected variables between records meets business rules requirements for transaction)
- comparison of values consistent (e.g., comparison of base rate between records does not match indicated pay change)

Between tables

- check key relationship across tables (e.g., list of expected employ ids matches appropriately across tables with employee data)
- cross-check in domain (similar to within table check)
- test for logical relationships (e.g., observation in separate evaluation table matches employee work history sequence)
- valid date relationships (e.g., separate work history tables join to produce acceptable representation of employee's work history)

Between systems

- check key relationships across tables between systems
- cross-check in domain
- test for logical relationships
- valid date relationships

DATA QUALITY ACTIVITIES AND PROCESSES

The DQP activities identify where quality problems exist, ascertain the magnitude of the problems, and propose solutions. These activities require the DQP to regularly measure data quality levels, provide mechanisms to share data quality information, and maintain organizational accountability for data quality. The activities identified in this section occur throughout the development, enhancement, and maintenance of the DQP life cycle.

PROCESSES

The following is an outline of the process steps necessary for the establishment of a DQP:

- Develop detailed procedures that provide a logical, organized approach to addressing and resolving data quality issues using the high-level process depicted in Figure 1. The DQ process must be scaleable for use on small or large quality problems.
- Develop data quality metrics that measures the level of data quality in information systems monitored.
- Define procedures that apply the quality metrics to the data for each information system in order to monitor its data quality level over time.
- Identify procedures for communicating DQP activities.
- Develop a Data Ownership Policy and define procedures for data owners to perform data quality functions. Clear identification of data and process owners and definition of their responsibilities will facilitate an effective DQP.

ACTIVITIES

The following are the activities needed to establish the DQP processes:

- Identify or verify the authoritative sources of data and assign the necessary data ownership responsibilities.
- Define data quality measurement criteria.
- Propose recommendations to improve the level of data quality based on results of data reviews.
- Develop goals, objectives, plans, and tasks for data quality improvement activities.
- Maintain data quality for current Information Systems by establishing and communicating procedures to personnel whose job function it is to create, update, and delete data .
- Maintain data quality for current Information Systems by developing edit and validation rules that filter the data before storing it in the database, if applicable.
- Report the status and results of data quality improvement activities to the necessary personnel, e.g., Upper Management, Information Systems Management, application support personnel.
- Regularly publish data quality level information for identified Information Systems to audiences at the necessary organizational levels.
- Manage and control DQP work products.
- Coordinate the necessary data quality

improvement tasks with the appropriate data and process owners and application support teams.

DATA QUALITY METRICS

Data quality metrics are defined and categorized into two groups. *Generic* metrics apply to all columns and/or all tables (e.g., a record count metric, such as the number of records in the tables). *Specific* metrics apply to specific columns, or combinations of columns, in specific tables (e.g., an accuracy metric, such as the number of valid values in a table).

DEFINING DQ METRICS

The following are principles to consider when defining data quality metrics:

- Metrics should be insensitive to changes in the number of records in the warehouse;
- Metrics should accurately reflect the degree to which the data meets the associated data quality need;
- Metrics should be independent of each other, so that no two metrics are actually measuring the same effect; and
- The number of metrics chosen should be kept to a reasonable number, as too many metrics can often confuse rather than clarify.

DATA QUALITY CATEGORIES

We classify Data quality metrics into categories that describe the methods used to analyze quality of data. A data value is generally accepted as having high quality if it meets the appropriate combination of the categories that are applicable to the element. Following in Figure 2 are the categories used to group the measures of data quality.

DATA QUALITY REPORTING

The Data Element Quality Scorecard contains the measured level of quality for each data element in the DQP. The scorecard lists data quality categories as columns and data elements as rows. Single summary statistics are created for four types of cells on the worksheet:

- Data element and data quality category combination (e.g., Field1/Valid);
- Quality category across all data elements (e.g., Valid for Field1 –> Field4);
- Data element for all quality categories scored (e.g., Field1: Valid, Unique, Complete etc. - some categories may not be scored); and
- Overall data quality including all data elements and quality categories scored.

FIGURE 2

Name	Description	Example
Valid	Data element passes all edits for acceptability	A Person record has a Name that contains numbers
Complete	Data element is (1) always required or (2) required based on the condition of another data element	A Payroll record misses a value for Person
Consistent	Data element is free from variation and contradiction based on the condition of another data element	A New Hire record has a Hire Date before their birth date Leave of Absence is checked, but the employee is at work
Unique	Data element is unique—there are no duplicate values	Two Person records have the same Social Security Number
Timely	Data element represents the most current information resulting from the output of a business event	A New Hire record references an Organization that has been sold
Accurate	Data element values are properly assigned	An HR Organization record has an inaccurate or invalid hierarchy
Precise	Data element is used only for its intended purpose, i.e., the degree to which the data characteristics are well understood and correctly utilized	HR Organization Department codes are used for different organizational entities between different records

The summary statistics calculation uses mathematical principles and formulas that allow for uniquely categorizing data quality problems. Both row and column totals in all cases are equal to or less than the individual cell totals. This is because any one record may have encountered multiple filters that identified suspect information. Filters must not encounter any suspect fields in order for the record to have complete quality data. The formula for calculating cell total is described in a latter section.

DATA QUALITY REPORTS

The DQP produces a set of reports for different purposes.

Detail Field Suspect Report. The report lists records that are of suspect for each filter. It is created to help the Information System representatives or service centers to examine the suspect records and make corrections when appropriate.

Summary Filter Suspect Report. The report lists the description and summary of suspect records by filter in the current run of the DQP. It is useful for the application developers to examine the appropriateness of the filters they have implemented.

Summary Field Suspect Report. The Summary Field Report presents the number of suspect records in two dimensions: data elements in row and data quality categories in column. A data element is a collection of one or more source fields in the system. For example, employee's date of birth is a data element, while the

employee's name may contain three fields in the source system, the first name, the last name and the middle name initial. Individual cells in the report contain the count of suspect records that fall into the data element and data quality category.

Since a data element may contain more than one source field, and a data quality category may contain more than one filter, a cell in the report may contain multiple filters. The DQP incorporates an algorithm that prevents a multiple count of the same record for each cell, including the row and column totals. Therefore, a row total is the number of records that do not meet one or more data quality criteria for the data element. On the other hand, the column total is the number of records for which one or more data elements do not meet the data quality criteria in that data quality category.

Finally, the total at the lower right corner is the total number of records that do not meet DQP criteria in at least one of the data elements that are audited. As for the *Trend Analysis,* the total number of a row or a column or the whole table is the number of suspect records, not the number of errors by filter or field. Therefore, the total is not the simple arithmetic sum of its elements. This report is illustrated in Figure 3a.

Figure 3a

DW Data Quality Field Report Records Processed: 9,999								
Element Name	Data Element Quality Category (# Suspect Records)							Total
	Valid	Unique	Complete	Consistent	T..	A..	P..	
Field1	0	0	0	N/A	N/A	N/A	N/A	**0**
Field2	259	N/A	176	N/A				**260**
Field3	0	2	228	N/A				**228**
Field4	720	N/A	604	4				**720**
Total	**720**	**2**	**1000**	**4**				**1000**

Data Quality Scorecard.
The DQP scorecard converts the Summary Field Suspect Report into a percentage of records that pass all the data criteria set in the DQP, i.e., cell percentages are calculated according to the following formula:

Percentage = [1 – (Cell count in the worksheet) / (Total number of records)] × 100%.

An example of a Data Quality Scorecard is shown in Figure 3b.

Figure 3b

HRDW Data Quality Scorecard Records Processed: 9,999								
Element Name	Data Element Quality Category (% Quality Data)							Total
	Valid	Unique	Complete	Consistent	T..	A..	P..	
Field1	100.00%	100.00%	100.00%	N/A	N/A	N/A	N/A	**100.00%**
Field2	97.40%	N/A	98.24%	N/A				**97.40%**
Field3	100.00%	99.99%	97.72%	N/A				**97.72%**
Field4	92.80%	N/A	93.95%	99.96%				**92.80%**
Total	**92.80%**	**99.99%**	**90.00%**	**99.96%**				**90.00%**

Trend Analysis. The report presents the number of suspect records by field for the current and previous runs of the DQP. It presents to the Information System representatives or service centers the progress of the quality of the data for which they are responsible.

DATA QUALITY STAFFING

Collectively, the members of the DQP are responsible for both the data quality management process and data integrity in the core systems. The job descriptions that follow are not necessarily full time positions but represent the roles and responsibilities for various aspects of a DQP. As the number of systems monitored increase and the DQP is better established, the time commitment for these activities will vary.

The members of the DQP must have well-defined roles, responsibilities, and authority to successfully improve the quality of the data in the organization. The DQP may have individuals serving one or many roles. It is important to understand that the creation of a separate DQP group is an option, but not required.

Each member of the DQP is assigned to at least one of the roles listed below:

- Data Warehouse Data Quality Manager
- Information Systems Manager
- Data Warehouse Data Quality Personnel
- Application Developers
- Data Users / Data Producers
- Business Unit Functional Experts

The following section contains a description of each role and defines the function or purpose of the role in the following two scenarios: day-to-day data quality operations; and DQP meetings, forums, or activities. The responsibility description attaches accountability and authority to a role. Each role will list the skills required of an individual to perform the job on the DQP.

DATA WAREHOUSE DATA QUALITY MANAGER

Daily Data Quality Role: Gains economic support for the data quality management process, defines the budget, and monitors the development and maintenance schedules.

DQP Role: Serves as a full-time member by participating in every DQP meeting in the capacity of ensuring adequate resources and funding for performing data quality improvement and maintenance activities.

Skills: Project management, data quality concepts, and knowledge about the Business processes that produce the data.

Responsibility: Keeps the data quality management initiatives on-track by being on schedule and within budget. Data Quality Personnel and Developers should receive direction form the Data Quality Manager as the scope and focus of the DQP evolves.

INFORMATION SYSTEMS MANAGER

Daily Data Quality Role: Gains economic support for system projects, defines the budget, and monitors system development / maintenance schedules. Implements data administration policies, procedures, criteria and standards for information systems data. Reviews and reconciles Information Systems data models at logical levels of detail, if applicable. Implements procedures for communicating data requirements and resolution activities among all relevant personnel.

DQP Role: Serves as a part-time member by participating in selective DQP meetings in the capacity of ensuring adequate resources and funding are available to support data quality improvement and maintenance activities.

Skills: Project management, systems life-cycle development, systems integration, and knowledge about the Business processes that produce the data.

Responsibility: Ensures data quality improvement and maintenance tasks are included and tracked in the DQP project plans.

DATA WAREHOUSE DATA QUALITY PERSONNEL

Daily Data Quality Role: Ensures all data quality management tasks are performed in sequence and in an expeditious manner. Identifies or captures data quality problems, defines data quality requirements, assesses the specified data, and develops reports, charts, and summaries that depict the data quality status of the Data Warehouse. Designs automated components and manual processes that will identify, measure and communicate the quality of the data from the source systems in the data warehouse. Supports the analysis of source data, extract/transform. Measures the data quality levels at regular intervals to monitor data improvement activities.

DQP Role: Serve as full-time members by participating in every DQP meeting in the capacity of providing information both on the status of data quality issues and specifics about each data quality issue. Presents the root cause of poor data quality and recommends methods to improve the data quality levels.

One member is responsible for facilitating every DQP meeting to ensure that the meeting objectives are met and manages data quality improvement and maintenance activities.

Searches for the causes of poor data quality and resolves incompatibilities between the systems.

Skills: Business area process knowledge, database management, system life-cycle development, data quality metrics, and basic programming logic.

Responsibility: Institutionalize the data quality management process and knowledgeable about data quality issues.

APPLICATION DEVELOPER

Daily Data Quality Role: The role is typically a role performed by those who develop and maintain the data warehouse and system(s) that capture data that feed the data warehouse. A system analyst or programmer manages the physical database of his/her respective system.

DQP Role: A system analyst or programmer from each project team serves as an invited member to participate in DQP meetings or activities depending on the data quality issue or the type of information that is planned to be improved. Communicates information about the data structure and content in their respective system.

Skills: Business process knowledge, database management, systems life-cycle development, structured programming.

Responsibility: Accountable for the integrity of the physical data structures and the extent that the physical database structure reflects the requirements provided by the business units. Uses standard data definitions when creating or modifying the system's database. Responsible for communicating system modifications to the DQP that affects the format or content of the data.

DATA USERS/DATA PRODUCERS

Daily Data Quality Role: A role played by every information system user who interacts with data in the selected Information Systems as part of their job function. This role includes Managers who are accountable for the business process, but may not directly interact with data in the systems. The personnel identify data quality issues and may participate in the data correction processes.

DQP Role: Serves as an invited member to participate in DQP meetings or activities depending on the data quality issue or the type of information that is to be improved, thereby representing their perspective of data usage. Understands the business process and how data are captured and maintained in the source system. Considered the customer of the data warehouse or source systems. These individuals set quality expectations for the DQP by defining requirements for data quality. They are involved in the data quality review and cleanup activities.

Skills: Knowledgeable about the business function and processes that produce the data.

Responsibility: Responsible for using data as originally intended and for notifying the DQP of data quality problems. Additionally, they are accountable for integrity of data within their job function.

FUNCTIONAL EXPERTS

Daily Data Quality Role: A role played by people who are

considered expert in a particular business function. Provides the knowledge and expertise essential to developing and validating data checks, filters and business solutions.

DQP Role: Serves as an invited member to participate in DQP meetings or activities depending on the data quality issue or the type of information planned for improvement. Understands the Data Users'/ Data Producers' data quality requirements. Also, provides solutions that enable data to be captured in a way that satisfies both the operational data quality requirements and the Data Users'/Data Producers' requirements. Collects and verifies business rules that ensure quality data and resolve issues or conflicts concerning data usage or interpretation.

Skills: Knowledgeable about the business environment, business function, and technology and processes that produce the data.

Responsibility: Responsible for defining and validating the names and definitions of data elements within their subject of expertise to meet the Data Users'/Data Producers' needs. Accountable for the integrity of data definitions and ensuring that the Data Users'/Data Producers' maintain data values that are in accordance with the definitions.

DQP TECHNICAL FRAMEWORK

SYSTEM REQUIREMENTS

A system for reporting Data Quality is incorporated into a production stream or used as a stand alone system to do periodic 'ad-hoc' reporting. The best approach is a modular one that allows for relatively easy modification and expansion to suit the needs of the warehouse developers, data owners, and other members of an organization that need to monitor data quality.

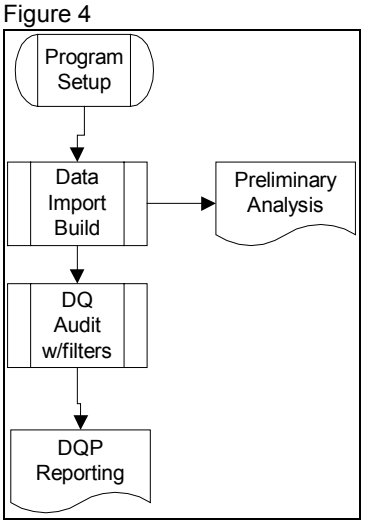
The following are key features of an application for Data Quality Reporting:

- Modular programming that is easy to maintain and is extendable.
- Portability and scalability to other platforms with access to extensive data preparation and analytic tools.
- Easily maintained support files that that can be modified to build the DQP database as well as activate or de-activate new filters.
- Self-documenting support files for identifying data quality issues and preparing the data for analysis.

DQP PROCESS FLOW

Figure 4 is a high-level diagram showing the process needed to evaluate data for Data Quality purposes. The preliminary analysis represents a profile of the characteristics of the data prior to applying 'business rules' for accessing data quality. The DQ Audit with filters represents a detailed reporting process that produces a scorecard and detailed reports for measuring and resolving data quality issues.

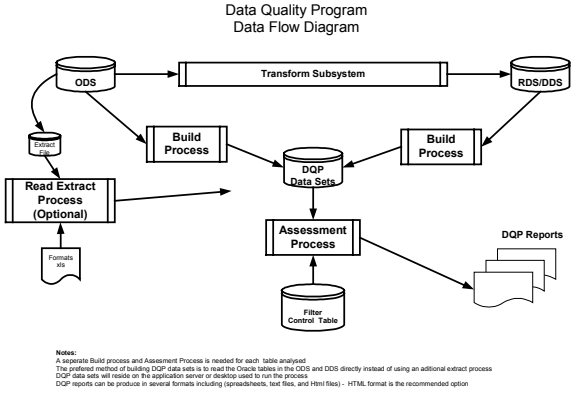
Figure 4



DQP DATA FLOW

Figure 5 is an example diagram showing the data flow in an automated system written using Base SAS® and Macro control language. Customization of this application occurs through entries in tables or l spreadsheets. These tables specify the format of the input data and the SAS statements used to establish the filter conditions.

Figure 5



FILTER CREATION/REFINEMENT

Filters represent the 'Business Rules' used to measure the quality of the data in the DQP. These 'rules' form the basis of communication about the details of measuring quality data. A robust solution would include a central repository of these filters and an easy way to document and modify them. It should also provide the following:

- Easy to add, activate, or de-activate filters.
- Ability to implement a wide range of filters without having to create customized programming logic.
- Flexibility to include filters that are more complex if the user is familiar with a programming language.
- Ability to point to a diverse set of lookup tables for consistency and completeness checks.

- Easy documentation of the filters that are active in the DQP at any time.

Most filters, in our experience, can be implemented with one or two SAS statements. Filters that are more complex may be implemented using macros. This allows the actual SAS code used in the system to serve as documentation when discussing filter criteria with the business specialist involved in the DQP.

Filters may be thought of as program statements that result in logical binary fields (0,1). These fields are summed in order to determine the number of unique records with suspect data, and allow the creation of a Data Quality Scorecard. There can be multiple filters (with the same DQ category) applied to a single column of data. The presence of any condition that triggers a filter acts as a unique counter for identifying which records contain suspect data. The summation of these fields allows the calculation of overall data quality for the entire table.

Following are excerpts from a set of filters used to measure data quality. Included in Figure 6 is an example of fields contained in a filter control table. It is listed to demonstrate the functionally and flexibility of an automated solution.

Filters for Field=SSN Type=Character

#	Description	Cat	SAS Code
N1	Missing value	Com	If ssn=' '
N1	SSN is Invalid	Val	MACRO SSNCheck(substr(ssn,2,9))

Filters for Field=EMPLID Type=Character

#	Description	Cat	SAS Code
S1	Missing value	Com	if emplid=''
S2	Not unique	Uni	if not (first.emplid and last.emplid)
S3	Invalid - Length is not 6 or not in the range (0 - 999999)	Val	if length(emplid)^=6 or not (0<input(emplid,7.)<=999999)

Filters for Field=Hire Date Type=Date

#	Description	Cat	SAS Code
D1	Invalid - Hire date after May 1, 2001 or before Jan 1, 1930	Val	if hire_dt >mdy(5,1,2001) or hire_dt < mdy(1,1,30)
D2	Missing value	Com	if hire_dt=.
D3	Hire date after termination date, if available	Con	if termination_dt^=. and hire_dt > termination_dt
D4	Employment Hire date not consistent with Job date	Con	if hire_dt ^= PS_JOB.HIR_EFFDT

Filters for Field=Referral Source Type=Character

#	Description	Cat	SAS Code
R1	Value not in lookup table	Val	SD7: sample.LOOKUP
R2	Missing value	Com	if Referral Source=' '

Figure 6

Specifications for FILTERS control	
Field	*Specification*
System	System that the filter audits.
Table	Table that the filter audits.
Field	Field that the filter audits i.e. Character, Numeric, Date.
Type	Type of the field.
Filter CD (#)	Code together with System, Table and Field to uniquely identify the filter.
Filter Description	A text description of the filter
DQ Category	Data quality category of the filter. The categories include: Valid, Complete, Unique, Consistent etc. (See Figure 2)
Active	Indicates whether to include the filter in the current auditing process. Default: No
SAS Code	SAS codes used to define the filter. Code must be provided DQ categories other than Complete or Valid. The code can be one or more lines of SAS statements, or begin with following key words: For DQ category *Valid*, **LIST**: the list of valid codes, or **SD7** : a named SASFILE with valid codes, or **EXCEL**: a named Excel file with valid codes. For any DQ category, **MACRO**: the name of a SAS macro with its parameters.

IMPLEMENTATION PLANNING

Two major streams require coordination in the establishment of a Data Quality Program. These include the establishment of the staffing and process to carry out the DQP, and the technology for the implementation of an automated reporting environment.

A Data Quality Program, like a Data Warehouse, is never completely finished, but changes over time. The best practice for successful implementation includes IT acting as the custodians of the data, with the business units and the organization serving as the owners of quality and accuracy. With upper management attention, this creates a 'hands-on' focus where measuring and improving data quality is a corporate strategy. The DQP then functions to proactively diagnoses and resolves data management issues before they have an adverse effect.

The following are some of the pre-requisite steps needed to establish and sustain a Data Quality Program.

STAFFING AND PROCESS

- Confirm roles and responsibilities for DQP personnel
- Clarify role of DQP Application Support
- Identify personnel who will fill these roles.
- Confirm scope of information (tables and fields) to be included in 1st iteration of DQP
- Establish business rules and define initial filters for major data sources
- Decide on Reporting Frequency
- Perform initial assessment of these elements
- Refine business rules and filters and re-measure
- Establish Reporting Distribution Process

TECHNICAL CONSIDERATIONS

- Confirm Technical Options
- Obtain necessary software components
- Determine Size of Platform for Initial DQP
- Obtain Hardware platform if necessary or identify existing equipment for use
- Configure connectivity for data source access
- Configure programs and options for anticipated environment
- Initiate DQP Processes

CONCLUSION

Many companies are beginning to understand the value of data and the knowledge that it creates as one of their most fundamental assets. The greater prevalence of Data Warehousing is making data available for analysis for more areas of the organization. Few organizations have taken the initiative for making complete data management and quality control of information a strategic initiative.

Organizations need actionable metrics that allow them to quantify improvements for the systems that supply essential information. A Data Quality Scorecard used by all levels of management and the establishment of a Data Quality Program are the best approach to effect change and focus on the issues related to improving the value of information.

Organizations that make data management and improving data quality a strategic initiative benefit in several ways. These include reducing processing costs due to fewer reconciliation activities. Improved data quality provides for increasing sales through better prediction techniques and winning significant contracts through better analysis of data. Greater employee and customer satisfaction through careful and accurate attention to reliable sources of information benefits all levels of an organization.

REFERENCES

PricewaterhouseCoopers LLP, Global Data Management Survey 2001 The new economy is the data economy. 2001.

PricewaterhouseCoopers LLP, Human Resources Data Quality Program – Partner Briefing, March 2001.

ACKNOWLEDGMENTS

Information from PwC internally published documents provided by Mark Miller and Steven Yan was used in the preparation of this paper. Renee Hansen provided additional input and assisted with editing.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Rob Phelps – Robert.W.Phelps@us.pwcglobal.com
Phil Nousak – Phil.Nousak@us.pwcglobal.com

Data Warehousing - Lessons Learned

Fran Akridge , VerizonWireless

Abstract: The panel will outline, in a coordinated effort, lessons learned in starting, growing, and improving data warehouses - both from the IT perspective and from the end user perspective, as well as from management and non-management perspectives. Not surprisingly, lessons learned fifteen or more years ago, when we were calling the warehouse we were trying to design a database, and lessons learned on ongoing basis apply across time. Some lessons are obvious, such as, "Partner, don't compete," and "Give everyone credit." Some are not, such as "Foster broadly based advisory teams and listen, listen, listen." Some are controversial, like "Something is better than nothing." And some are heretical, such as, "SAS won't do it all." The panel is likely to reference SAS ACCESS, PROC SQL, and PROC DBLOAD. It also is likely to address UNIX, Microsoft WINDOWS, and mainframe operating systems. The SAS and the database skill levels addressed range from beginning to advanced, but the lessons are especially addressed to those with leadership and team-member roles in making the warehouse work.

Biotech Warehouse – Stretching the Limits of Columns

Larry Bramblett, Data Warehouse Solutions, LLC
San Ramon, California

Data Warehouse development is often focused on specific data marts that support data analysis for revenue reasons. For most businesses, the ultimate goal is to maximize revenue gain with the focus on the customer as a revenue generator. Companies that have a manufacturing component know that while the customer is important, the improvement of a process that cuts the costs of manufacturing the product also impacts the revenue bottom line. Instead of target marketing, or credit management, process improvement and results analyses are the goals of the manufacturing warehouse. This paper outlines a case study of a Bioengineering company and the journey to create a Metadata driven SAS data warehouse.

The scientists in bioengineering are moving from manual processes and systems to highly automated processes and systems. They are driven by a goal to improve both quality and quantity of products. Currently, there is a mix of both types of systems found. While the intent and design of the new, automated systems was to support the existing process, reality drives the process and the analysis of those systems to produce similar but non-conforming information.

Quality is the key concern which mandates that a "single vision of the data" supports the evaluation and analysis of both manual and automated processes within a single application.

SAS was chosen to manage the collection, conforming and transformation of the data into a single data platform. The vision for the project was to create a data model of the bio engineering process that is independent of the type of process and its location. The goal of the model was also to support US government regulations and the FDA SN88 model. While this project was highly directed around a specific model, the metadata driven process design would support other manufacturing process systems as well.

A considerable amount of time was devoted to the process data model design. Team members that represented various process areas worked together to build the model. The result came from extensive

discussion of each process and each system. The SAS team worked with the client team to provide data as the model was being created, testing each assumption of the model. SAS/Warehouse Administrator provided the platform to extract, transform and create a view of the model that the team could evaluate. During this process it became clear that traditional column and row tables would fail to provide the analysis the end-users, the bioengineering scientists, needed.

The metadata model focused on a specific product and it data collection points. Linkage of the metadata describes the exact point in the manufacturing process that a measurement is collected. In the prototype product the metadata defined over five thousand data point of collected data.

The creation of the metadata driven process drove a view that has many repeating rows with only the lowest level metadata changing along the manufacturing process. They found data types for data points that were random along the product process. Finally, there was a need to analyze at the cell level rather than the column heading. Standard data warehouse table views did not satisfy the need to view the data as a metadata string with its associated collected value. SAS Proc Transpose provided a means to transpose the data from a traditional point of view to one that supported column headings created from the cell level data. This transpose process produced unusable tables of values and associated metadata strings that contained over 5 thousand columns! Created columns headings were less that satisfactory since only 32 bytes could support the transpose and collected data values were still in a ‘text’ state.

The manufacturing model contained metadata that describes each collection point data type in terms of readings, units, date, date time, and so on. Using the metadata data types, actual values could be

transformed but column headings still present a problem.

Multiple levels using SAS Proc Transpose, Arrays and Macros provided the solution. By creating tables that were half metadata string and half transposed, various levels of the manufacturing process could be created and reported on. In the end, a single row that represented the product manufacturing process was created with the correct data type for each collected data point value.

SAS Warehouse Administrator provides the process tool to document, load and transform the metadata and collected data values. Using both generated code and custom SAS procedures; SAS Warehouse Administrator provides the platform that socializes the manufacturing process into a “single vision of the data”.

Larry Bramblett, Data Warehouse Architect
Data Warehouse Solutions, LLC
San Ramon, California

USE OF SAS-ETS AND THE BLS-CENSUS DATA FERRETT FOR THE COMPREHENSIVE EVERGLADES RESTORATION PROGRAM

Richard A. March, South Florida Water Management District, West Palm Beach, Florida

ABSTRACT

This paper describes the proposed use of SAS/ACCESS, the BLS-Census Data FERRETT and SAS/ETS in the analysis of socioeconomic and environmental justice data for the Comprehensive Everglades Restoration Plan http://www.evergladesplan.org,

a co-operative plan between the South Florida Water District and the Jacksonville District of the U. S. Army Corps of Engineers. Particular attention will be paid to the use of the DataFerrett developed jointly by the U. S. Bureau of the Census and the U. S. Bureau of Labor Statistics http://ferret.bls.census.gov

The data ferret is an electronic data review and extraction tool designed to facilitate review and extraction of socioeconomic data from massive BLS and Census Bureau databases. The latest release of the DataFerrett is available in a Beta release in JAVA for use on the World Wide Web. One form in which the DataFerret can output data is as a SAS data set. A SAS data set will be extracted and analyzed using crosstabs and frequencies. An additional analysis relating shrimp harvest data from National Marine Fisheries Service to South Florida Water Management District operational policies using PROC PDLREG in SAS/ETS will also be presented

INTRODUCTION

In this paper, four major subject areas will be discussed: (1) an overview of the Federal statistical resources available online including the Census-BLS Data FERRETT and other resources available online; (2) an overview of data warehousing, with particular emphasis on data warehousing at the South Florida Water Management District and SAS applications at the District ; (3) an overview of the Comprehensive Everglades Restoration Program (CERP) and how SAS is being and will be used in the CERP; (4)an application of SAS-ETS using PROC PDLREG; and (5) summary and conclusions.

FEDERAL STATISTICAL DATA RESOURCES AVAILABLE ONLINE

The BLS-Census Data FERRETT (Federal Electronic Review Retrieval Extraction and Tabulation Tool) is one of a set of data extraction tools available over the Internet for accessing data from large Federal statistical databases. Major participants involved in the Data FERRETT include: (1) the Bureau of Labor Statistics of the U. S. Department of Labor: (2) the Demographic Surveys Division of the Survey Modernization Programming Branch of the Bureau of the Census of the U. S. Department of Commerce; and (3) the Division of Public Health, Surveillance, and Informatics of the Centers for Disease Control.

At the present time, data from the following data sets are available on the FERRETT system:

Current Population Survey (CPS)
- Basic Monthly Survey, January 1994 to present;
- 1992-2000 March Supplement;
- 1996, 1998 and 2000 February Displaced Worker Supplement;
- 1996, 1998 and 2000 February Job Tenure Supplement;
- 1995 May Race and Ethnicity Supplement;
- 1995, 1997, and 1999 February Contingent Worker Supplement;
- 1994-1998 October School Enrollment Supplement;
- 1994, 1996 and 1998 November Voting and Registration Supplement;
- 1999 April Food Security Supplement;
- 1998 August Food Security Supplement;
- 1997 April Food Security Supplement;
- 1996 September Food Security Supplement;
- 1995 April Food Security Supplement;
- 2000 August and 1998 December Internet and Computer Use Supplement;
- 1997 October Computer Ownership/Internet Supplement;
- 1994 October Computer Ownership/Uses Supplement;
- 1998 June Fertility and Birth Expectations Supplement;
- 1995 June Fertility and Marital History Supplement;
- 1997 May Work Schedules Supplement;
- 1995, 1997, 1999 Veterans Supplement;

1997, 1999 American Housing Survey (AHS) - National Survey

1998 American Housing Survey (AHS) - Metropolitan Sample

Survey of Income and Program Participation (SIPP)
- 1992 10 wave longitudinal;
- 1993 9 wave longitudinal;
- 1996 Panel Waves 1-12 Core;
- 1996 Panel Wave 1 Topical Modules;
- 1996 Panel Wave 2 Topical Modules;
- 1996 Panel Wave 3 Topical Modules;
- 1996 Panel Wave 11 Adult Disability Topical Module (FTP only);

1997 Survey of Program Dynamics (SPD);

1993 National Health Interview Survey (NHIS from the National Center for Health Statistics).

1988-1994 National Health and Nutrition Examination Survey III (NHANES from the National Center for Health Statistics).

1994 Mortality - Underlying Cause-of-Death (from the National Center for Health Statistics).
1996 National Ambulatory Care Survey (NAMCS from the National Center for Health Statistics).
1996 National Hospital Ambulatory Care Survey (NHAMCS from the National Center for Health Statistics).

Future data to be placed on the Data FERRETT include:

Population Estimates
County Level Poverty estimates
Economic Census
Fedstats Quick Facts/Mapstats Database
-EPA Data -Crime Data
-Labor Data -U. S. Statistical Abstract
FBI Uniform Crime Data by County
Economic Census
National Hospital Discharge Survey
Multicause of Mortality

The data release schedule for Census 2000 data products is presented at the web site below: http://www.census.gov/population/www/censusdata/c2kproducts.html
In general, first Census 2000 data, Congressional redistricting data, were released in March, 2001 Place and census tract data, including selected population and housing characteristics, is scheduled for release in May and June, 2001. Other Census 2000 data will be released between now and 2003. There is an increasing reliance on the Internet and CD-ROM and DVD media for disseminating Census data. Data not yet on the FERRETT system can be extracted through the Census Quickfacts (http://quickfacts.census.gov/qfd/) system.
Other Census data access tools include: Censtats, Map Stats, TIGER Maps , US Gazetteer, 1990 Decennial Lookup, and Data Extraction System. MAPSTATS is an interactive tool which links the Census Quickfacts database to State and County maps.
Quickfacts also has a non-graphic link that allows the user to extract historic census counts for all counties within an individual state. These data can be downloaded to ASCII files for import into SAS. Another site with a good link between its graphical data and its time-series tabular data is the CLIMVIS (Climatic Visualization) site maintained by the National Climatic Data Center (NCDC) of the National Oceanic and Atmospheric Administration (http://www.ncdc.noaa.gov/onlineprod/drought/xmgr. html) Although the South Florida Water Management District has access to the NCDC data through DBHYDRO, for online acquisition of data for a single climatic region, it is generally just as easy to access that data from the web-site above, unless one requires greater temporal resolution than one-month or greater spatial resolution than the NCDC climate division (there are seven in Florida).
The Census Bureau Website indicates that: “QuickFacts includes Census 2000 counts for all persons plus race groups, Hispanic origin, and persons under 18. Other characteristics asked of everyone will be added later in the summer of 2001. Characteristics asked of a sample of households, like income and education, will become available in 2002. New statistics will appear in American FactFinder shortly before they appear in QuickFacts.“ http://quickfacts.census.gov/qfd/faq.html

An example Census Quickfacts extraction using Florida State totals and Broward County totals is given in Table I below:

Table I : Sample Florida and Broward County Census Data Extracted using Census Quickfacts

People QuickFacts	**Broward County**	**Florida**
Population, 2000	1,623,018	15,982,378
Population, percent change, 1990 to 2000	29.3%	23.5%
White persons, percent, 2000 (a)	70.6%	78.0%
Black or African American persons, percent, 2000 (a)	20.5%	14.6%
American Indian and Alaska Native persons, percent, 2000 (a)	0.2%	0.3%
Asian persons, percent, 2000 (a)	2.3%	1.7%
Native Hawaiian and Other Pacific Islander, percent, 2000 (a)	0.1%	0.1%
Persons reporting some other race, percent, 2000 (a)	3.0%	3.0%
Persons reporting two or more races, percent, 2000	3.4%	2.4%
Persons under 18 years old, percent, 2000	23.6%	22.8%
Persons of Hispanic or Latino origin, percent, 2000 (b)	16.7%	16.8%
High school graduates, persons 25 years and over, 1990	690,696	6,616,094
College graduates, persons 25 years and over, 1990	168,799	1,624,405
Homeownership rate, 1990	68.0%	67.2%
Single family homes, number 1990	277,639	3,368,567
Households, 1990	527,860	5,138,360
Persons per household, 1990	2.35	2.46
Family households, 1990	337,284	3,541,308
Median household money income, 1997 model-based estimate	$37,832	$32,877
Persons below poverty, percent, 1997 model-based estimate	11.7%	14.4%
Children below poverty, percent, 1997 model-based estimate	17.5%	21.8%

Similar data down to the census tract level is available through American Factfinder. (http://factfinder.census.gov/servlet/)
In a press release dated May 29, 1997 the Census Bureau and the Bureau of Labor Statistics announced the release of the Federal Electronic Research and Review Extraction Tool or FERRET. Cavan Capps of the Bureau of the Census and John Bosley of the Bureau of Labor Statistics, along with their staffs are the primary developers of the Data FERRETT. The second t in FERRETT was added when the FERRETT system was integrated into the Data Web (http://www.thedataweb.org/)
On July 31,2000 SAS announced that it had entered into an agreement with the Census Bureau giving the Bureau unlimited access to SAS technology solutions, including any new products created by SAS during the five-year term of the contract. The SAS news release stated, "Besides strengthening the existing relationship between SAS and the Census Bureau, the contract is also the first of its kind for the Census Bureau that is broad enough to include any and all products from a technology vendor."
Data can be accessed directly from the Census Bureau's web site using the 1990 Decennial Lookup, Data Extraction System, and the FERRETT system. The sas transport dataset that comes from FERRETT is created with the export engine and proc copy. Since the sas transport dataset from FERRETT is created with the xport engine and proc copy, only xport & proc copy can be used to convert the transport dataset to a regular SAS dataset. Listed below is the SAS code needed to convert the transport dataset to a regular SAS dataset.

```
libname trans xport "c:\temp\q16363.trn";
libname new "c:\temp";
proc copy in=trans out=new;
run;
```

(The directory and filename listed in the SAS code above are used for example; the above code and description are drawn from the FERRET FAQ website, http://ferret.bls.census.gov/ferret_faq.html).

Other Social Science Data Extraction Tools

A number of other Data Extraction Tools are currently available for accessing 1990 Census of Population Data and other Census Reports. A fairly complete list of "Social Science Data Extractors" is available from the Center for Demography and Ecology at the University of Wisconsin-Madison (http://www.ssc.wisc.edu/cde/datalib/extract.htm)
One of the best of these is the Basic Tables: 1990 Demographic Profile Generator from the Missouri State Census Data Center
http://mcdc2.missouri.edu/websas/xtabs3menus/mo/

"This application is entirely menu driven and requires no code input from the user. The reports generated by this application are all done "on the fly" using a SAS(r) program which is dynamically invoked. This application generates a single 1990 "Basic Tables" (demographic profile) report for any of the supported geographic units, including census tract, block group, city (no size limit), 5-digit ZIP code, state, county or metro area for anywhere in the United States."

Another useful Website for Federal statistical data is Fedstats (http://www.fedstats.gov/). FedStats provides information for Federal agencies reporting expenditures of at least $500,000 per year in one or more statistical activities including:

1.planning of statistical surveys and studies, including project design, sample design and selection, and design of questionnaires, forms, or other techniques of observation and data collection
2.training of statisticians, interviewers, or processing personnel
3.collection, processing, or tabulation of statistical data for publication, dissemination, research, analysis, or program management and evaluation
4.publication or dissemination of statistical data and studies
5.methodological testing or statistical research
6.data analysis
7.forecasts or projections that are published or otherwise made available for government-wide or public use
8.statistical tabulation, dissemination, or publication of data collected by others
9.construction of secondary data series or development of models that are an integral part of generating statistical series or forecasts
10.management or coordination of statistical operations
11.statistical consulting or training

Among the major Federal Agencies with data retrievable on-line through FEDSTATS are:
Bureau of Economic Analysis
Bureau of Justice Statistics
Bureau of Labor Statistics
Bureau of the Census
Bureau of Transportation Statistics
Energy Information Administration
Economic Research Service in the Department of Agriculture
Environmental Protection Agency
Internal Revenue Service, Statistics of Income Division
National Agricultural Statistics Service, U. S. Department of Agriculture
National Center for Education Statistics
National Center for Health Statistics
Science Resources Studies, National Science Foundation
Social Security Administration, Office of Research, Evaluation, and Statistics
Office of Management and Budget

Although not listed as principal statistical agencies by Fedstats, Federal Agencies producing statistical data important in the Comprehensive Everglades Plan include the National Marine Fisheries Service, the U. S. Geological Survey, and the U. S. Army Corps of Engineers, the South Florida Water Management District's principal Federal partner on the Comprehensive Everglades Restoration Plan. The statistical data produced by these agencies are of less interest to the lay user but are critical to the success of the Comprehensive Everglades Restoration Plan. An example using the data extraction capabilities of the NOAA Fisheries Economics website http://www.st.nmfs.gov/st1/commercial/index.html) and SAS/ETS is presented in a later section of this paper.
A useful source of historical census information is the United States Historical Census Browser., maintained at the Fisher Library at the University of Virginia .

(http://fisher.lib.virginia.edu/census/) This site is maintained in co-operation with the Inter-University Consortium for Political and Social Research, a unit of the Institute for Social Research at the University of Michigan. (http://www.icpsr.umich.edu/) Commercial websites, such as Economagic (http://www.economagic.com), offer a variety of time-series data sets for downloading. This site is described: "This page is meant to be a comprehensive site of free, easily available economic time series data useful for economic research,in particular economic forecasting. This site (set of web pages) was started in 1996 to help students in an Applied Forecasting class. The idea was to give students easy access to large amounts of data, and to be able to quickly get charts of that data. This
is also useful during class, so that when we use the computer and overhead projector facility in class, we can quickly retrieve series and do manipulations in class.
At this time, there are more than 100,000 time series for which data and custom charts can be retrieved.

Data Warehouses and Data Warehousing Tools Used at the South Florida Water Management District

According to Webopedia (http://www.webopedia.com/) a Data Warehouse is "A collection of data designed to support management decision making. Data warehouses contain a wide variety of data that present a coherent picture of business conditions at a single point in time. Development of a data warehouse includes development of systems to extract data from operating systems plus installation of a warehouse database system that provides managers flexible access to the data. The term data warehousing generally refers to combine (*sic*.) many different databases across an entire enterprise."
Bill Inmon, in Building the Data Warehouse , defines a data warehouse as: ". . . a subject-oriented, integrated non-volatile, time-variant collection of data in support of management decisions." (Bill Inmon, quoted in Marc Demarest, "A Data Warehouse Evaluation Model," reprinted from. Oracle Technical Journal, October, 1995, v. 1, No. 1, p. 29)
http://www.hevanet.com/demarest/marc/oracle7.html).
Recently, Inmon has suggested the need to move beyond the data warehouse to what he calls "the exploration warehouse." An exploration warehouse is a copy of some or all of the enterprise data warehouse designed specifically for exploration. The exploration warehouse contains detailed data and historical data copied from the enterprise data warehouse. The exploration warehouse is created directly from the enterprise data warehouse.
The exploration warehouse can be created and recreated very quickly should the data miner decide that data is needed in a different manner or that different data is needed. The data miner can use the exploration warehouse as seen fit with no consideration for the performance impact on other users. The data miner is the sole user of the exploration warehouse so there is no conflict with resource utilization with other warehouse analysts.
Cavan Capps of the U. S. Bureau of the Census, one of the primary developers of the Data FERRETT and the Data Web has stated: "In the past, data have been system-specific. Data sets have been inextricably bound to the hardware and software systems that house them, and to the agencies that administer them. Access to data has depended upon access to the machines or the software systems that house the data. Recent technological developments make it possible to break this bind and allow for wider access to data, regardless of differences in underlying hardware and software, and regardless of organizational boundaries."(Cavan Capps U.S. Bureau of the Census The Data Web and FERRETT: Innovations in Integrating Distributed Federal, State, and Local Data,")
http://www.spc.uchicago.edu/datalib/ia2000/prog/capps.txt
The South Florida Water Management District answered the question, "What is a Data Warehouse?" as follows: "Data Warehouse describes a collection of operational data from many sources that has been summarized and reconciled and is presented to the decision-maker in insightful ways. The stored information derives from day-to-day operational data and has been combined, formatted and optimized in ways conducive to analytical thinking with the purpose of providing organizational decision-makers with timely information necessary to effectively make critical business solutions. The decision-maker looks at the gathered information in many ways in order to understand the critical underlying issues and trends facing the business. Analytical thinking happens through spontaneous probing, by asking questions until a problem is understood. The benefit of the Data Warehouse comes in the ability to make effective decisions from it. Cost of the Data Warehouse is reflected in its design, implementation, execution, refinement and maintenance. What end users notice most is both the quality and reliability of the data and the usefulness of the tools used to view it."
http://cobweb2.sfwmd.gov/iwebB501/pln/proj/warehouse/warehouse.htm
The data warehouse, as envisioned by District staff and management would include, but not be limited to, Projects, GIS Data , Surface Water Modeling Data , Groundwater Modeling Data, Census Data , Budget Information, and Personnel Information. Each of these areas has its own set of independent data management, modeling, analysis, and presentation tools. One of the major concepts being advanced in the CERP Data Management Plan (DMP), currently under development is a "data model." The CERP team in the kick-off statement for development of the DMP stated: "Our goal is to create a Data Model. Data modeling is the process of defining the grammar, vocabulary, and content to be used to represent information in a database system." DataModel.org has defined a Data Model as: "Basically, a datamodel is any method of visualizing the informational needs of a system and typically takes the form of an ERD (Entity Relationship Diagram)." (http://www.datamodel.org**)** The SAS/EIS software and SAS/MDDB Data Server software have not yet been implemented at SFWMD.
At the present time, the primary "corporate" hydrologic and water database in use at the South Florida Water Management District is an ORACLE database known as DBHYDRO. A menu-driven Oracle Forms 4.5 application HYDRO_PREP is used to define the data to be retrieved from DBHYDRO. Separate databases are maintained by different organizational units within the agency for GIS Data (in ARC-INFO), Real Estate Data (Land Acquisition Management Information System (LAMIS) in ORACLE),

Permit Data Regulatory Database ((REGDB) in ORACLE), ROSS Human Resources and Payroll System. Recently, the Water Management and the Corps of Engineers made the decision to use the Oracle 8i System as the Central CERP Data Warehouse (the CERP Zone),

"In the 1990s SAS caught up with the rest of the computer world going from a programming style language, to a graphic and menu oriented, user friendly package. The program was also rewritten in C. (Presentation by R. Thomas James to South Florida Water Management District Unix Users' Group, August 28, 1995)." In March, 1999 the South Florida Water Management District completed its IT Strategic Plan, FY 2000-2003. The District IT Strategic Plan has several Key Findings relating to Data Warehousing and Integration, including:

"The District lacks an organized information structure that permits easy access to all District data, information, and knowledge and provides integrated tools to analyze and synthesize data. Currently District data is housed in isolated systems that are generally not accessible to one another or by end users. Many different tools on many different platforms are used to analyze and synthesize these data. The resulting reports are often generated in incompatible formats that cannot be easily distributed. "Information in each operational system is often not readily available to District Employees, as well as to the public. As a consequence, data is often duplicated and extra systems are created. System users must know and use multiple processes to access and update the same data . . .. "It is acknowledged that a certain level of compatibility is necessary to ensure that each tool can meet the needs of all District users. However, needs also exist that must be met by very specific solutions. Therefore, a level of 'service utility' needs to be defined. These utilities include network interfaces/protocols, database systems, operating systems and platform standards. Beyond that point, customization can occur to meet the specific needs without diverging from the standards that allow for the integration. Significant evidence of integration includes

Ease of data sharing

Compatibility with the defined service utility layers of the infrastructure.

Web interface. . . .

District employees constantly deal with integration issues when they exchange information with other organizations. Integration issues occur when the District or the other agency:

Have a different version of the software

Utilize a different vendor's software

Utilize different media types

Utilize different data standards"

(South Florida Water Management District, I. T. Strategic Plan, pp. Key Findings-6-11).

The I. T. Strategic Plan has identified a strategy, "Encourage adherence to industry standard software, utilize a lowest common denominator standard when using dissimilar types for exchanging, and develop a knowledge base of exchange methodologies among organizations and publish it on the District's Web (South Florida Water Management District, I. T. Strategic Plan, p. Key Findings-11). Even though the main Data Warehouse for CERP will be an Oracle 9i System much of the data will be exchanged with SAS applications. Through SAS/ACCESS Interface to Relational Databases.

The Comprehensive Everglades Restoration Program

The Comprehensive Everglades Restoration Program is a joint program between the U./// S. Army Corps of Engineers and the South Florida Water Management District. The Comprehensive Everglades Restoration Program is authorized under Section 601 of the Water Resources Development Act of 2000 (WRDA 2000). This act provides, in part,

"(A) IN GENERAL- Except as modified by this section, the Plan is approved as a framework for modifications and operational changes to the Central and Southern Florida Project that are needed to restore, preserve, and protect the South Florida ecosystem while providing for other water-related needs of the region, including water supply and flood protection. The Plan shall be implemented to ensure the protection of water quality in, the reduction of the loss of fresh water from, and the improvement of the environment of the South Florida ecosystem and to achieve and maintain the benefits to the natural system and human environment described in the Plan, and required pursuant to this section, for as long as the project is authorized."

On May 12, 2000 a Design Agreement between the Department of the Army and the South Florida Water Management District for the Design of Elements of the Comprehensive Plan for the Everglades and South Florida Ecosystem Restoration Project (CERP) was signed. This Design Agreement, among other things, mandates the preparation of a Master Program Management Plan (MPMP). This Master Program Management Plan contains an Appendix F, discussing Programmatic Activities. Programmatic activities are defined as " . . . activities and tasks that are not linked to a specific project, but involve or affect more than one project or the entire restoration program. These activities include Restoration Coordination and Verification (RECOVER), public outreach, socioeconomic and environmental justice studies, program management and technical co-ordination activities." (Master Program Management Plan (http://www.evergladesplan.org/pm/docs/mpmp_08182000/MPMP%20Appendix%20F%20-%20Final%20000818.pdf))

Environmental justice, a particular concern of the CERP, is mandated by Executive Order 12898 signed by President Clinton on February 11,1994. This Order states, in part, "To the greatest extent practicable and permitted by law, and consistent with the principles set forth In the report on the National Performance Review, each Federal agency shall make achieving environmental justice part of its mission by identifying and addressing, as appropriate, disproportionately high and adverse human health or environmental effects of its programs, policies, and activities on minority populations and low-income populations." Census data, particularly small-area studies, play an important part in targeting CERP-related activities to identify and address environmental justice issues.

The Water Resources Development Act of 2000, the Federal Legislation authorizing much of the work under the CERP specifically addresses environmental justice concerns as shown below.

OUTREACH AND ASSISTANCE-

(1) SMALL BUSINESS CONCERNS OWNED AND OPERATED BY SOCIALLY AND ECONOMICALLY DISADVANTAGED INDIVIDUALS- In executing the Plan, the Secretary shall ensure that small business concerns owned and controlled by socially and economically disadvantaged individuals are provided opportunities to participate under section 15(g) of the Small Business Act (15 U.S.C. 644(g)).

(2) COMMUNITY OUTREACH AND EDUCATION-

(A) IN GENERAL- The Secretary shall ensure that impacts on socially and economically disadvantaged individuals, including individuals with limited English proficiency, and communities are considered during implementation of the Plan, and that such individuals have opportunities to review and comment on its implementation.

(B) PROVISION OF OPPORTUNITIES- The Secretary shall ensure, to the maximum extent practicable,that public outreach and educational opportunities are provided, during implementation of the Plan, to the individuals of South Florida, including individuals with limited English proficiency, and in particular for socially and economically disadvantaged communities"

.

Example Using SAS/ETS and NMFS Data-

Data on monthly West Coast of Florida pink shrimp landings were obtained as an ASCII file from Guy Davenport of the National Marine Fisheries Service Miami Office. Data covering the period May, 1966 to August, 1998 was analyzed. The NMFS data extraction tool could not be used to generate the data set because monthly landings before 1990 are not on the system. The NMFS ASCII file was imported, using the import Wizard into SAS, and a SAS data set "shrimppounds.sas" was set up.
The NMFS data was augmented with monthly flow data (maximum flow and average flow data for the Taylor Slough Bridge located at the mouth of the Caloosahatchee River, near Fort Myers) from the District's DBHYDRO database. Harvest data was obtained by water bodies; however, for the purposes of this presentation only the aggregate West Coast of Florida data is presented.
A polynomial distributed lag model using the iterated Yule-Walker (method= ITYW) autocorrelation correction method is estimated. The iterated Yule _Walker method is described as: follows:

Let $\boldsymbol{\varphi}$ represent the vector of autoregressive parameters
$\boldsymbol{\varphi} = (\varphi_1, \varphi_2, \varphi_3, \ldots, \varphi_m)'$
and let the variance matrix of the error vector
$\nu=(\nu_1, \ldots, \nu_N)'$ be Σ
$E(\nu\nu') = \boldsymbol{\Sigma} = \sigma^2 V$.

If the vector of autoregressive parameters φ is known, the matrix **V** can be computed from the autoregressive parameters. $\boldsymbol{\Sigma}$ is then $\sigma^2 V$. Given $\boldsymbol{\Sigma}$, the efficient estimates of regression parameters can be computed using generalized least squares(GLS). The GLS estimates then yield the unbiased estimate of the variance , σ^2.

The Yule-Walker method alternates estimation of β using generalized least squares with estimation of φ using the Yule-Walker equations applied to the sample autocorrelation function. The YW method starts by forming the OLS estimate of β
. Next, φ is estimated from the sample autocorrelation function of the OLS residuals using the Yule-Walker equations. Then **V** is estimated from the estimate of φ , and is estimated from **V** and the OLS estimate of σ^2. The autocorrelation corrected estimates of the regression parameters β are then computed by GLS using the estimated Σ matrix. These are the Yule-Walker estimates.
If the ITER option is specified, the Yule-Walker residuals are used to form a new sample autocorrelation function, the new autocorrelation function is used to form a new estimate of φ and **V**, and the GLS estimates are recomputed using the new variance matrix. This alternation of estimates continues until either the maximum change in the φ-hat estimate between iterations is less than the value specified by the CONVERGE= option or the maximum number of allowed iterations is reached. This produces the Iterated Yule-Walker estimates. Iteration of the estimates may not yield much improvement.
The Yule-Walker equations, solved to obtain φ-hat and a preliminary estimate of σ^2, are R φ-hat=-r.

This discussion, along with a discussion of alternative autocorrelation correction methods, is in the on-line SAS/ETS User's Manual at the link below.
http://calcul.si.uji.es/Programes/SAS/ets/chap8/sect21.htm

Because pink shrimp harvest is, by definition, non-negative, the regression was forced though the origin using the noint option, as shown in the SAS code below.

```
proc pdlreg data=shrimppounds;
model all=sinterm costerm real_price aveflow(12,2,1) maxflow(12,2,1)/noint NLAG=12 METHOD=ITYW;
output out=shrimp P=predict R=resid;
run;
```

The variables included in the above model statement are:
All = monthly pink shrimp landings in the West Coast of Florida (Gulf) region, pounds;
Time= a time-trend variable equal to 1 in May, 1966 and increasing one unit per month thereafter;
Sinterm= SINE((2π/12)*Time);
Costerm =COSINE((2π/12)*Time);
Real_price = (Pink shrimp revenue/Pink shrimp landings)/CPI(1982-84 base), ($/pound)
The CPI data was retrieved from the BLS web site:
http://stats.bls.gov/cpihome.htm

Aveflow= average monthly flow (cfs) through the S-79 water control structure, as reported by the South Florida Water Management District;
Maxflow= peak flow (cfs) during a particular month through the S-79 water control structure, as reported by the South Florida Water Management District

The pink shrimp landings and revenue data will periodically be updated from the NMFS web site: http://www.st.nmfs.gov/st1/commercial/landings/monthly_landings.html

"Florida is divided into three areas (East coast, West coast, and inland waters) in our surveys. Florida east coast and inland water landings are summarized with South Atlantic states and Florida west coast landings are summarized when Gulf landings are requested." http://www.st.nmfs.gov/webplcomm/plsql/webst1.ft.help.species_

The terms sinterm and costerm are included to capture the seasonality of pink shrimp landings. The underlying theory is based on spectral analysis, which is explained in the SAS/ETS User's Guide under PROC SPECTRA at the link below.
http://calcul.si.uji.es/Programes/SAS/ets/chap17/sect1.htm
The model estimation results are shown below.

The SAS System 13:29 Thursday, April 12, 2001

The PDLREG Procedure

Dependent Variable All

Ordinary Least Squares Estimates

SSE	3.67472E14	DFE	367
MSE	1.00129E12	Root MSE	1000643
SBC	11501.0457	AIC	11465.6794
Regress R-Square	0.8199	Total R-Square	0.8199
Durbin-Watson	0.6403		

NOTE: No intercept term is used. R-squares are redefined.

Variable	DF	Standard Estimate	Approx Error	t Value	Pr |t|
sinterm	1	618053	86585	7.14	<.0001
costerm	1	-1079908	85167	-12.68	<.0001
real_price	1	301966	43690	6.91	<.0001
Aveflow**0	1	-182.6894	41.9764	-4.35	<.0001
Aveflow**1	1	-140.5207	47.8809	-2.93	0.0035
Aveflow**2	1	-5.4397	47.4891	-0.11	0.9089
Maxflow**0	1	146.7356	23.3197	6.29	<.0001
Maxflow**1	1	70.6262	30.1774	2.34	0.0198
Maxflow**2	1	16.9913	30.4533	0.56	0.5772

Yule-Walker Estimates

SSE	1.65816E14	DFE	355
MSE	4.67087E11	Root MSE	683438
SBC	11280.5936	AIC	11194.1427
Regress R-Square	0.3123	Total R-Square	0.9187

NOTE: No intercept term is used. R-squares are redefined.

Autoregressive parameters assumed given.

Variable	DF	Standard Estimate	Approx Error	t Value	Pr >|t|
TIME	1	7070	1606	4.40	<.0001
costerm	1	-1040981	122991	-8.46	<.0001
sinterm	1	619130	125400	4.94	<.0001
real_price	1	47085	81588	0.58	0.5642
Aveflow**0	1	8.5493	61.0774	0.14	0.8888
Aveflow**1	1	-97.2570	45.5037	-2.14	0.0333
Aveflow**2	0	0	.	.	.
Maxflow**0	1	4.2912	36.5003	0.12	0.9065
Maxflow**1	1	53.0982	28.0952	1.89	0.0596
Maxflow**2	1	6.5254	13.3478	0.49	0.6252

Additional output from PROC PDLREG and simulated and forecast values are available from the author upon request.

The model estimated above can be used to model simulated impacts to West Coast of Florida pink shrimp landings from changes in exogenous variables including water flows, real price, and seasonality. A number of the scenarios being examined for the Comprehensive Everglades Restoration Plan will alter flows through the S-79 water control structure. The results show that there is a strong seasonal pattern to pink shrimp landings, and that increases in average flows tend to increase pink shrimp landings, while increases in maximum daily flows, representing flushes of fresh water, tend to decrease landings. All of these effects take place with a lagged effect. It is hypothesized that this reflects the spawning and growth cycles cycle of pink shrimp. Increases in real prices are associated with increased landings; this relationship needs to be further investigated through an econometric model that examines both supply-related factors and demand-related factors.

Conclusions

The BLS and Census Bureau Data FERRETT system and other data extraction tools are useful in generating SAS data sets (and ASCII files for import into SAS and other statistical applications) for statistical and econometric analysis. The Data FERRETT is better suited, at the present time, to extracting cross-sectional and recent data; BLS and the Census Bureau have other data extraction tools better suited to generating long-time series (pre-1990). Other Federal, state, and regional agencies, such as NMFS and the South Florida Water Management District, generally have usable databases, although these agencies, particularly the Water Management District, have made less progress in making their data readily accessible than have BLS and the Census Bureau.
A major challenge to all agencies in building data warehouses will be appropriately integrating spatially disaggregated data (such as TIGER) files and time-series data sets with a long period of record and/or a short time-step. A number of sites, including National Climatic Data Center, are developing point-and-click data access tools, like MAPSTATS, which are integrated with a geographical information system.

Data Quality – Spinning Straw Into Gold

Bob Brauer, DataFlux Corporation™, Cary, NC

DATA QUALITY TODAY

The issue of data quality is a simple one. As IT spending soars and organizations continue to spend a large portion of their IT budgets on Business Intelligence applications such as data warehousing, customer-relationship management, data mining, marketing automation, and sales force automation, the importance of the underlying source of data that feeds these applications has become increasingly higher. After all, well into the Information Age, organizations have become entirely data-driven. Rare is the employee who does not come into daily contact with an element of information originating from a company's various databases. Critical business decisions and allocation of resources are made based upon what is found in the data. Prices are changed, marketing campaigns created, customers are communicated with, and daily operations evolve around whatever data points are churned out by an organization's various systems. In other words, companies are living and dying as a result of the information contained within their data.

THE ISSUE OF DATA QUALITY

This is hardly a distressing affair. After all, what we yearn for from our information systems is more efficiency and increased effectiveness in every facet of the organization, and that is precisely what these business intelligence systems can deliver, with one small catch. The data that serves as the foundation of these systems *must be good data*. Otherwise, we fail before we ever begin. It doesn't matter how pretty the screens are, how intuitive the interfaces are, how high the performance rockets, how automated the processes are, how innovative the methodology is and how far-reaching the access to the system is, *if the data is bad - the systems fail*. Period. And if the systems fail, or at the very least provide inaccurate information, every process, decision, resource allocation, communication, or interaction with the system will have a damaging, if not disastrous, impact on the business itself. Worst of all, all the money and resources spent within IT on the development and deployment of these Business Intelligence systems will go down the proverbial toilet, and can negatively effect, sometimes severely, the operations and success of the company we all dedicate our lives to on a daily basis.

WHY ARE DATA QUALITY ISSUES SO RAMPANT?

Nothing discussed up to this point falls short of obvious. While it is certainly true that data quality is often overlooked in the race for glory and on-time delivery during the development of information systems, rare is the individual who claims that data quality is not an important issue and *should* be addressed. However, rarer still seems to be the individual who strives to actually make it a high priority issue during systems development, and something that is part of the agenda of every status meeting that discusses the success of a particular information system. This can be quantified by a report from the META Group indicating that 75% of companies in the United States have yet to implement *any kind of data quality* procedures to their systems development lifecycles. Of those that have, the majority report that the current implementations fall far short of what the organization requires. This is a phenomenon worth considering closely.

There are several reasons why the issue of data quality often gets swept under the carpet during the design, development, and deployment of information systems. While there are certainly others, three primary reasons will be discussed here. They are *attractiveness of the subject matter, the cost and difficulty in implementing data quality, and the inability to demonstrate ROI.*

The first reason to tackle is *attractiveness of the subject matter*. Data Quality is right up there in frequency with Cobol performance-tuning and fax-machine maintenance as a career objective among today's IT professionals. Most engineers are interested in the latest and greatest hot technologies and areas such as graphical interface and screen development and all the aesthetic arts involved. They also tend to have an interest in applications programming and all of the latest technologies surrounding it such as Java, Visual Basic, Cold Fusion, and all of the algorithmic challenges that solutions development offers. And of course, the Internet and its many flavors of technological know-how also tend to seduce the more ambitious among us engineers from what we consider to be the mundane tasks of data quality. After all, it is a safe bet that most IT professionals can count the number of people who consider themselves to be a "data quality engineer" on one hand. This is an unfortunate circumstance as some of the most challenging algorithms that the art of computer science has to offer can be found in the practice and implementation of data quality to a problem set that is more often than not unique from organization to organization. Also, as more and more exciting technologies begin to emerge in this area as we are seeing now, the momentum of the science should also increase.

Perhaps it is the word "quality" that causes many to cringe when the term is discussed, and another term ought to be interchanged with it to broaden its appeal. "Director of Data Management", "Data Effectiveness Researcher", and "Data Infrastructure Engineer" would be among the list of candidates that executive management may want to consider when assigning titles to what we all agree is a very important issue within the organization.

A second major reason that Data Quality is often ignored is because of the general consensus that the *implementation of data quality procedures and tools is a costly and resource-intensive undertaking*. While it is certainly not disputed that the science of data quality can and should extend beyond software technology, there are a great deal of software solutions that are available in the industry that will take an organization a long way in overcoming their data quality challenges, and enable it to enjoy the fruits and savor the success that these applications can deliver. Unfortunately, these solutions have traditionally been very expensive (in the six and seven figure ranges), very difficult to work with (engineers from the vendor as customization consultants long-term are the norm), encumbering to implement (data typically has been required to be exported off-line before even simple quality analysis and other various data quality functions can occur), and therefore have had very long implementation cycles (often 18 months). Doing the math clearly demonstrates that these kinds of solutions have traditionally been well out of reach financially from all but a very select few, and even more so for those of us who do not have the luxury of multi-year development schedules.

Fortunately, this is dramatically changing, and data quality technologies are emerging that eliminate practically all of the aforementioned barriers to introducing data quality into the development life cycle, as well as practically anywhere else along the information technology landscape. It is important to point out that since barriers such as high cost of entry, usability, lack of piece-wise availability, and time of implementation are being eliminated, the idea of high Data Quality is permeating out of the enterprise IT departments. It is now found departmentally, on a project-by-project basis, and even in non-IT departments such as marketing, sales, and operations. This has allowed the technology to be utilized not only on the largest DB2 driven mainframe-oriented systems, but all the way down to Microsoft Access databases and at every platform step along the way. In fact, there are now free data quality analysis tools available to any organization that can be implemented in minutes on practically every platform. This can help an organization ascertain to what extent they may be facing data quality issues. Giving these free-of-charge analysis tools an opportunity to provide an information system a thorough data quality checkup is a no-brainer. Vendors have discovered that demonstrating data quality issues easily and quickly via technology are a pre-cursor to a customer acquiring further technology from the vendor that can resolve many of these same issues. SAS® and DataFlux have pioneered a great deal in this area and these innovations are now currently available.

This is a good lead-in to the third primary reason that data quality is often overlooked, and that is the *inability to demonstrate clearly the return-on-investment of data quality technologies*. Since there are many different flavors and aspects of data quality, there are differing degrees to the extent of this situation. For example, undoubtedly one can easily calculate postage saved in a marketing database that has a duplicate record level of 30% of the records (the same individual or organization appearing multiple times within a dataset under different variations of their name and address) if those duplicates can be discovered and removed. However, it is often not so clear what dollar amount can be placed on data that is inaccurate or otherwise missing from a dataset, or to what degree data inconsistency within a dataset can affect the results of queries and reports that are used as the basis of decision-making, and what the financial impact is of poor decisions that are made as a result of faulty or inaccurate information. Since this is often the case and ROI analysis of data quality procedures are far from a science, persuasion of management and spending authorities can be a difficult challenge when resources are doled out from the top of the mountain.

Interestingly enough, data quality lends credence to the phenomenon that the most obvious and blatant solutions are often those most overlooked. Sometimes exercises in data quality impact analysis produce ROI figures that are so astronomical that they are simply dismissed as absurd.

To demonstrate this, let's consider the example of a typical banking institution.

Research has shown that the average banking customer contributes $3000 per year to his or her bank (the culmination of mortgage interest, auto loans, banking fees, securities management, etc.), or $15,000 over a five-year period. Suppose a bank with a million customers implements data quality technology across the enterprise enabling a greater success in all data-driven activities such as customer interaction, marketing personalization, cross-selling, and accurate business analysis results in an increase of a mere 2% of new business. The resulting impact numbers can be astounding. This would be the equivalent of adding 20,000 new customers, multiplied by $15,000 over a five-year period, or $300 million dollars. And that is at a paltry 2%. It should be clear that while this is a simplified analysis, similar more-detailed analysis numbers that produce similar results often leave executives scratching their heads and thinking, "What have I missed here?" Usually, the answer is nothing.

THE RISKS AND COSTS OF NON-QUALITY DATA

The previous example is indicative of some of the processes that can reap huge benefits to an organization via data quality technology. However, inattention to data quality does not just result in missed opportunities, it can leave broad risk exposure and countless shortfalls from a data quality perspective, including lost dollars, lost customers, and even legal ramifications. The following are some examples that demonstrate the great pains that can be suffered if we continue to avoid data quality as a management priority.

- In a conservative estimate, more than 175,000 IRS and state tax refund checks were marked as 'undeliverable' by the post office last year.
- Three zeros inadvertently added and reported as a trade volume amount of an inside executive of a public company in Atlanta caused its stock to plummet over 30% in one day.

- After an audit, it was estimated that 15-20% of voters on voter registration lists have either moved or are deceased when compared to data gathered from post office relocation data.
- An acquiring company learned long after the deal was closed that their new consumer business only had ½ the customers as they thought because of the large presence of duplicate data in the customer database.
- A fiber-optics company lost $500,000 after a mislabeled shipment caused the wrong cable to be laid at the bottom of a lake.
- A mailing order company faced massive litigation because it was unable to catch bogus, insulting names from being entered into the catalog request section of their website that were ultimately mailed to unsuspecting and then humiliated receivers of the catalog.
- The US government estimates that billions of dollars are lost annually due to poor data quality.

It is staggering. Estimates have shown that 15-20% of data within an organization's databases can be erroneous or otherwise unusable, leading to an enormous effect on the bottom line.

The issue is compounding to a greater extent in the age of the Internet. We now have less control over the data collection mechanisms that feed our information systems such as website forms or B2B XML channels. These external data suppliers are growing more common by the day. Also, we have a much higher level of access to corporate data via the Internet by our customers, demanding a greater degree of accuracy still.

A CLOSER LOOK AT ACTUAL DATA QUALITY ISSUES

So far, we have discussed data quality and its ramifications in broad strokes. Now, the discussion turns to the kinds of basic data quality issues that IT professionals run into regularly, and how these can affect the results of the business intelligence applications that we rely on daily.

INCONSISTENTLY REPRESENTED DATA

Unfortunately, data can be ambiguously represented. This fact often is positioned at the very root of an organization's data quality issues. If multiple permutations of a piece of data exist within a dataset, then every query or summation report generated by the dataset must account for each and every instance of these multiple permutations. Otherwise, important data points can be missed and can severely impact the output of these processes.

For example, a company name can be represented a multitude of ways:

IBM, Int. Business Machines, I.B.M., ibm, Intl Bus Machines

As can a business title:

VP Sales, Vice President Sales, V.P. Sales, Director of Sales, VP SALES

Or an operating system:

Windows NT, WINDOWS, WIN NT, Windows NT 5, Win NT 5.0

They all have the same meaning, but are represented very differently. It is obvious to surmise what kinds of analytical problems can and will arise if the same data is dissimilarly represented within a dataset as these examples demonstrate.

Imagine a life insurance company wanting to determine the top ten companies that their policyholders work for in a given geographic region in order to tailor policies to those specific companies. Inaccurate aggregation results are likely because of all the permutations of data for a given company name will be difficult to account for.

Imagine a marketing campaign that personalizes its communication based upon the business title of the prospect. Variations in business titles can have a nightmare effect on these types of focused campaigns, and can cause improper personalization or too many generic communication pieces to be generated, wasting dollars on both material production and creative efforts of the group.

User base platform analysis by a software company could produce improper results if the data looks like it does in the platform example cited.

While these are simple data inconsistency examples, these and other similar situations are endemic to databases worldwide. Fortunately, data quality technology now exists that identifies these various permutations of data and can rectify the situation a number of ways, including physically standardizing the data within the dataset, creating synonym tables/filters, or correcting undesired permutations before they enter the dataset in the first place.

DUPLICATE DATA

Another common example of a data quality issue is duplicate/redundant data. Again, because data can be ambiguously represented, the same customer, prospect, part, item for sale, transaction, or other important data could be occurring multiple times. In cases like these, the redundancy can only be determined by looking across multiple fields.

The following are examples of duplicate data that cannot be caught without some form of data quality technology (or else long, endless hours of human inspection, unlikely to catch as high of a percentage, and impossible with anything more than small volumes):

Robert Smith, 100 E Johnson Street

Bob Smythe, 100 East Johnson
Dr. Robert J. Smith, 100 E. Johnston St.

Ms. Kathleen Anderson, Box 12 – 9 Canary Street
Katie Andersen, 9 Canary St. #12

Large Camping Knife
Knife, Camping Lg.

The Briggs Corporation, Saint Louis
Brigs Corp, St. Louis

Problems that can arise from redundant data within a dataset include inaccurate analysis, increased marketing/mailing costs, customer annoyance, and relationship breakdown across a relational system. Again, as data such as this serves as the foundation and infrastructure of our business intelligence systems, it is imperative that these situations be identified and snuffed out in order to achieve success.

DATA INTEGRATION

One of the close sisters of duplicate data is the issue of data integration. This becomes a data quality issue when the columns that constitute the join fields between multiple datasets may contain data that is inconsistently represented. For example, trying to combine a customer table with an outside demographic data source will have undesirable results if the join column is a column commonly containing ambiguous representations of data such as company name:

Data Source A (Customer Dataset)
Columns: Customer Name, Contact
Data: First Bank of Denver, Joe Snow

Data Source B (Demographics)
Columns: Company Key, Num Employees, Business Type, Annual Sales
Data: The 1st Bank of Denver, 850, Financial, $62 million

Obviously a standard SQL join statement would not recognize that these two banks are the same and therefore the demographic data would not be joined to the customer data.

One way to achieve a join that would indeed succeed in this scenario is by using a match code that *unambiguously* represents the company name. Data quality algorithms can be used to generate this unambiguous code. The code itself might be represented by something covert, such as **RX19E4**, however the same code will be generated when any permutation of the "First Bank of Denver" is passed through the match code generation algorithm. This unambiguous code then becomes the basis of the match between data sources, and can be constructed using any number and combination of columns. These codes can be stored as an extra column in each data source, stored in a temporary table or file, or generated solely at runtime.

While data integration may not be considered a "quality" problem by some, the same types of algorithms and procedures apply that can achieve much higher match rates and therefore much better success when combining data from multiple sources. Often, these integrated datasets form the basis from which many business intelligence applications thrive.

DATA VERIFICATION AND AUGMENTATION

One final area of data quality that will be discussed here concerns the area of resolving missing and/or inaccurate data by using an external data reference. This includes not only filling in missing values and replacing inaccurate values, but also adding additional data values to a record or data observation that provides a more complete picture of the entity that is being stored in the dataset. A common example of this is using the United States Postal Service's master address database to verify and/or correct existing addresses within a database, as well as append other useful demographic postal data such as Zip+4, carrier routes, congressional districts, counties, delivery points, etc. This can greatly increase address integrity, as well as provide a basis for additional applications such as geocoding, mapping, and other visualization technologies that require a valid address as a starting point. Obviously, technology such as this can go a long way as an integral part of a business intelligence application.

OTHER DATA QUALITY ISSUES

In addition to the data quality issues discussed here, there are many others that should be given careful consideration by any organization who has invested in business intelligence applications. These include but are not limited to data structural incompatibilities, missing values, numerical data quality issues (can be identified using techniques of statistical analysis), cross-column-based correctness analysis (such as determination of correct gender from name data), and a whole lot more. In the aggregate, all of these issues if overlooked can become extremely hazardous to the health of a data-driven information system.

THE IMPERATIVE OF ASSESSING DATA QUALITY

Now that we have established a firm understanding of what Data Quality is, as well as its impact, it is now time to turn our thoughts to what can be done about it. As the old adage declares, every great journey begins with the first step. In the case of data quality, this first step is *data quality assessment*. The fact that a data quality assessment step is missing from the development plans of many project leaders is not only dumbfounding, it truly ought to be grounds for dismissal. Every system that has data relies on that data heavily (or else the data wouldn't be there) for its success, and anybody who eliminates data quality from the development process is acting irresponsibly. This may have been understandable in the past as analysis tools were not readily available to assess data quality, but now not only are they available, they are often available for free and require very little time to utilize. Now that we understand the impact of poor data quality, inattention to it is inexcusable.

In addition to the analysis and exception-finding tools available for this purpose, data quality assessment should at the very least consist of the following questions:

- How do we know our data isn't bad?
- What would constitute "bad data" from our perspective?
- If it were bad, what kind of effects might it have on our various systems?
- What points of data collection might cause data quality issues to occur?
- If we had a more accurate data foundation, what could we then accomplish?
- Where specifically are our costs of data imperfections high?
- What non-technology processes could be introduced to increase our level of data quality?
- What technology is available to assist us in our efforts?
- What quantifiable ways can we demonstrate ROI to executive management?

Attention to these questions as well as ones that pertain specifically to a certain type of system or business intelligence application will shed much light on how these issues can be contained if not resolved entirely. A meticulous data quality assessment will undoubtedly provide the blueprint for a solid data quality solution.

WHAT SAS AND DATAFLUX ARE DOING ABOUT DATA QUALITY

On June 9, 2000 SAS acquired the DataFlux Corporation in an effort to enhance the data infrastructure solutions currently offered to its customers, and underscore the importance of data quality as a cornerstone of any data-driven initiative.

DataFlux has developed many award-winning data quality technologies that are used throughout multiple industries today. These technologies tackle many of the issues that were discussed throughout this paper and many of those not discussed here, as well as providing easy to implement, cost-free data quality assessment as also emphasized herein. SAS has integrated some of these technologies currently, and will continue to integrate many more into SAS products such as the Data Warehouse Administrator, as well continuing to offer the DataFlux product line to the market.

More specifically, the DataFlux offering includes a suite of data quality and data integration tools that can assist significantly in the development of a bulletproof data foundation integral to any data-driven business intelligence endeavor. dfPower™ Studio 4.0 is a comprehensive data quality and data integration solution that focuses on many data quality issues such as standardization, matching, data verification, de-duplication, data integration, accuracy, and data quality business rule management. These technologies are delivered via an intuitive interface, and are packaged with many other enabling technologies such as database access, providing an easy-to-use and easy-to-implement multi-faceted data quality solution.

For developers, DataFlux also provides its Blue Fusion™ SDK, a series of functional libraries that contain many of the algorithms that comprise the core of dfPower Studio 4.0. The covers have been taken off and the algorithms delivered as components, allowing developers to build data quality directly into any application.

The two offerings also work very well together as a completely integrated solution, providing the ability to include data quality and better data integration at practically every point within the organization.

A GOLDEN OPPORTUNITY

In conclusion, it is very clear that data quality processes can have tremendous impact on the bottom line of an organization. Now that the available technologies in this area are gaining momentum and improving immensely with every new release date, the effects of poor data quality are no longer inescapable. Well thought-out data quality assessment combined with state-of-the-art data quality technology will take an organization a long way in achieving its goals of accuracy, consistency, usability, and completeness of all of its electronic data assets. An understanding and facilitation of these concepts will continue to be a bellwether for business intelligence success.

REFERENCES

English, Larry P. 1999. Improving Data Warehouse and Business Information Quality – Methods for reducing costs and increasing profits. Jon Wiley and Sons, Inc. New York, NY.

META Group, Inc. Stamford, CT

The author may be contacted at:

Bob Brauer
DataFlux Corporation
4001 Weston Parkway, Suite 300
Cary, North Carolina 27513
(919) 674-2153
Fax: (919) 678-8330
Email: bobb@dataflux.com
Web: www.dataflux.com

Paper #P108

Using the SAS ACCESS Engine for DB2 OS/390 to Bulk Load Tables

Robert E. Maitland Jr., BlueCross BlueShield of Florida, Jacksonville, Florida
Tom Weber, SAS Institute, Cary, North Carolina
George Bischoff, Bank of America, Richmond, Virginia

ABSTRACT

The Bank of America Enterprise Data Warehouse resides in a DB2 OS/390 environment. The optimal method for joining a SAS data set with a DB2 table is to convert the SAS data set to a DB2 table. SAS version 8.2 supports bulk loading capabilities (within the SAS ACCESS engine for DB2 OS/390) by utilizing the IBM stored procedure DSNUTILS. Normally, data is inserted into a DB2 table one row at a time. Now, data can be bulk loaded into the table in a fraction of CPU time. Furthermore, this utility provides additional functionality. For example, DB2 tables can now be runstat from within SAS.

INTRODUCTION

The primary sources of data for many SAS users reside in data warehouses and data marts. Many of these data warehouses and data marts are built in relational data base management systems (rdbms) that are not SAS. SAS ACCESS provides a link for SAS users to utilize the power of the rdbms to process their data before using SAS to analyze it. With version 8, SAS has enhanced this power through the SAS ACCESS LIBNAME engines. This paper will focus on using the SAS bulk load capability to load a SAS data set into a DB2 table on OS/390 via the SAS ACCESS LIBNAME DB2 engine. First, we will provide a brief discussion of why users need to create tables in the rdbms and some of the factors they must take into consideration. Then, we will compare the methodology between the bulk load and non-bulk load technique. Next, we will provide an example using the bulk load.

USER TABLES IN AN RDBMS

Data warehouses tend to be large. Data marts are built from data warehouses to provide a higher level of performance for a line of business(s). They have fewer columns and may have business rules/processing applied before loading. The data mart can be on a platform or an rdbms structure different from the warehouse. Users can do a high percentage of their work via the data mart but not all. Therefore, they have a need to go back to the warehouse to obtain information not provided in the data mart. Furthermore, there is still a need to be able to handle data from external sources. The tables in the warehouse tend to be large versus the data sets that users need to join to them. Therefore, the optimal method of joining a SAS data set to an rdbms table is to convert the SAS table to an rdbms table.

When creating tables in an rdbms, there are some performance considerations users need to be aware of. First, the index/keys of the table(s) in the rdbms should be duplicated on the user table. They do not have to have the same name, but the same columns should be on both tables. These columns must be stored in the same format For example, if one of the keys on the rdbms table is in small integer, then the corresponding column on the user table needs to be in small integer. Creating that column in decimal will degrade performance. Next, the user needs to create an index on the table with the columns that replicate the index/keys on the rdbms table(s). The user then needs to run the program that calculates the statistics for the user table and index. These steps are required for the rdbms to use the index(s) of the tables in the rdbms.

THE ADVANTAGES OF BULK LOAD

In version 6, SAS provided the DBLOAD procedure to load tables in DB2. With version 8, users can create tables using the LIBNAME ACCESS ENGINE DB2. Both of these methods use SQL in the background to create the tables in DB2 and load the tables with SQL inserts. That is, the tables are loaded one row at a time. (Of course users can always use pass through SQL to generate the create table and SQL insert statements themselves.) This process is slow. Furthermore, DB2 writes a line to the log for each row inserted. This can cause the log file to over flow. System performance can be degraded. Also, users have to leave SAS to run the DB2 utility that computes the statistics on the user table.

Version 6 of DB2 provides the capability of calling a utility through a stored procedure. SAS version 8.2 calls the DSNUTILS procedure to perform the bulk load. Large amounts of data can then be loaded in the user table in a fraction of CPU time. Furthermore, DSNUTILS has a great deal of functionality. Users have access to that functionality through SAS. They now have more control over the parameters. For example, they can turn the log off. More importantly, the runstats utility can be called through DSNUTILS. Users no longer have to leave SAS to call the runstats utility.

THE DSNUTILS STORED PROCEDURE

When the DSNUTILS procedure is called, SAS generates the files needed for the utility:

- SYSIN
- SYSREC
- SYSPRINT.

The control statement is generated in the SYSIN file. This file contains the instructions on how to load the table and the structure of the physical sequential file used to load the table: the SYSREC file. The SYSPRINT file stores the results of the utility. By default, the SYSPRINT file is also written to the SAS LOG.

There are a number of ways and options available when calling DSNUTILS via SAS (see Weber). For this paper, we are going to focus on the process of:

- Creating the table
- Creating the files needed to load the table.
- Creating the index.
- Loading the table.
- Doing the runstats.

The data can be found in the SASHELP library. It is the company data.

CREATING THE TABLE

The DB2 table is created using a SAS data step and specifying a LIBNAME statement with the DB2 engine. This is the part of the data step that generates the DB2 table:

```
LIBNAME MYTABLE DB2 IN =
'DATABASE.TABLESPACE';

DATA MYTABLE.TEST_TABLE(
DBTYPE=(DEPTHEAD='CHAR(15)'
        JOB1='CHAR(15)'
        LEVEL1='CHAR(16)'
        LEVEL2='CHAR(13)'
        LEVEL3='CHAR(20)'
        LEVEL4='CHAR(30)'
        LEVEL5='CHAR(30)'
        N='SMALLINT')
.
.
SET TEST_DATA;
RUN;
```

The LIBNAME statement invokes the DB2 engine. Table's created/read will have the same owner/creator as the user id. To access different tables use the AUTHID option (see SAS OnLine Documentation).

The DB2 data base and table space that the table will be loaded in are contained in the IN option.

The column names and types for the DB2 table will match the variable names and types in SAS data set found in the SET statement (TEST_DATA). The DB2 default for character variables is VCHAR. The default for numeric values is FLOAT. Using the DBTYPE option can overwrite the default values. For example, N will be created as SMALLINT in the DB2 table.

INVOKING DSNUTILS AND CREATING THE FILES NEEDED TO LOAD THE TABLE

Now we will add the options to the data step statement that will invoke the DSNUTILS utility and create the files it needs. However, we are not ready to load the table because we have not created an index. We need to runstat the table and the index. The only way of creating the index in DB2 is through the SQL pass through. We cannot create the index until we have created the table. Parameters are available which allow us to create the files and run them later.

```
LIBNAME MYTABLE DB2 IN =
'DATABASE.TABLESPACE';

DATA MYTABLE.TEST_TABLE(
DBTYPE=(DEPTHEAD='CHAR(15)'
        JOB1='CHAR(15)'
        LEVEL1='CHAR(16)'
        LEVEL2='CHAR(13)'
        LEVEL3='CHAR(20)'
        LEVEL4='CHAR(30)'
        LEVEL5='CHAR(30)'
        N='SMALLINT')
DB2LDUTIL=YES DB2LDEXT=GENONLY
DB2LDCT1='RESUME NO
          LOG NO
          STATISTICS TABLE(ALL)
                     INDEX(ALL)'
DB2IN='USERID.NAME.SYSIN'
DB2REC='USERID.NAME.SYSREC'
DB2RECSP=1000 DB2SPC1=1000 DB2SPC2=1000);
SET TEST_DATA;
RUN;
```

The DB2LDUTIL option calls the DSNUTILS procedure. The default is no.

The DB2LDEXT=GENONLY causes the files to be created but keeps the utility from being invoked. The default value is GENRUN, where the files are created and the utility is invoked.

```
The DB2LDCT1='RESUME NO
             LOG NO
             STATISTICS TABLE(ALL)
                        INDEX(ALL) '
```

statement passes these DB2 options to the SYSIN file. An alternative to RESUME NO is REPLACE; but, be careful. The REPLACE parameter causes everything in the DB2 database/tablespace to be replaced. The LOG NO turns off the log. The STATISTICS TABLE(ALL) INDEX(ALL) parameters call the runstats utility.

The DB2IN and DB2REC options are used to give names to the SYSIN and SYSREC files. If names are not given then the files are generated using the user id as the first node and system generated information for subsequent nodes. Since we need to call these files in a later step, we passed the system data set names.

The next three parameters are used to increase the file space:

- DB2RECSP=1000
- DB2SPC1=1000
- DB2SPC2=1000

CREATING THE INDEX

The index is created using pass through SQL and DB2 parameters.

```
PROC SQL;
 CONNECT TO DB2(SSID=XXXX);
 EXECUTE(
  CREATE INDEX TEST ON USERID.TEST_TABLE
(LEVEL1,LEVEL2)
  USING STOGROUP SMSSG1
        PRIQTY 72000
        SECQTY 18000
        ERASE NO
        FREEPAGE 0
        PCTFREE 0
        BUFFERPOOL BP2
        CLOSE NO)
  BY DB2;
  %PUT &SQLXMSG;
 DISCONNECT FROM DB2;
QUIT;
```

LOADING THE TABLE AND INVOKING RUNSTATS

Now we will use the files created in the first step to load the table and invoke the run stats utility.

```
DATA  MYTABLE.TEST_TABLE(DB2LDUTIL=YES
DB2TBLXST=YES
DB2LDEXT=USERUN
DB2IN='USERID.NAME.SYSIN'
DB2REC='USERID.NAME.SYSREC');
 V=0;
RUN;
```

The DB2TBLXST=YES option tells SAS that the table already exists. The default is NO. DSNUTILS loads an existing table. It does not matter when the table is created.

The DB2LDEXT is now set to USERUN. This tells SAS to invoke the load utility and use files previously created. These files are specified with the DB2IN and DB2REC parameters. The call to RUNSTATS is in the SYSIN file.

Note, there is no set statement. All the information we need to load the table is in the DATA statement. However, we need something to prime the pump. Therefore, we use the assignment statement: V=0. Since V is not a column in the table, it is ignored in the processing.

SUMMARY

Version 6 of DB2 provides the capability of calling a utility through a stored procedure. SAS version 8.2 calls the DSNUTILS procedure. DSNUTILS has a great deal of functionality. We have provided an example that utilizes some of that functionality to bulk load a SAS data set into a DB2 table and do the run stats through SAS.

REFERENCES

IBM Inc. (1999), *DB2 UDB for OS/390 V6 Utility Guide and Reference*, IBM Inc.

SAS Institute Inc. (1999), *"SAS ACCESS Engine DB2", SAS Version 8 Online Documentation*, Cary, NC: SAS Institute Inc.

Weber, Tom, *Load Utility Interface,*Unpublished White Paper, SAS Institute Inc. (2000), Cary, NC: SAS Institute Inc.

ACKNOWLEDGMENTS

Special thanks to Hoang Le for assistance.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Robert E. Maitland Jr.
BlueCross BlueShield of Florida
Enterprise Business Intelligence Tools Support
4800 Deerwood Campus Parkway DCC3-2
Jacksonville, Fl. 32246

Phone: 904-905-1401
EMAIL: bob.maitland@bcbsfl.com

APPENDIX

```
FILENAME OUT1 'C:\SSU Paper\SASHELP_COMPANY_DATA';

DATA TEST_DATA;
 INFILE OUT1;
 INPUT
 DEPTHEAD $Char15.
 JOB1     $Char15.
 LEVEL1   $Char16.
 LEVEL2   $Char13.
 LEVEL3   $Char20.
 LEVEL4   $Char30.
 LEVEL5   $Char30.
 N        $2.;
RUN;

LIBNAME MYTABLE DB2 IN = 'DATABASE.TABLESPACE';

DATA MYTABLE.TEST_TABLE(DBTYPE=(DEPTHEAD='CHAR(15)'
                                JOB1='CHAR(15)'
                                LEVEL1='CHAR(16)'
                                LEVEL2='CHAR(13)'
                                LEVEL3='CHAR(20)'
                                LEVEL4='CHAR(30)'
                                LEVEL5='CHAR(30)'
                                N='SMALLINT')
     DB2LDUTIL=YES DB2LDEXT=GENONLY
     DB2LDCT1='RESUME NO
               LOG NO
               STATISTICS TABLE(ALL)
               INDEX(ALL)'
     DB2IN='USERID.NAME.SYSIN'
     DB2REC='USERID.NAME.SYSREC'
     DB2RECSP=1000 DB2SPC1=1000 DB2SPC2=1000);
 SET TEST_DATA;
RUN;

PROC SQL;
 CONNECT TO DB2(SSID=XXXX);
 EXECUTE(
  CREATE INDEX TEST ON USERID.TEST_TABLE (LEVEL1,LEVEL2)
  USING STOGROUP SMSSG1
        PRIQTY 72000
        SECQTY 18000
        ERASE NO
        FREEPAGE 0
        PCTFREE 0
        BUFFERPOOL BP2
        CLOSE NO)
  BY DB2;
  %PUT &SQLXMSG;
 DISCONNECT FROM DB2;
QUIT;

DATA  MYTABLE.TEST_TABLE(DB2LDUTIL=YES DB2TBLXST=YES
                         DB2LDEXT=USERUN
                         DB2IN='USERID.NAME.SYSIN'
                         DB2REC='USERID.NAME.SYSREC');
 V=0;
RUN;
```

Using the SAS/ACCESS® Libname Technology to Get Improvements in Performance and Optimizations in SAS/SQL Queries

Fred Levine, SAS® Institute, Cary NC

ABSTRACT

This paper highlights the new features in the SAS/ACCESS libname engines that, when used judiciously, can improve overall engine scalability in the areas of loading/extraction, ASYNC I/O, and SQL-based query optimizations.

The new loading/extraction engine features are:

- Multi-row reads
- DBKEY
- Bulk loading
- Multi-row writes

The new ASYNC I/O features are:

- PreFetch
- SAS server task switching

The new SQL-based query optimization features are:

- Implicit SQL-Passthru
- WHERE clause optimizations

Some of the above features are new in 8.2 whereas others existed in prior releases but have been enhanced in 8.2.

INTRODUCTION

All of the SAS/ACCESS libname features discussed in this paper are designed to improve scalability in specific processing environments that warrant such improvements. The more knowledge you have of your underlying data and how your DBMS server and network are tuned to process that data, the more knowledge you will have to be able to make wise decisions about when to use these features to your advantage.

The examples used in this paper come from various SAS/ACCESS libname engines. Not all features have been implemented in all SAS/ACCESS engines. Please see the SAS/ACCESS documentation for engine-specific information where applicable.

Loading/extraction features

The features described below are designed to improve engine performance in the area of loading and extracting data. They are implemented as libname and/or dataset options and are easy to use.

MULTI-ROW READS

The ability to extract your data as fast as possible is of paramount importance. SAS/ACCESS libname engines accomplish this internally by exploiting native API-controlled multi-row read capabilities of the underlying DBMS. To activate this feature, you specify the number of rows returned in a single read operation via libname and/or dataset options. The following example uses SAS/ACCESS to Oracle to specify 1000 rows per read operation using the READBUFF dataset option:

```
libname myora oracle user=scott pw=tiger
        readbuff=1000;
```

All tables from the schema defined by the above libname statement will be read in 1000 row blocks. This feature can improve performance for very large tables when your DBMS server and network are optimally tuned (it should be noted that setting the TCP Packet size on your network can make a big difference in performance.)

The SAS/ACCESS libname engines that support API-controlled multi-row reads are Oracle, Oracle Rdb, DB2/Unix, Sybase, ODBC, and OLE/DB. Please note that not all DBMSs support API-controlled multi-row reads although they often provide this feature transparently. For further engine-specific information about multi-row reads, please see the SAS/ACCESS documentation.

DBKEY

The DBKEY data set option is used to improve performance when joining a very small SAS data set to a large DBMS table. Please note that if this feature is not used prudently it will not improve performance, and in some cases can actually degrade performance. The following conditions must be met to see performance improvements with DBKEY:

- a very small transaction SAS data set
- a large master DBMS table

In addition, performance is often improved when DBKEY is used in conjunction with the DBNULLKEYS data set option (see the DBNULLKEYS section below for details).

HOW DBKEY WORKS

When you join a large DBMS table and a small SAS data set, the DBKEY option enables you to retrieve only the required subset of the DBMS data into SAS for the join. If you do not specify this option, SAS reads the entire DBMS data into SAS and then processes the join.

Using DBKEY can decrease performance when the SAS data set is too large. This is because DBKEY causes each value in the SAS transaction data set to generate a new result set (or open cursor) from the DBMS table. For example, if your SAS data set has 100 rows with unique key values, you request 100 result sets from the DBMS which can be very expensive. You must determine whether using this option is appropriate, or whether you can achieve better performance by reading the entire DBMS table (or creating a subset of that table).

Internally, the SAS/ACCESS libname engine generates a WHERE clause of the form:

```
where column = host-variable
```

Note that 'column' is what was specified on the DBKEY= option. The *host-variable* takes the value of 'column' from each row in the SAS data set and generates a separate DBMS result set for each of these values. As a result, only rows in the DBMS table that match this WHERE clause are retrieved. Without DBKEY, the above WHERE clause would not get internally generated

forcing SAS to read all the rows from the DBMS table before processing the join.

USING THE DBNULLKEYS OPTION WITH DBKEY

The DBNULLKEYS data set option is used in conjunction with DBKEY and has a direct effect on the internally generated WHERE clause described above. If there is NULL data in your DBMS table, then the generated WHERE clause will account for NULLs as follows:

```
where ((column = host-variable)
  OR ((column IS NULL) AND (? IS NULL)))
```

This WHERE clause generated the extra NULL conditions in response to DBNULLKEYS=YES which tells the SAS/ACCESS libname engine that the DBKEY column from the DBMS table contains NULL data. However, when your DBKEY column from the DBMS table does not contain NULL data, you should specify DBNULLKEYS=NO which causes the SAS/ACCESS libname engine to generate a simpler form of the WHERE clause without the check for NULLs:

```
where ((column = host-variable)
```

It should be noted that when DBNULLKEYS=YES, the more complicated WHERE clause that checks for NULLs has the potential to be much less efficient than the simpler form of the WHERE clause. This is why DBKEY works best for DBMS tables that do not contain NULL data. Consider the following example:

```
/* SMALL SAS DATA SET */
5? data saslib.small;
6?   do id=1 to 1000000 by 100000;
       smalltext='Fra small'; output; end;
       run;
7?
NOTE: The data set SASLIB.SMALL has 10
      observations and 2 variables.


/* LARGE ORACLE TABLE */
33? data oralib.master;
34?   do id=1 to 200000;
    text='Master table';output; end;run;
35?
NOTE: The data set ORALIB.MASTER has 200000
observations and 2 variables.
```

Neither the SAS table nor the Oracle table contain NULL data.

The following data step join uses DBKEY with DBNULLKEYS=YES :

```
11? data temp;
  set saslib.small;
  set oralib.master (dbkey=id
dbnullkeys=yes) key=dbkey ;
run;

NOTE: There were 10 observations read from
the data set SASLIB.SMALL.
NOTE: There were 10 observations read from
the data set ORALIB.MASTER.
NOTE: The data set WORK.TEMP has 10
observations and 3 variables.
NOTE: DATA statement used:
      real time           20.55 seconds
      cpu time            0.03 seconds
```

The following data step join uses DBKEY with DBNULLKEYS=NO:

```
38? data temp;
  set saslib.small;
  set oralib.master (dbkey=id  dbnullkeys=no)
key=dbkey ;run;
 40?
NOTE: There were 10 observations read from
the data set SASLIB.SMALL.
NOTE: There were 10 observations read from
the data set ORALIB.MASTER.
NOTE: The data set WORK.TEMP has 10
observations and 3 variables.
NOTE: DATA statement used:
      real time           0.03 seconds
      cpu time            0.03 seconds
```

Notice the difference in real time. When DBNULLKEYS=YES the extra NULL conditions on the generated WHERE clause caused this job to take 20.55 seconds. When DBNULLKEYS=NO, yielding the simpler WHERE clause, the real time dropped to a mere .03 seconds.

The SAS/ACCESS libname engines will also check to see if a DBKEY column is NON-nullable, and if so, generate the simpler, more efficient WHERE clause regardless of the DBNULLKEYS value. However, if the DBKEY column was not originally declared as NON-nullable but you do not have any NULL data for that column, then you will need to set DBNULLKEYS=NO to reap the maximum performance benefits of DBKEY.

Please see the SAS/ACCESS documentation for further information about DBKEY.

BULK LOADING

The purpose of bulk loading is to provide the highest possible load performance utilizing native DBMS load utilities. The ability to exploit native load utilities has huge performance implications when loading a large data warehouse. The SAS/ACCESS libname engines allow you to easily invoke these native load extensions via libname and/or dataset options.

Below is an example of bulk load syntax using SAS/ACCESS to Teradata:

```
libname mytera teradata database=john
   user=john pw=doe bulkload=yes;
```

The above syntax tells the Teradata engine to use the native FastLoad to insert rows into tables scoped to the connection defined by the above libname statement. There are many other options that are used in conjunction with BULKLOAD=YES. See the SAS/ACCESS documentation for further information on BULKLOAD options.

Note that DBMS-specific bulk load facilities are not transactional in that they do not use programmatic SQL insert statements to load data. Rather, they cause the data from the input data set to be bulk copied as a unit to the DBMS table. As a result, error conditions behave differently under bulk load, i.e., no rollbacks are issued.

In addition to bulk copying to empty DBMS tables, some bulk loaders also allow appending rows to existing DBMS tables.

We have compared loading a modest size 32,000 row SAS dataset into an Oracle table using both the native Oracle SQL*Loader and standard SQL inserts. We have observed that Oracle's SQL*Loader was 3 times faster than using conventional SQL inserts. If a modest size table can produce these

performance improvements, you could expect to see much greater performance gains with a very large table.

It should be noted that using a DBMS's native bulk loader can also impede performance for very small tables due to the processing overhead of setting up the loader. It is therefore recommended that bulk loading be reserved for larger tables. You may need to experiment using bulk load with different size tables to determine how many rows are required to yield meaningful performance gains.

The SAS/ACCESS engines that support native bulk loading are Oracle, DB2/Unix, DB2/MVS, Sybase, Teradata, ODBC, and OLE/DB.

Please see the SAS/ACCESS documentation for further information about bulk loading.

MULTI-ROW WRITES

Although native bulk loading provides the fastest possible load performance, it is also possible to get faster load performance using conventional transactional processing. SAS/ACCESS libname engines allow you to do this by exploiting native API-controlled multi-row write capabilities of the underlying DBMS. To activate this feature, you specify the number of rows to be inserted for a single write operation via libname and/or dataset options. These options allow you to insert multiple rows at a time by specifying the number of rows to be inserted as a unit. The following example uses SAS/ACCESS to DB2/Unix to specify 100 rows per write operation using the INSERTBUFF dataset option:

```
libname mydb2 db2 database=sample user=john
       using=doe insertbuff=100;
```

The above syntax instructs the DB2/Unix engine to insert 100 rows at a time when loading data into tables scoped to the connection defined by the above libname statement.

It should be noted that multi-row writes have an impact on error handling since errors are associated with buffers rather than with individual rows. Errors are therefore not discovered until a later point in the processing.

The optimal value for multi-row write options such as INSERTBUFF vary with factors such as network type and available memory. You may need to experiment with different values to determine the best value for your site.

The SAS/ACCESS libname engines that support API-controlled multi-row writes are Oracle, DB2/Unix, ODBC, and Oracle Rdb. Just like multi-row reads, not all DBMSs support API-controlled multi-row writes.

For further information about multi-row writes see the SAS/ACCESS documentation.

ASYNC I/O features

ASYNC I/O is the area of processing that allows SAS/ACCESS libname engines to exploit asynchronous execution of calls into the underlying DBMS for the purpose of improving performance and optimizing client requests in a SAS server environment. In general terms, asynchronous execution refers to events that are not coordinated in time, such as starting the next operation before the current one is completed.

PREFETCH

PreFetch is a SAS/ACCESS facility that can speed up a multi-step SAS job by exploiting the asynchronous processing capabilities of an underlying DBMS. The SAS job must be read-only to use this facility, that is, any SAS statement that creates, updates, or deletes DBMS tables would not be a candidate for PreFetch. It should also be noted that at the time of this writing, PreFetch is only supported by SAS/ACCESS to Teradata.

When reading tables, SAS/ACCESS programmatically submits DBMS-specific SQL statements on your behalf to the DBMS. Each of these SQL statements has an execution cost. When PreFetch is enabled, the first time you run your SAS job SAS/ACCESS will identify those SQL statements with a high execution cost and store them in a DBMS-defined macro. On subsequent runs of the SAS job, SAS/ACCESS will extract the stored SQL statements from this macro and submit them in advance to the DBMS which will "prefetch" the rows selected by these stored SQL statements. It should be noted that PreFetch improves elapsed time only on subsequent runs of a SAS job since on the first run, SAS/ACCESS merely stores the selected SQL statements for subsequent use. For this reason, PreFetch should be used only for static SAS jobs that are run frequently.

Below is an example of using the SAS/ACCESS to Teradata PreFetch facility:

```
libname mytera teradata database=john
      user=john pw=doe prefetch='tr_store1';

proc print data=mytera.emp where emp.salary >
    100000;
proc print data=mytera.sales where
    sales.commission > 0;
proc print data=mytera.sales where
    sales.product = 'truck';
proc print data=mytera.newsales;run;
```

The first time you submit the above job SAS/ACCESS to Teradata will create a 'tr_store1' macro to store the SQL statements associated with the above proc prints if it is determined that they have a high execution cost.

For subsequent runs of this job, you need only specify:

```
libname mytera teradata database=john
      user=john pw=doe prefetch='tr_store1';
```

SAS/ACCESS to Teradata will now "prefetch" the rows associated with the stored SQL statements by utilizing the native asynchronous processing capabilities of Teradata. PreFetch can also be specified as a SAS global option.

For further information see the SAS/ACCESS to Teradata documentation.

SAS SERVER TASK SWITCHING

SAS server task switching is a mechanism designed to maximize multi-client throughput in a concurrent SAS server environment. This SAS server environment can either be a SAS/SHARE server or a SAS Integrated Object Model (IOM) server (please see SAS documentation for more information on SAS/SHARE® and IOM).

The basic idea behind task switching in a SAS server environment is that a SAS/ACCESS libname engine running on a SAS server must not impede response time for other clients while the engine waits for a lengthy DBMS operation to complete for a specific client. By default, when the engine is processing a request for a specific client other client requests are suspended since the SAS server does not implement time-slicing.

To get around this potential problem, SAS/ACCESS has

implemented a SAS invocation option 'DBSRVTP' that enables a SAS/ACCESS libname engine to voluntarily give up control of a client task to another client task for targeted DBMS operations in a SAS server environment. The primary benefit of this task-switching mechanism is to allow the engine in the SAS server to respond to many different clients more efficiently.

It should be noted that the individual DBMS operations that enable task-switching may vary from engine to engine due to DBMS-specific differences in the execution time of these operations.

Below is an example of invoking SAS in a server environment using the 'DBSRVTP' option to enable task-switching using SAS/ACCESS to Sybase:

```
sas –dbsrvtp sybase
```

Multiple engines can also be specified. The example below invokes SAS using the 'DBSRVTP' option to enable task-switching using SAS/ACCESS to Sybase, ODBC, and Informix:

```
sas –dbsrvtp '(sybase ODBC informix)'
```

The SAS/ACCESS libname engines that support SAS server task switching are DB2/Unix, Informix, ODBC, OLE/DB, Oracle, Sybase, and Teradata. Please see the SAS/ACCESS documentation for further information on the DBSRVTP option.

SQL-based query optimizations

SQL query optimizations is a performance improvement feature that transparently offloads SQL processing that normally would occur in SAS to the underlying DBMS. There are two contexts in which these transparent SQL optimizations occur:

- Proc SQL
- WHERE clauses surfaced from any SAS procedure

The facility that allows this to happen in proc SQL is called Implicit SQL Passthru. The facility that allows this to happen in SAS WHERE clauses is referred to as the WHERE optimizer.

IMPLICIT SQL PASSTHRU

Implicit SQL Passthru (hereafter referred to as Implicit Passthru) is a proc SQL feature that, for performance-sensitive SQL operations, will transparently convert your proc SQL query into a DBMS-specific SQL query and directly pass this converted query to the DBMS for processing. This mechansim has several advantages over traditional passthru:

1. You often get similar performance improvements to traditional passthru while having your queries seamlessly integrated into SAS.

2. If your site uses more than one SAS/ACCESS libname engine, you need only be familiar with SAS SQL syntax to get performance improvements for multiple DBMS engines.

3. Java and MFC-based thin client applications that submit generated SAS SQL to a SAS server (such as Enterprise Guide) will transparently get the performance benefits of Implicit Passthru on the SAS server without having to be cognizant of DBMS-specific SQL syntax.

For a proc SQL query to be a candidate for Implicit Passthru, it must reference a single SAS/ACCESS engine libref and contain one or more of the following:

- the DISTINCT keyword
- JOINS (inner and outer)
- SQL aggregate functions
- UNIONS

The above SQL operations have been targeted as key performance-sensitive operations that can often be processed faster by a DBMS.

It should also be noted that, beginning in release 8.2 of SAS (and beyond), SAS/ACCESS libname engines support a subset of SAS functions for which underlying DBMSs have equivalents. This means that supported SAS functions in proc SQL queries will get translated into their DBMS equivalents and will be passed along with the query to the DBMS.

Note that SAS functions do not by themselves trigger a pass-through. Instead, a query must be a candidate for pass-through as listed in the above bulleted list. Once the query is a candidate, then any supported SAS function for which the DBMS engine has an equivalent will simply be passed along with the rest of the query.

DISTINCT PROCESSING

The DISTINCT keyword triggers Implicit Passthru since in many cases the result set will be much smaller than the initial size of the table. By offloading this processing to a DBMS server, only the result set rows get transmitted across the network back to proc SQL, hence greatly reducing network time when the number of rows in the result set is much smaller than the number of rows in the table. Consider the following example using SAS/ACCESS to Ingres:

```
proc sql;
  libname ing ingres database=clifftop;
  select distinct state from ing.creditcard;
```

In the above example, a user wants to determine how many states are represented by a group of credit card customers. The result set could have a maximum of 50 rows. If this table contained 5 million rows, then you can quickly see the advantage of passing this query to Ingres for processing. The advantage is that transmitting a maximum of 50 rows across the network is much faster than proc SQL having to read all 5 million rows into SAS and then do its own DISTINCT processing.

PASSING DOWN JOINS

JOINS are another SQL operation that can normally be processed more efficiently by the DBMS when the result sets are much smaller than the input tables. Since in SAS SQL it is possible to join as many as 32 tables, you can quickly see the benefit of letting the DBMS do the processing since the performance cost of transmitting all rows from all join tables into SAS for processing can be formidable. Implicit Passthru can pass both INNER and OUTER joins to the DBMS for processing.

INNER JOINS

Passing INNER join queries to a DBMS is straight forward since all datasources support ANSI 1992 INNER join syntax.

The following INNER join query uses SAS/ACCESS to Informix and will get passed to Informix for processing:

```
proc sql;
  libname nfx informix user=john pw=doe
      database=rockbridge server=server1;

  select * from nfx.cust, nfx.sales,
                nfx.orders where cust.custnum
```

```
                    = orders.ordernum and
                      sales.salesrep = 'SMITH';
```

OUTER JOINS

Passing down OUTER join queries is more complicated than passing down INNER joins since some datasources do not support ANSI 1992 OUTER join syntax. The datasources that have non-standard OUTER join syntax are Oracle, Sybase, Informix, and ODBC.

The Implicit Passthru facility has the capability to pass down OUTER join queries for all SAS/ACCESS libname engines, regardless of whether they support ANSI 1992 OUTER join syntax. For ANSI-compliant data sources, passing down OUTER join queries is straight forward with no restrictions. For those engines whose underlying datasources support non-standard OUTER join syntax, Implicit Passthru will convert SAS SQL ANSI-compliant OUTER join syntax to the non-standard datasource-specific OUTER join syntax, although with some restrictions (see below).

The following OUTER join query exemplifies this conversion to non-standard syntax using SAS/ACCESS to Oracle:

```
proc sql;
  select * from eng.JOIN11 left join
   eng.JOIN22 on JOIN11.x=JOIN22.x right join
     eng.JOIN33 on JOIN11.x=JOIN33.x;
```

Implicit Passthru will convert the above SAS ANSI-compliant OUTER join text to:

```
select JOIN11."X", JOIN22."X", JOIN33."X"
        from JOIN11, JOIN22, JOIN33
          where JOIN11."X" (+) = JOIN33."X"
            and JOIN11."X" = JOIN22."X" (+)
```

in keeping with Oracle-specific OUTER join syntax which uses the ' + ' operator in the WHERE clause to tag non-preserved tables in OUTER join queries.

RESTRICTIONS ON NON-ANSI OUTER JOIN SYNTAX

Although Implicit Passthru can generate non-standard datasource-specific OUTER join syntax for those datasources that require it (Oracle, Sybase, Informix, and ODBC), there are some restrictions.

1. For queries that use SAS/ACCESS to Sybase, any OUTER join that references more than two tables AND contains a WHERE clause will not get passed. ***Note that more than two table OUTER joins will get passed without a WHERE clause, as will a two table OUTER join with a WHERE clause.*** This restriction exists since Sybase can return different results than SAS when a WHERE clause is applied to an OUTER join with more than two tables.

2. For queries that use SAS/ACCESS to Informix, only two table OUTER joins will get passed. Informix uses a WHERE clause in lieu of an ON clause which makes the integration of ON clauses and WHERE clauses from a SAS SQL query difficult to convert into an Informix-specific query.

3. For queries that use SAS/ACCESS to ODBC, more than two table OUTER joins are supported as long as there are no INNER joins specified in that same query. This restriction exists due to limitations in ODBC OUTER join syntax.

4. Oracle, Sybase, and Informix do not support FULL OUTER joins.

It should be noted that in spite of the restrictions listed above, in many cases these restrictions will not be an issue and you should often be able to reap performance benefits from these non-standard datasources.

SQL AGGREGATE FUNCTIONS

SQL aggregate functions represent another area of SQL operations that are processed more efficiently by a DBMS. This is because typically, they search a table and perform behind-the-scenes calculations, yielding a single row result set. These SQL aggregate functions include:

- MIN
- MAX
- AVG
- SUM
- COUNT

Below is an example of passing down a query that contains an SQL aggregate function using SAS/ACCESS to Oracle:

```
libname ora oracle user=john pw=doe;
proc sql;
   select count(*) from ora.employees;
```

UNIONS

UNIONS will also get passed to the DBMS. Since typically a UNION eliminates duplicate rows, this processing is more efficient when passed to a DBMS.

Below is an example of a UNION using SAS/ACCESS to DB2/Unix:

```
libname mydb2 db2 database=sample user=john
  using=doe;
proc sql;
 select * from mydb2.music_titles
     union
 select * from mydb2.discontinued_CDs;
```

PARTS OF A QUERY CAN GET PASSED DOWN

In the event that a query that gets passed down to a DBMS results in a failure returned from the SAS/ACCESS engine, proc SQL will attempt to pass down a simpler version of that query. Any portions of the query that cannot be handled by the DBMS are handled by proc SQL [Church 1999].

Below is an example of a part of a query getting passed down using SAS/ACCESS to Sybase. It was mentioned above that there is a restriction on Sybase queries where more than two tables in an OUTER join cannot be specified with a WHERE clause due to differences in the way Sybase evaluates these queries. The query below contains a three table OUTER join with a WHERE clause:

```
libname syb sybase user=john pw=doe;
 proc sql;
  select emplinfo.department, emplinfo.lastname
   from syb.employees left join syb.emplinfo
          on employees.empid=emplinfo.employee
                      left join syb.dept
          on dept.deptno=emplinfo.department
             where employees.empid > 100;
```

Based on the above query, Implicit Passthru will generate the following to pass down to Sybase:

```
select emplinfo.department, emplinfo.lastname
     from employees, emplinfo
  where employees.EMPID *= emplinfo.employee
       and (employees.EMPID > 100)
```

Note that Implicit Passthru generates Sybase-specific OUTER join syntax which uses the ' * ' operator to tag preserved tables in OUTER join queries.

In this example the first two tables in the OUTER join along with the WHERE clause gets passed to Sybase since there is a restriction of specifying more than two tables in an OUTER join with a WHERE clause. After the result set of this query is returned from Sybase, proc SQL processes the remaining join of the 'dept' table from the original SAS query. So you can still get perfomance benefits even when pass-down restrictions exist for the non-standard datasources.

PASSING DOWN SAS FUNCTIONS

It was mentioned earlier that, starting in release 8.2 of SAS, SAS/ACCESS libname engines support passing down a subset of SAS functions for which the underlying datasource has equivalents. Although the presence of SAS functions in proc SQL queries do not trigger a pass-down by themselves, they will get converted and passed down as part of queries that meet the aforementioned Implicit Passthru criteria.

Below is an example of a query using SAS/ACCESS to Oracle that contains the SAS function 'UPCASE' :

```
proc sql;
  select distinct UPCASE(joinchar.name) from
     ora.joinchar;
```

This query meets the Implicit Passthru criterion of DISTINCT processing and is therefore a candidate for getting passed down. It also contains the SAS function 'UPCASE'. Since SAS/ACCESS to Oracle supports Oracle's equivalent function 'UPPER', the SAS function 'UPCASE' gets converted to its Oracle equivalent and gets passed along with the rest of the query. Below is the generated query that gets passed to Oracle:

```
select distinct UPPER(joinchar."NAME")
     from JOINCHAR
```

If a SAS function that is not supported by the DBMS engine appears in the query, then the query (or possibly just the part of the query that contains the SAS function) will not get passed.

It should be noted that the subset of SAS functions supported in SAS/ACCESS libname engines varies for each engine. Please see the SAS/ACCESS documentation for information about which SAS functions are supported for a specific engine.

It should also be noted that the current list of supported SAS functions will potentially be expanded in subsequent releases of the SAS system.

WHAT DISQUALIFIES A QUERY FROM GETTING PASSED?

Any query that contains more than one SAS/ACCESS libref will be disqualified since different librefs may refer to different DBMS connections. For example, two different connections from the same SAS/ACCESS libname engine could possibly point to two different servers, hence precluding the passing of JOINS.

In addition, any query that contains one or more of the following will be disqualified [Church 1999] :

- data set options
- the INTO clause
- the COALESCE function
- remerging
- SAS functions (prior to 8.2)

The above constructs are disqualified because they are either SAS-specific or non-standard in an underlying DBMS.

For example, the COALESCE function is not supported in all DBMSs.

Data set options are generally too SAS-specific to be usefully converted, so they also preclude a pass-down. This includes the DBCONDITION option which does not get processed for Implicit Passthru.

As discussed above, the SAS function restriction has been lifted for 8.2 since SAS functions often have DBMS equivalents.

WHERE CLAUSE OPTIMIZATIONS

The WHERE clause optimizer is the facility that passes down SAS WHERE clauses. This facility has been in SAS/ACCESS products since V6. SAS WHERE clauses can be surfaced in any SAS procedure that operates on rectangular data, i.e.,

```
libname ing ingres database=musicalia;
 proc print data=ing.orders;
       where dateordered > '01jan1996'd;run;
```

SAS/ACCESS engines internally generate programmatic DBMS SQL when accessing DBMS tables specified in SAS procs. The SAS/ACCESS WHERE processor will parse through the SAS WHERE clause, convert it to a DBMS-specific WHERE clause, and append the converted WHERE clause to the programmatic SQL statement. In the above example, the SAS DATE9. value gets converted into an Ingres-specific date value before getting passed to Ingres.

It should be noted that the SAS/ACCESS WHERE optimizer is a different facility than Implicit Passthru, that is, any WHERE clause that is part of an Implicit Passthru query gets processed by Implicit Passthru. If a proc SQL query is not a candidate for Implicit Passthru but nevertheless contains a WHERE clause, then that WHERE clause will get passed to the SAS/ACCESS WHERE optimizer. This is because in the NON-Implicit Passthru context proc SQL is just another SAS procedure.

SAS FUNCTIONS GET PASSED IN THE WHERE OPTIMIZER

Beginning in 8.2, SAS functions will also get passed down in the SAS WHERE clause (providing that the SAS/ACCESS engine supports a DBMS-equivalent just like in Implicit Passthru).

So now you can specify:

```
proc print data=ora.joinchar;
   where LOWCASE(name) = 'Alison';run;
```

and SAS/ACCESS to Oracle will generate the WHERE clause with the Oracle equivalent function 'LOWER'.

If a SAS function specified in the SAS WHERE clause is not supported by the SAS/ACCESS engine, then the SAS/ACCESS WHERE optimizer will return an error and SAS will evaluate the WHERE clause.

DBCONDITION CAN BE USED WITH THE WHERE OPTIMIZER

It should also be noted that the DBCONDITION dataset option (which lets you pass DBMS-specific SQL conditions to the DBMS) works in concert with the WHERE processor. That is, any DBCONDITION WHERE clause will be ANDed to the SAS WHERE clause. All other DBCONDITION subsetting will be appended to the SAS WHERE clause. Below is an example of using DBCONDITION with a SAS WHERE clause using SAS/ACCESS to Oracle:

```
proc print data=ora.join1(dbcondition="order
by x1"); where x1 > 0; run;
```

The following is what gets passed to Oracle:

```
SELECT  "X1" FROM JOIN1  WHERE  ("X1" > 0 )
ORDER BY  x1
```

HOW DO I KNOW MY QUERY IS GETTING PASSED DOWN?

It is easy for you to determine if your query has been passed to the underlying DBMS using the SASTRACE option. SASTRACE is a SAS system option that has SAS/ACCESS specific behavior. SASTRACE shows you the commands sent to your DBMS by the SAS/ACCESS engine.

The SASTRACE syntax used for SAS/ACCESS engines is:

```
SASTRACE = ',,,d';
```

The `',,,d'` gives information about SAS/ACCESS engine calls to a relational DBMS. The following example shows how SASTRACE can be used to determine that an Implicit Passthru query got passed to the DBMS:

```
5? options sastrace=',,,d';
  6? proc sql;
  7? select distinct * from ora.join1;

DEBUG: Open Cursor - CDA=2059746056 0
979854457 orusti 299 SQL
DEBUG: PREPARE SQL statement:  1 979854458
orprep 63 SQL
 SELECT * FROM JOIN1 2 979854458 orprep 64
SQL
Prepare stmt:  select distinct join1."X1"
from JOIN1 3 979854463 prepare 671 SQL
DEBUG: Open Cursor - CDA=2062403848 4
979854463 orusti 299 SQL
DEBUG: PREPARE SQL statement:  5 979854463
orprep 63 SQL
  select distinct join1."X1" from JOIN1 6
979854463 orprep 64 SQL
SQL Implicit Passthru stmt prepared is:  7
979854463 ip_util 378 SQL
  select distinct join1."X1" from JOIN1 8
979854463 ip_util 379 SQL
DEBUG: Close Cursor - CDA=2059746056 9
979854463 orustt 370 SQL
```

In the SASTRACE example above you can see all statements passed to the DBMS. Since individual tables in Implicit Passthru are separately prepared before the query of which they are a part get passed , we can see the programmatic SQL statement 'SELECT * FROM JOIN1' in the SASTRACE output. Note that SASTRACE also tells you that a prepared statement comes from Implicit Passthru with the statement "SQL Implicit Passthru stmt prepared is:". So we can see from this example that the proc SQL query was passed to Oracle for processing.

For the WHERE processor, we would see the converted WHERE clause as part of the prepared statement if it got passed to the DBMS. Consider the following example:

```
16?  proc print data=ora.joindate; where
dateval > '01jan1960'd; run;

DEBUG: Open Cursor - CDA=2058853128 21
979855721 orusti 299 PRINT
DEBUG: PREPARE SQL statement:  22 979855721
orprep 63 PRINT
 SELECT * FROM JOINDATE 23 979855721 orprep
64 PRINT
DEBUG: PREPARE SQL statement:  24 979855721
orprep 63 PRINT


SELECT  "DATEVAL" FROM JOINDATE  WHERE
("DATEVAL"
>TO_DATE('01JAN1960','DDMONYYYY','NLS_DATE_LA
NGUAGE=American') ) 25 979855721 orprep 64
PRINT
```

In this example, we first see the initial prepare of the table followed by the prepare of the same table that now includes an Oracle-specific WHERE clause that has converted the SAS date to an Oracle-specific date. We now know from the SASTRACE output that this WHERE clause was passed down to Oracle.

ENABLING AND DISABLING QUERY OPTIMIZATIONS

SAS/ACCESS gives you the ability to enable and disable a range of SQL-based query optimizations on a SAS/ACCESS libname statement. The libname option to do this is called DIRECT_SQL and specifies what types of generated SQL you wish to pass down to the datasource. By default, all SQL query optimizations are enabled.

Using this option, you can enable/disable the following types of generated SQL:

- Implicit Passthru
- SAS functions (in SAS WHERE clauses and Implicit Passthru)
- OUTER joins involving more than two tables
- WHERE clauses (both SAS WHERE clauses AND Implicit Passthru WHERE clauses)
- ALL of the above or any combination of the above

For example:

```
libname mydb2 db2 database=sample user=john
          using=doe DIRECT_SQL=(NONE);
```

will disable all generated SQL, including Implicit Passthru, SAS functions, multi-table outer joins, and WHERE processing.

Another example:

```
libname mydb2 db2 database=sample user=john
        using=doe DIRECT_SQL=(NOFUNCTIONS
                              NOMULTOUTJOINS);
```

will disable outer joins involving more than two tables and passing

down SAS functions in any context. This means that Implicit Passthru queries that do not contain SAS functions or more than two table outer joins will still get passed, as will WHERE processor queries that do not contain SAS functions.

WHY DISABLE THESE OPTIMIZATIONS?

There are several reasons why at times you may want to disable some or all of these optimizations:

1. NULL data is often processed differently in DBMSs than in SAS. If your DBMS data contains NULLS, there may be times when you need to disable Implicit Passthru and the SAS/ACCESS WHERE clause optimizer. This is because you can potentially get different results depending on whether SAS or the DBMS is doing the processing.

2. There are times when specifying complex OUTER joins with more than two tables can produce different results in SAS and an underlying DBMS. You may wish to disable just this feature without disabling other types of generated SQL. This is not a common occurrence, but it is possible.

   For further information concerning 1. and 2., see the SAS/ACCESS white paper "***Potential Result Set Differences between Relational DBMSs and the SAS System"*** on the SAS Data Warehousing Web page: http://sasprod.unx.sas.com/service/news/feature/15jan01/accessv8.html

3. Some DBMS equivalents of SAS functions might produce slightly different results than SAS in very specific cases. Again, this is not common, but is possible.

4. In some cases, SAS can process the mathematical functions more efficiently than the DBMS equivalents.

For further information on the DIRECT_SQL libname option, please see the SAS/ACCESS documentation.

CONCLUSION

You have now seen some of the new performance features in the SAS/ACCESS libname engines that, when used judiciously, can improve overall engine scalability in the areas of loading/extraction, ASYNC I/O, and SQL-based query optimizations. Some of these features are already providing benefits for the SAS/ACCESS user community, and we will continue to focus on and improve scalability in our engines into the future.

We welcome your feedback in all areas of SAS/ACCESS development.

REFERENCES

Church, L. (1999). "Performance Enhancements to PROC SQL in Version 7 of the SAS System". *Proceedings of the twenty-fourth Annual SAS Users Group International Conference, 24.*

ACKNOWLEDGMENTS

Thanks to Doug Sedlak and Brian Hess for their technical contributions to the ASYNC I/O section of this paper.

Also thanks to Lewis Church who was my partner in implementing SQL Implicit Passthru. Much of the sophistication of Implicit Passthru can be credited to Lewis's SQL expertise.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Fred Levine
SAS Institute Inc.
100 SAS Campus Drive
Cary, NC 27513
Work Phone: (919) 531-6826
Fax: (919) 677-4444
Email: fred.levine@sas.com

EMERGING TECHNOLOGIES

SECTION CHAIRS

Jack Shoemaker
STATPROBE Technologies

Clara Waterman
Maxim Group

EMERGING TECHNOLOGIES

All I Really Want... A Wish List for New SAS® Software Enhancements

Peter Parker, US Dept. of Commerce

Abstract

Having used both SAS software and computers for over 21 years, I've seen many changes in both, almost always good. Major improvements will continue to be made and we'll look back five years from now, amused at what we called state-of-the-art today. Remember when a 486 PC with a 20 MHz clock speed, 4 MB of RAM and a 300 MB hard drive was considered the dream machine? Remember when packing a million transistors on a chip was a major breakthrough? The latest computer chips have over 42 million transistors.

In order to advance technology, we must first determine what needs to be improved. While SAS software has proven itself to be adaptable, dependable, and robust, it, too, will become better. Here is my wish list of changes that I hope will become part of the SAS software system soon:

System Enhancements

1. Run- time SAS applications
2. Multi-threaded processing

Application Development Enhancements

1. Import Wizard for most types of data
2. GUI code generator
3. Intelligent Printing
4. Merge Wizard
5. Coding Critique Wizard
6. SAS code templates
7. SAS macro window overhaul

Introduction

I've been using SAS Software for a long time (20+ years) and I've used it on ancient mainframes, keypunching out PROC statements and submitting stacks of fragile punch cards to computer operators. Now I type my code on my recently state-of-the-art PC and submit and review my computing jobs in the privacy of my office. Much has changed since then, quicker and easier submission of number-crunching queries, color-coded code, improved interfaces, and tweaking of the software syntax. However, much has not changed. You still have to write code, use DATA steps and PROC statements, and use logical, correct syntax. There is still Base SAS software that can do anything that lower level languages like FORTRAN and COBOL can do, but with much less typing and debugging. However, if one wants to read directly non-SAS data files like dBase, he either has to write a dynamic data exchange program (DDE, not a coffee break program) or buy another SAS software product like SAS/ACCESS®. A lot has changed, yet a lot has not.

This paper will focus on the backbone product of SAS software, BASE SAS, and how that product should evolve beyond tweaking and more Windows-like graphical user interfaces (GUI). It appears that most of SAS software's growth has been in creating more software products. This progression is desirable for those users who have advanced econometric or graphics needs, but BASE SAS is the only product that most people need for number crunching. They want to perform advanced data manipulations without having to buy the other SAS products like SAS/ACCESS,

which quickly add up the costs of running SAS software.

I've examined the changes that need to be made to the overall architecture of SAS software (system enhancements) and to the programmer interface (application development enhancements). Every year, the SASware Ballot® asks users what changes that they want. This is my formal response. This is my SAS software wish list.

System Enhancements

Run- time SAS applications

Ever since running SAS software on PCs became viable for serious data processing, I've been requesting a run-time version. So have many other SAS programmers. A run-time version of a program is a compiled form of that software that will enable any PC, regardless of whether that software is installed, to use that particular customized application. For example, one can create a Microsoft PowerPoint presentation, save it as a run-time version, and distribute it to other users, regardless if they have the PowerPoint software on their PCs. As long as they have the Windows operating system, they can run that particular PowerPoint presentation. However, they would not have any other PowerPoint capability, such as creating new presentations or editing the current one. Having this similar functionality with SAS software could have many useful applications.

Suppose with SAS software, I create an program that reads a particular SAS data set (e.g., Textile and Apparel Imports for 2001). I can include SAS Macro Windows, so that when the program is run, the user will be able to generate a report, based on parameters that he passes through the screen prompts, such as what particular textiles commodities he wants to see and from what countries they were exported. If one could compile this program in a run-time version of SAS software, then he could copy this runtime program and SAS data sets to a CD-Rom and distribute it to clients. Then the clients could run pre-made queries on this textile and apparel data, limited only by the scope of that run-time program. They would not be able to run other SAS applications or create other SAS programs. They'd only be able to run that one SAS program that the licensed SAS user has created.

Obviously, the lack of run-time SAS software is legal/business concern, rather than a technological issue with the SAS Institute. Having a run-time version of SAS software may invite license abuse. Rather than having several SAS software licenses within a company, it would need only a few copies for the programmers. The users, themselves, could run runtime queries, instead of relying upon the IT staff. Since there is no run-time software, some companies may buy the next closest thing, SAS/IntrNet™, not only for running SAS software on the Internet, but also for running SAS software on the Intranet. They would have to weigh the cost of SAS/IntrNet software versus individual licenses. Another option around the lack of runtime software is to use an OBDC interface to the SAS system. For example, our office has created applications in Borland Delphi software that interfaces with SAS software to read the SAS data and produce reports.

Multi-threaded processing

Since many servers today have multiple CPUs, it would advantageous to run software that optimally uses all of these processors. In theory, the application will run faster, subject to bottlenecks such as how fast the hard drives can deliver the data to the CPUs to be processed. In order to use all of these CPUs, the software must be multi-threaded.

The SAS software is single-threaded. A thread is a list of instructions to perform a certain task. Multiple-threaded software can perform several tasks simultaneously if the resources are available. Only by running multiple applications such as multiple SAS jobs at the same time could SAS software run faster on a multiple processor system. Each SAS job would not run faster, but they

wouldn't have to share one processor among them.

The Windows NT operating system uses symmetrical multiprocessing where the system can load multi-threaded applications among the different processors, trying to balance the computational load rather than having certain processors always do specific type of processes. Running multiple single-threaded applications could also use these multiple processors.

With SAS version 8 software, there is a new product, SAS/MP CONNECT®, bundled with SAS/CONNECT®. Besides having to buy yet another SAS product, this new product is not a solution for "easy to use" and "true" multiprocessing. Additional SAS/CONNECT coding is required, emulating the concept of logging on to different hosts (i.e., starting a remote SAS session on each processor), where all the hosts are on the same PC. As well, the multi-processing only works when the tasks are independent. One can merge two SAS data sets, but cannot sort the newly combined data set until that merge is complete. The more desirable form of processing, not yet provided by the SAS Institute, is where the two processes are dependent but the former one does not need to be done completely before the latter one can begin. In this example, the SORT process wouldn't need to wait on the merging of two data sets before it can start sorting the combined data. It could start the process as soon as a few records have been merged.

Ideally, the best case of multi-processing involves these three elements:

1. Built into Base SAS software (so one doesn't have to buy yet another SAS product).
2. Works transparently with all existing SAS code (so one doesn't have to modify their current code).
3. Works for all tasks, regardless if independent or dependent.

Application Development Enhancements

Import Wizard for most types of data-

Some people may consider the wide array of software available as being an over abundance of technological riches. We have so many choices and so many ways of doing things. We may have too many choices, leading to confusion. Data can be stored in several ways, spreadsheets (Lotus, Quattro, Excel), rich text, ASCII text, binary, dBase, Paradox, SAS software, SPSS, etc. Often we need to read this information into a SAS program for data manipulation. One can use OBDC interfaces to read directly a dBase file into SAS software or to write a SAS data set into either dBase database files or Quattro spreadsheets. The coding is not difficult. However, for each type of data, we have to write and debug SAS code to handle it. Rather than having to write this code, the SAS software should handle it for you, or at least, attempt it. One should be able to make a LIBNAME reference to a spreadsheet and in the code, write-

```
LIBNAME ABCSPREAD
c:\mydata\spreadss.wk3;
.

.
.
DATA XYZdata;
INSERT abcspread;
.
.
```

Here a LIBNAME statement points to a spreadsheet file on the c: drive (c:\mydata\spreadss.wk3) and using an "Insert" command (this keyword doesn't exist, but could be a name for this code), SAS software would figure out how to insert this spreadsheet into the temporary SAS data set (XYZdata). One could have parameters in the "Insert" statement that could specify which ranges of columns and which folders in the spreadsheet to use, and any other relevant information to define how the non-SAS file is to be converted into a SAS file.

In Base SAS version 8 software, there is already a limited import wizard that can read in delimited, comma separated and tab delimited files, all being different types of ASCII text files. One can also read in Microsoft Excel, Microsoft Access, Lotus 123, and dBase files, but he needs to purchase yet another SAS product, SAS/ACCESS. As noted elsewhere, this import/export feature should be bundled with Base SAS software and not be another costly extra.

GUI code generator-

While easy to use, SAS software uses its original paradigm of manually writing code, be it DATA Steps or PROCS or system options. Seasoned SAS users easily can rattle off this code, rarely having to look up the syntax in the many SAS manuals. However, to reuse the popular tirade of the 90's, the SAS Institute needs to have a paradigm shift. An alternative Base SAS software would use objects instead of code to set up a program. However, the programmer would be able to modify the code behind the objects, when necessary. As an analogy, when CompuServe started to provide remote service, users had to memorize keywords and literally type in "Go Mail" or "Go Billing" to navigate around the product. Then AOL 1.0 was released and the user rarely had to type in commands, instead clicking on a "Mail" or a "Billing" icon. Other current software uses this object-oriented look. In Lotus Notes, the information database/email system software, one can create database fields as objects on the screen with a pick list containing all valid responses. All of the Lotus Notes components can be viewed as objects, whether they be database fields, agents, action buttons, forms, sub-forms, views, pages, navigators, outlines, framesets, folders or other Lotus Notes constructs.

Perhaps a SAS-Lite version could be created for neophyte SAS programmers or for experienced programmers to create quick jobs. There could be SAS objects for SAS data sets, non-SAS data sets, and SAS work files, and they all can be dropped into action boxes that filter, sort, and merge these objects. The results of these actions boxes can be connected to generator boxes that print reports, create frequency tables, run regressions and other SAS PROC features. A programmer would have access to an alternative view where he can see the code generated and then change it, if desired. Just like Netscape Composer may create a web page, one could open the generated html file with any ASCII editor and make changes to it. As improvements are made to the SAS-Lite interface, the programmer will have less reason to edit the actual code.

Since pictures are more intuitive than words, one can look at a page of objects and understand what a program does, rather than having to read pages of code. This concept is similar to creating a program flowchart that is also the program.

Intelligent Printing

Printing is a problem for all software, including SAS software, spreadsheets, databases and word processing. Printing should be as effortless as pressing the print button and what's on the screen will be transferred perfectly to paper. It doesn't always work that way. Often, especially in SAS software, you have to adjust the fonts and page sizes and play with the code to have the report fit correctly on the pages. With some fine-tuning, page breaks will be correct, instead of several lines later on the next page. It becomes an art to have the printer output suitable for distribution.

One solution suggested to me years ago by a software representative (not from the SAS Institute) is to go completely paperless. The solution to printing is not to print. However, in many ways, paper is better than a computer screen. It has better resolution. It can be marked up by hand and it is more portable; you can fold it up and put in your back pocket. I'm always annotating it, underlining words and highlighting sections. Shifting paradigms here is not a recommended alternative.

The solution is to have the software, operating system and printer all communicate better, with minimal user intervention necessary. The only intervention suitable would be to tell the printer to duplex (print on both sides of the

paper) or to decide which printer to use. All other housekeeping duties should be done by the computer, regardless of the fonts used or of what type of report or table is being used. One should be able to use SAS options to determine page-size or line-size, but I doubt many programmers would bother if the software took care of it well. I only change these options from the default when my reports don't print correctly.

Merge Wizard-

Perhaps the most confusing part of Base SAS software is using the merge feature. While it has the powerful function of combining data sets together, it may cause problems not discernable immediately to the programmer. For instance, does he want to merge two data sets only if they have common values for a certain variable? Does he want multiple records when there's a match or just one? What if the data sets have common variables besides the matching one? What value will the variable then have, after the merge? It all depends on how you define "merge". A graphical user interface (GUI)-based Merge wizard might ask the right questions to the programmer to determine what he really wants. The merge wizard could illustrate the different types of merges, whether it's one-to-one merging or match merging and then show the results, using simple examples of combining two small data sets.

This merge wizard could evolve into a more advanced assistance to programmers. If you just want to append one database to another, the wizard may suggest that you simply use a "SET" statement or some other technique that is better suited. Eventually, one won't even need the many SAS software manuals. One can have assistance via the SAS wizard. The merge wizard could become an artificial intelligence agent for all SAS software features.

Coding Critique Wizard-

It's not hard to write adequate code. You sketch out the basic workflow logic, type the code as you think it up, create some test data, debug the program on the test data, and then run it against real data. If it runs within a reasonable amount of time, the program is a success. The job is done and it's time to move on to other projects. Never mind that the program may have some wasteful sorts or inefficiently written DATA steps. It may not matter at the time that the code is so poorly written that a year from now someone may have to spend long hours attempting to decipher it.

Are you writing good SAS code? Is this SAS code that you would want someone else to read? Would you publish it in a SUGI paper? We shouldn't ask only whether it works and what can we get away with. While shame does have its place in programming, among other aspects in life, your current code also may not work in the future. I found that some of my SAS 6.12 code did not work in SAS 8.0. I had to tighten up the code. That means I had to be more careful with coding syntax. The message is basic. Do it right the first time, especially if you plan to reuse it.

Artificial Intelligence software that reads your program and critiques it may be a unique approach to fixing clunky programs while they're still fresh. A similar idea is used in checking out websites. You submit your website address to one of these services and it rates your page, based on load time, browser compatibility, dead links, spelling and HTML design. A SAS code critique wizard could give warnings on merges, inform you of inefficient coding, and give examples of better coding. For instance, if you don't have a KEEP statement and you don't use all of the data fields, it should remind you to use a KEEP statement in a DATA step. If you have included a KEEP statement but not use all of the fields listed in it, the wizard should inform you to trim down that KEEP statement.

SAS code templates-

I'm always taking old programs and recycling them. Why write a program from

scratch, spending a lot of time typing and debugging, when I already have a similar one? The SAS Institute should be able to implement something on that order, creating program templates that one can use to quickly build a new program.

For example, Lotus Notes, the information database and workflow software, provides several database templates as a starting point. They have templates for creating a document library, a registration interface, and a discussion database, among others. The SAS Institute should be able to implement similar templates. For instance, you may have a template that reads in SAS data sets, filters the records, and run reports based on parameters passed through macro windows. Another SAS template may read in data, run validity checks and then run certain statistical procedures. Another SAS template may be set up to read in non-SAS data sets, filter out the data, compute new data fields and then create html output for a web site. The SAS Institute could determine, with market research, how most people use SAS software, and create these templates based on the most common needs.

SAS macro window overhaul

SAS macro windows provide an interface for users to pass parameters through simple windows forms. If one wants to query the textile and apparel import data for knit shirt imports from China for the year 2000, one may create a SAS report, by querying the data using a simple macro window that may look like this:

Textile and Apparel Query-

Textile and Apparel commodities-
[ ]

Country-
[ ]

Year- (select with "x")
1998 _ 1999 _ 2000 _ 2001 _

However, this interface is only a "simple" window, not much different than the clunky front ends to DOS software that we stopped using years ago. It needs more component objects, such as check boxes, radio boxes and dialog lists, to make it easier to use. In the above example, one needs to type in the first two parameters. For the type of commodity, is it spelled "knit shirts" or "knit-shirts"? Is the country "China" or "People's Republic of China"? What if it's for a country that hard to spell like "Madagascar" or has changed its name several times, like "Burma/Myanmar"? What if you have 20 years of data? How do you get it all on the form? What if you select more than one year? Can you have multiple commodities and multiple countries? Obviously this macro window has its limitations. Here is an improved window:

Textile and

- Kni
- Wool Trousers
- Oxford Cloth shirts
- .

>--Select a Country--

Country-

Year-

- 1998
- 1999
- 2000
- 2001

In order to be a "complete" interface, it needs most of the work done by the computer, minimizing errors such as the user misspelling the country name or typing in "swimsuits" when "swimwear" is the appropriate term used in the database for beach clothing. Instead of a "DOS" approach, SAS software needs to emulate the interface that Lotus Notes and Borland

Delphi enables application developers to create.

Conclusions

There's always room for improvement.

But before you can make improvements, you need to determine what needs to be changed. You need a wish list. Most of the changes listed here are for repairing existing SAS software products (macro windows, intelligent printing). Some are major systems changes, such as making SAS software a true multi-threading application. Some are major transitions, such as replacing written code with objects/symbols. One should expect many of these changes within the next five years, especially the small fix-ups. However, I don't anticipate that the major changes may ever occur. SAS software has been around for over 25 years, and it has a responsibility to support legacy systems. SAS software versions 9, 10, and 11 will still need to support SAS version 5 and 6 applications. While that may be the responsible action to take, it also limits how much the software can evolve. Backward compatibility has that inherent short-coming, just as Microsoft still has some ties to running DOS applications, although its Windows and Windows 2000 operating system are far beyond the simple command line operating systems.

Will SAS software evolve like I've outlined in this paper? Not likely, but after all, this is only a wish list.

References

Bentley, John E. (2001) "SAS® Multi-Process Connect: What, When, Where, How, and Why" in SUGI26 Conference Proceedings. Cary, NC: SAS Institute.

Garner, Cheryl (2000) "Multiprocessing with Version 8 of the SAS® System" in SUGI25 Conference Proceedings. Cary, NC: SAS Institute

Parker, Peter (2000) "Optimizing SAS® Software on Windows NT: A Guide to Performance Tuning Your Applications Server" in SUGI25 Conference Proceedings. Cary, NC: SAS Institute

Parker, Peter (2000) "SAS® Software Macros: You're Only Limited By Your Imagination" in SESUG Conference Proceedings and NESUG Conference Proceedings. Cary, NC: SAS Institute

Contact Information

Your comments and questions are valued and encouraged. Contact the author at:

Peter Parker
US Dept. of Commerce
Room 3100
1401 Constitution Ave., NW
Washington D.C. 20230
202-482-1449
peter_parker@ita.doc.gov
http://otexa.ita.doc.gov

Version 9: Scaling the Future

Diane Olson, SAS® Institute Inc., Cary NC
Robert Ray, SAS® Institute Inc., Cary NC
Presented by Scott Mebust

ABSTRACT

In Version 6, SAS introduced the SPDS (Scalable Performance Data Server) as a technology to exploit SMP (Symmetric Multi-Processor) hardware for speeding up data services. In Version 9, BASE SAS introduces the SAS Scalable Architecture (SSA). SSA makes parallel processing and partitioned I/O constructs available to the entire SAS System for the first time.

What does that mean to you? It means that some elements of your jobs will take full advantage of SMP architectures to reduce time-to-solution for critical tasks. This paper presents an overview of this new strategy to improve your compute and I/O bound tasks, including specifics on how you may get a significant performance increase for your SAS jobs.

INTRODUCTION

The data processing and warehousing industry is bombarded with immense volumes of data from both the "bricks and clicks" sides of the business; however, the data processing time remains unchanged. In response to this data onslaught, SAS has a strategy that will be incorporated in an evolutionary manner over the next several releases, starting with Version 9.

Changes in Version 9 include performance enhancements for both I/O and compute-bound problems. These performance enhancements are realized by a combination of strategies:

- partitioned I/O
- multi-threaded I/O
- parallel multi-threaded computation

This paper begins with an overview of SSA and enumerates the problems Version 9 starts to address. Features to remedy these problems are explained, along with how you can make use of the new features. Preliminary performance benefits resulting from these changes are included.

SAS SCALABLE ARCHITECTURE

Successfully scaled performance is not obtainable by simply installing faster processors or I/O devices. Achieving true scalability is a balancing act involving the choice of scalable hardware and some choices about the software that is specifically designed to leverage it. For the Version 9 SAS user, these choices include selecting optional enhanced procedural algorithms, choosing the best SAS engine to create and access data, how (or indeed, if) data sets are partitioned, as well as the number of threads launched to best process those partitions.

SSA combines the legacy strengths of SAS with the ability to create a highly scalable solution in order to meet the evolving market demands, taking advantage of hardware advances. This involves embracing elements of the classic MVA (Multi-Vendor Architecture) server, SPDS data server, and technologies for lightweight threading. Integration of SPDS-like technology means employing threads to distribute I/O requests across multiple controllers. These threads may in turn create other threads to manage the processing of blocks of data. Thus, SAS flows from a mostly synchronous model to one that allows I/O functions to run asynchronously across multiple processors in an SMP environment. Properly used, an SMP environment can employ the CPUs in the system to process data simultaneously, avoiding bottlenecks that slow the computation of results.

When trying to speed a single task or problem, there are two types of scalability to consider - the inherent scalability of the problem in question and the scalability of the software solution for that problem. Problem scalability can vary greatly. The problem of sorting, for example, generally scales computationally on the order of $N log_2 N$, where N is the number of records to be sorted. However, if the I/O device cannot keep pace with the CPU, then the scalability will be linear with the size of the file (N). A full SQL join will scale computationally as $N*M$, where N and M are the table row counts. Therefore, doubling the number of observations for both tables in a full join could consume four times the CPU resources. However, if the process were I/O bound, the actual scalability would be linear with the combined size of the tables ($N+M$). Therefore, reducing time-to-solution is a complex problem involving both CPU and I/O optimizations.

With software scalability, the goal is to apply additional physical resources (CPUs or I/O channels)

and have the real time-to-solution be lowered by a proportional amount. The real time is the focus here, and not the combined CPU time. It is a foregone conclusion that extra CPU cycles are consumed to manage a set of process threads across multiple CPUs. The portion of the original problem that can actually be processed in parallel governs the amount of scalability achieved from the software solution. For instance, although a partitioned data set can be read in parallel for SORT, the sorted data is written out in a linear fashion to preserve the sorted state. If writing the data takes 50% of the original time, then only 50% of the process is scalable; this represents the limit of scalability for this problem. Further improvements in time-to-solution can only be achieved by increasing the speed of the output I/O devices.

EVOLUTION

Rewriting the entire legacy MVA code to be thread-safe (i.e. where the code runs with no side effects in a threaded environment) much less thread-hot (i.e. where the code is thread-safe and exploits the threaded environment for faster execution) would be a major undertaking. To avoid delays in delivering releases to the customer, the move to thread-hot code has been planned in evolutionary stages.

In Version 9, some shared code modules used by selected high profile procedures are being reworked to be thread-hot. In the I/O arena, a new subsystem was created to enable reading blocks of data versus record-by-record reading as in previous SAS versions. Several multi-partitioned engines will allow simultaneous block-mode reading of data from multiple partitions in parallel. A small number of procedures are being converted to exploit this new subsystem in Version 9. These include SORT, SUMMARY, DMREG and REG, with more procedures expected to come aboard in following releases.

BASE SAS procedures have a set of incremental goals for leveraging SMP architectures from the traditional MVA development environment. First, some key procedures' algorithms are being modified to decrease time-to-solution using the traditional single-partition data set. In these cases, both parallel and pipeline algorithms are being applied, as appropriate for the application. The opportunity to decrease the time-to-solution for single data partitions exists when the problem is somewhat CPU bound; in these cases, the procedure can use multiple threads, and thereby CPUs, to match the data processing rate with the I/O read rate. Once the read rate of a single partition is matched, additional increases in speed must be achieved by using either a faster single I/O channel or partitioning the data so that multiple partitions can be read simultaneously. Many BASE SAS procedures exhibit this "fence straddling" between being I/O and CPU bound, depending on the nature of the data being processed and the procedure options being used.

Not all procedures will be able to exploit the new parallel I/O system, which is capable of reading blocks of data across partitions simultaneously. PROC PRINT, for example, expects to print the observations of a data set from the first observation through the last; reading the data set simultaneously across threads does nothing to speed ordered access. PROC SUMMARY, however, does not necessarily depend on ordered access and so could leverage parallel data access.

In Version 9, the new I/O subsystem will have limitations that may prohibit parallel data access under certain circumstances. These include BY processing and some engine-dependent capabilities. In addition, there will be no parallel data writing capabilities. These exceptions will be addressed in forthcoming releases. See Table 1 for a per-engine description of factors that prohibit the use of the new I/O subsystem. Note that this table is expected to change in subsequent releases. This evolutionary plan allows quick delivery of performance enhancements to the customer as well as laying groundwork for future advances.

FEATURE	BASE ENGINE	ACCESS ENGINE	SPDS ENGINE
Compression	No	Yes	Yes
Encryption	No	Yes	Yes
BY processing	No	Yes	Yes
CEDA	No	No	No

Table 1 - Feature support per engine with new I/O subsystem

BENEFITS

The large data sets of today have changed significantly from that of a decade ago. The size of

today's large data sets would have been unthinkable in the Version 6 timeframe; even so, results from processing those gigabytes of data are still expected to be obtainable in a reasonable amount of time. Advances in hardware technology help improve throughput, but without software advancements, processing time can still be unacceptably slow. The migration to a threaded SAS architecture will lead to faster processing of not only today's data sets, but also those of the future.

Faster processing is possible because SSA provides the tools necessary to exploit multiple CPUs concurrently, to read data through multiple I/O channels simultaneously, and to overlap I/O and data processing. SAS applications will have the opportunity to utilize these tools to increase the total throughput of a single SAS server session. Of course, the degree of time-to-solution improvement gleaned is directly related to how CPU bound or I/O bound the problem is. SSA works in tandem with the MP-CONNECT technology; MP-CONNECT enables parallelism via multiple concurrent SAS invocations.

PROBLEMS

There are several potential bottlenecks standing in the way of a full solution. Which bottleneck that is restricting your performance depends on both the operation requested and the nature of the data itself.

Depending on the type of user request, some applications will be I/O bound, that is CPU time to actually process the data is negligible in comparison to the time required to read the data. Data sets with many variables and many observations are more likely to create this situation, but it is still dependent to a great extent on the operation requested. For example, requesting the means of a numeric variable is not
CPU intensive, and when run on a large data set the process will be I/O bound.

Conversely, PROC REG of such a data set is likely to be compute-bound since the CPU time to process the observations is greater than the time to read the observations.

The goal of adding threaded access is to alleviate both types of bottlenecks, solving the problems of

- Speeding I/O
- Speeding Computation

Some procedures, like PROC SORT, are not solely I/O bound or solely compute bound, but contain processes that are. Some of these processes may be I/O bound, while some may be compute bound. These procedures benefit from a combination of the solutions for speeding I/O and computation.

Of course, you are **always** going to have some bottleneck, but the software developer's ultimate goal is to have it be the hardware that constrains the time-to-solution.

SOLUTIONS

That ultimate goal is certainly a desirable one, but it seems esoteric without a structured implementation strategy. For Version 9, that strategy is enumerated by attacking problems on three fronts, including those applications that are I/O bound, those that are CPU bound, and those that are a mixture of each.

SPEEDING I/O

For I/O bound processes, the goal is to read as quickly as possible, keeping the computation process supplied with data. The solution for increasing data set access speed is two-fold; first, optimize the read rate for each data partition, and second, allow multiple partitions to be read simultaneously using multiple I/O controllers.

Traditionally, SAS has always read data sets record-by-record, a style that is extremely flexible and lends itself to noting, pointing and indexing. However, speed-reading is more important for I/O bound applications. Thus, the new I/O subsystem reads data sets a block at a time, resulting in lower function call overhead. Delivering a block of data is somewhat less flexible than record I/O, as it is not as easy to address a particular record in the data set; however it fulfills the need of the I/O bound problem – more data in less time.

The second part of the I/O solution is to simultaneously read partitioned data sets via multiple threads. Partitioned data sets are data sets that are located in different physical locations, but still comprise a logical data set. With this strategy, not only are the block reads done in parallel threads, but processing of the data can also be done in separate threads. Each thread reads and processes data independently; when all partitions have been read and processed, the application folds the result from each partition (or set of partitions) into a final solution. Note that more than one partition may be read and then processed in a single thread.

The degree of increased throughput still depends heavily on hardware considerations. If there is only one I/O controller, for example, it does not matter how many threads are launched; they would all serialize waiting on the controller. Assuming no hardware constraints, the limiting factor becomes the number of

partitions comprising the data set and the number of threads launched to process those partitions.

There has never been an engine that processed partitioned data sets shipped with BASE SAS. Beginning with Version 9, however, it is planned that the partitioning SASSPDS engine will be shipped with BASE SAS, available on all platforms. ACCESS users will also have the choice of Oracle, Sybase, DB2 or Teradata as their partitioned engine. This allows all SAS customers to employ partitioned data sets if they so desire.

SPEEDING COMPUTATION

When I/O is not the only limiting factor, parallel computation using modern thread constructs allows further speedup to occur. The two basic models of parallel computation are the boss-worker model and the pipeline model. Both models allow a computationally intensive task to be divided and distributed to multiple CPUs within a shared memory model using lightweight process threads (LWPs). Each SAS procedure that is reworked to improve scalability will use one or both of these techniques along with parallel I/O where appropriate in order to reach its scalability limit.

In order to utilize threads, the traditional SAS MVA procedure will create process threads that essentially run outside of the MVA environment, synchronizing back to the MVA process once some portion of work is complete. Since these new process threads do not execute in the current SAS MVA process space, they do not detract from and may even enhance the interactivity of the SAS environment during the execution of long-running procedures on SMP machinery.

SCALING PROC SUMMARY

One BASE SAS procedure that can leverage both the parallel and pipeline techniques is PROC SUMMARY. When a CLASS statement is used with PROC SUMMARY, the rate at which data can be classified, sorted and aggregated is often not as fast as the maximum data read rate. In order to keep pace with input and minimize amount of redundant aggregate groups in memory, we choose to pipeline this process by dividing the summarization process into distinct steps and assigning each step to a separate thread. Data is processed in blocks by each stage and passed on, much like an assembly line. Memory resources required by each step are not duplicated; there is only one thread per step. It is possible that some stages in the pipeline may be able to make use of multiple threads. In these cases, the memory domain of the stage is divided and assigned to separate threads, thus avoiding memory duplication within a stage.

Once parity with the maximum read rate of a single partition of data is achieved, the model must be replicated across multiple partitions to achieve further speed-up. If classification is performed from separate input partitions, the elements of the pipeline could be replicated for each partition. However, this could result in a great expansion of memory required to solve the problem, as each domain could contain partial aggregation results for identical output levels. If the aggregation space is shared across multiple partitions/thread, excessive shared memory access overhead is a possible result. Clearly there is a tradeoff involved for parallel multi-partitioned summarization.

When the CLASS statement is not used, only the aggregation step of the summarization process is left; this causes the problem to be I/O bound. In this case, scalability is achieved by allowing parallel aggregation across multiple partitions. The results of each partition aggregation are merged together and the input scan is complete.

RESULTS TO DATE – PROC SUMMARY

Thus far the work on PROC SUMMARY has focused on summarizing from a single partition data set using multiple CLASS variables. Using multiple CPUs, for some cases the time-to-solution is within fifty percent of the time it takes to simply read the data set and write the results. This is approaching the best that can be done with a single partition and represents a significant improvement over the current summary logic.

SCALING PROC SORT

Another BASE procedure that can benefit from multi-threaded design is PROC SORT. Sorting is generally I/O bound, but thread technology provides the opportunity to overlap I/O and processing in some steps. The boundness of internal sorting can be a function of the ratio of the sort key length to the record length, as this affects the number of CPU cycles used per key comparison. Therefore, to ensure that key processing keeps up with the maximum read rate, a process pipeline with fan-out is being used. When the size of the sort exceeds the available memory, the sort must shift to an external algorithm, spooling partially sorted results to one or more utility files. These partial results are finally merged together to produce the final result.

External sorting adds many complications, as well as many opportunities for performance improvements. By employing multiple threads in the V9 design, the phases of multi-way merging can be isolated and have dedicated threads for each phase. These phases include runs creation, utility file read-ahead, merging records and writing output. Effective utility file read scheduling is critical to reduce head seeks

during external merging. Anyone sitting near a hard drive during an external sort merge-back can attest to the "weed whacker" sound produced by heavy disk seeking. Seek elimination makes for more efficient (and quieter) external sorting. By assigning this read-ahead function to a separate thread, records flow smoothly into main memory for a merging thread to process.

RESULTS TO DATE – PROC SORT

Measurements thus far for Version 9 PROC SORT have focused on internal sorting. For sorts where the key size is less than fifty percent of the record length, it is relatively easy to sort as fast as the I/O rate since the job is primarily I/O bound. However, if the key increases beyond eighty percent of the record length, as many as four sorting threads can be effectively employed in order to keep pace with the input rate. The additional work comes from the fact that each key comparison takes longer than the time to read each record. Thus, as the I/O rates improve, the multi-threaded sort on SMP machinery will keep pace; this will provide the performance improvements the customer expects from their hardware investment.

USER'S RESPONSIBILITIES

To scale performance using Version 9 SAS, the customer's input is of utmost importance. First and foremost are the hardware decisions. Because some processes serialize, the maximum throughput of any given I/O device may become the limiting factor for the time-to-completion. Lots of slow disk drives are not necessarily equivalent to a couple of really fast ones. However, having only one really fast disk drive will not allow partitioning to decrease throughput time. SMP hardware appropriate for your I/O bound and/or CPU bound applications is necessary to SSA's effectiveness.

Another user responsibility is the SAS engine choice. If you wish to have multi-threaded access across partitions, you must choose an engine that supports partitioned data sets. It is planned that you will have the choice of using the SASSPDS engine or one of the partitioning ACCESS engines. The default BASE engine will not partition files.

Certain engines or attributes of a particular data set may limit your data to record I/O access. See Table 1 to determine if any of those limiting factors will affect your application. For example, the BASE engine is not planned to support block I/O reads when the data set is encrypted. If encryption was less important than performance, then the file could be rewritten without encryption and block I/O reads could be used.

SUMMARY

Partitioned I/O, parallel I/O, parallel processing and algorithm changes are all a part of the Version 9 strategy of speeding your time-to-solution. With these evolutionary changes, expect to see performance gains from selected applications using SMP hardware. The gains seen in Version 9 are merely the beginnings of the plan for scaled performance increases across all applications.

Your comments and/or questions are very welcome. Please contact the authors at:

Diane Olson - Diane.Olson@sas.com
Robert Ray - Robert.Ray@sas.com

Paper #P153

Avoiding eOverload:

Personalizing Web Content through Security, *e*Intelligence and Data Mining

Gregory S. Barnes Nelson STATPROBE Technologies Cary, NC

ABSTRACT

Today, the worlds of high-tech with high-touch are quickly converging. The landscape of this new frontier in business intelligence and customized content has challenged the traditional models of delivering knowledge to diverse constituents. To make the most effective use of these delivery methods, business needs detailed intelligence on how people use information from these sources. In this paper, we will explore the analytical techniques, technologies and tools used to answer real-world business questions, accelerate knowledge transfer, and foster more profitable relationships with customers, partners, employees and suppliers.

Specifically, we will explore the business and technology underpinnings of personalization. Then we will examine the analytic and technical approaches to understanding that data as delivered through eIntelligence, click-stream and log analysis techniques.

INTRODUCTION

This paper tells the story of personalization and measuring business value from a developer's perspective – that is, as a pragmatist who spends his time developing strategies for personalization and their implementation through web technologies. Here we discuss the concepts behind personalization and address some of the challenges that people will find as they unearth the hidden mysteries of web log and click-stream data.

According to a Forrester Research report, "As the general content of the Web gets broader, individuals will cease aimless surfing activity and gravitate toward sites that deliver products and services customized to their needs. Sites must plan now to respond to this expectation or risk being left behind as the Web changes to a personal medium."

In the first part of this paper we will describe the business and technology reasons for personalization. Although we will discuss this concept in much more detail later, we can think of personalization as an approach to delivering content dynamically, or "just-in-time", so that the content is specific to a user or group of users. Next we explore how we can begin to understand how people use information on our web sites through measurement. We will discuss various factors about the web site experience – from the perspective of the visitor as measured through web logs and click-stream analysis.

Although this paper is intended as a resource for people who have, or are creating, personalization strategies, we recognize that many questions go unanswered. Because the web has spawned numerous approaches to solving some of the problems of a stateless environment, it also creates problems as we attempt to monitor its usage. That is to say, the technologies that we use to make web sites more dynamic and personal often create complexities when we try and figure out what people have done on our site. So the story is really a story told from two perspectives – one describes how information is delivered and the other, how we describe or analyze patterns of activities when the content is dynamic.

THE NEED FOR PERSONALIZATION

The need for personalization is paramount in the world of eOverload. Information is coming at us from all fronts – radio, billboards, web phones, wireless PDAs, the ever-looming Internet… all trying to vie for our attention. Developers of web content, whether they are marketing, technical, or hobby, all want their content to be noticed.

Site differentiation is key in the world of *e*Overload. One author (Ramsey, 2001) has suggested the following as key to obtaining site differentiation:

1. Value added information
2. Frequent, useful content updates
3. Interactivity
4. Information personalized for specific users or user groups.

The first two of these is likely the responsibility of the business side of the house – or the content experts. Technology, however, can play a significant role in making sure that the visitor is engaged once they are on the site. Using client-side tools such as DHTML, JavaScript, and Java, we can make the site extremely dynamic and interactive. Finally, our last goal – personalization – allows us to create a true one-to-one relationship with a visitor.

Web personalization allows you to have a Web site that tailors Web content to a Web user's preferences and other profile information. In addition, a personalization system logs every Web page displayed to every user so you can develop a "clickstream" view of what they saw, when they saw it, and for how long. Just imagine what you could learn about your audience with a complete understanding of their Web usage.

In our net-centric world, we have seen these in action – everyone vying for your attention – sending invitations, gifts and even eCards. In the early days of web, advertising banner ads were unusual and effective. The thought of having 6 million people look at your ad each time they logged on to check their email was fascinating. The early marketers capitalized on this new media and found it very profitable. Just by measuring click-throughs (whether a person actually acted on the ad they saw), we could evaluate the effectiveness of an ad campaign. But soon, like the advertising we see on park benches and buses, the effect had diminished.

Unlike the simple site maps of even just a year or two ago, most modern sites are not static. For example, it is not uncommon that product taxonomies change regularly, marketing campaigns or new service offerings for content updates and even segmentation of content to different groups of users. Often content is personalized – the way that information is presented, the method of delivery and

even commerce models have been impacted our expectations of content (e.g., pricing models.[1])

So what do we mean by personalization? Is it just adding the "Dear Greg" on top of a web page? Personalization can mean a lot of things in the high-tech, high-touch world of eCommerce, here we adapt the following definition from Ramsey (2001):

> *Personalization means strategically targeting specific users or groups with content relevant to them, delivered at the time and manner most appropriate to them.*

By delivering dynamic interfaces to information so that the right people have access to the right information at the right time, we can solve some of the key challenges that marketing faces.

The Business Need for Personalization

As we take inventory of the types of models that we have in the web world, we can find at least four categories that describe our use of personalization technologies. Examples of these exist both on publicly available sites as well as in custom applications that sit behind our corporate firewalls.

For example, we may find personalization being used for:

- **Technical Benefits** – Information may be stored in a data repository, making updates more efficient and universal. In addition, we may find it technically more feasible to create content on demand, rather than having information stored in static files.
- **Security** – one method of creating secure access to content is referred to as conditional disclosure. Here sites provide customized views of information resources that are specific to a user or group of users. Information may be personalized to prevent access to certain content based on the role that they have been assigned to in the application.
- **Localization** – providing web content that is in the language of the intended audience is critical – especially for global sites catering to the needs of international audiences. We also think of localization in the context of portals that provide content for a specific geography or interest area (e.g., content subscriptions.)
- **"Relationship" Management** – perhaps, the most widely held reason for creating content on demand is so that information can be customized for a specific audience to establish a personal relationship with the organization (e.g., one-to-one marketing.)

In another paper (Barnes Nelson, 2001), we discuss four major "relationship" types in the web commerce world: customers, suppliers, partners and employees. As we think about these relationships, the rationale for using these dynamic interfaces seems apparent.

[1] "Amazon charging different prices on some DVDs", Rosencranz, L (2000)

Audience	Buzz-word(s)	Purpose
Customer/ Consumer	CRM – Customer Relationship Management B2C – Business to Consumer	Create a positive method of interaction, including recommendation engines, secure access to project information, cross-selling and key customer/organization information. (amazon.com, dell.com)
Partner	PRM – Partner Relationship Management Extranet	Establish communication methods, exchange of information (e.g., customer exchange standards), specifications (documents, CAD/CAM), inventory levels, etc. (ClinicalExchange.com)
Employee	B2E – Business to Employee Intranet	Provide customized views for the employee about his/her worklife. Examples include benefits information, expense/ payroll data, Knowledge management, document management and distributed team communication. (www.schedule.com)
Supplier(s)	SRM – Supplier Relationship Management EDI – Electronic Data Interchange Extranet	Building on the reengineering efforts in the late 1980's and 1990's, just in time ordering, business process mapping, electronic data interchange standards, etc. all can be provided through personalization engines. (www.perfect.com)

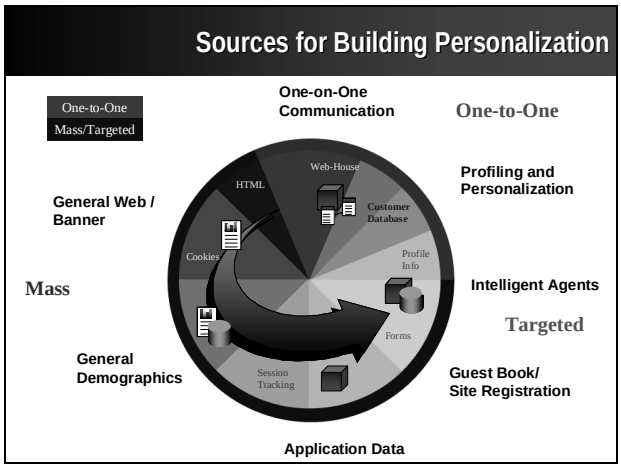
Table 1. The landscape of personalization techniques appropriate for types of business applications they intend to serve.

This table (Table 1) illustrates compelling reasons for creating interfaces that deliver information "just-in-time". In all of the cases cited here – on both sides of the firewall – content is being provided on demand. That is, there are few static HTML files that sit idly by waiting to be called. Instead there is usually some engine behind the scenes waiting for content to be requested. Usually, it is stored in a database or content repository so that information can be pulled "just-in-time". There are a number of vendors that provide content management solutions (for example, Vignette and Xpedio.)

Technologies for Personalization

As mentioned previously, there are a variety of ways that we can think about personalization. The diagram below examines this in the context of the audiences that are served and the goal of the interaction. As we move from Mass communication with web audiences to more personalized forms of interaction, we see an

increase in the complexity of the underlying technologies. As we move to one-to-one marketing approaches, a more robust personalization and profiling engine is required.



Here, we show a continuum of interaction regarding a web site visitor. As we move from the upper left around the circle to the upper right, we find the following trends:

- **General traffic patterns** - Static or Dynamic HTML is used to provide access to page views. The site may be used for informational purposes and has little or no personal interaction with the individual visitor. Gross analysis of patterns of traffic can be reporting on using web log analysis.

- **General demographics** - Cookies/ JavaScript provides the basic ability to monitor usage within a site (transient) and across visits (persistent). Cookies are used to store basic information about a visitor (computer – not a person) in order to provide some basic personalization.

- **Guest Book/ Registration** - Forms used to collect information in databases are a good source of gathering information about a visitor but often don't integrate well with web log data. Technologies such as collaborative filtering[2] can be used in response to form submissions to build dynamic interfaces. An example of this might include using collaborative filtering to recommend book choices based on what others have purchased in the past.

- **Specific demographics/ usage** - Specific utilization and repeat visit tracking can be accomplished when the site supports technologies such as session identification techniques such as JavaServer Pages (Sun), Active Server Pages (Microsoft) or SAS/IntrNet (SAS Institute). Here web server logs and application server logs can be combined to form a visitor profile. Repeat visits can be tracked as well as a comprehensive view of what actions were taken while on the site.

- **Profiling / Membership** - Customer provided authentication (userid and/or password) and personal/ membership data can be used in conjunction with intelligent agents to provide a personalized view of a web site. Intelligent agents can be used to filter information and provide the user with the content they desire. They work on behalf of the user observing their preferences and site interaction habits.

- **Integrated 360° View** - Internal databases that collect information from multiple touch-points such as call-center, sales, web, etc. are combined to create a true view of the visitor.

As you can see from this brief summary, there are a variety of technologies hard at work for us that provide these personalized, dynamic interfaces on the web. Now lets examine these technologies from the perspective of its participant.

Active versus Passive Participation

Up to this point, we have talked about personalization as a general concept. It may be helpful to understand it in the context of what the user or visitor does or sees. We differentiate here between some active participation in receiving dynamic information versus those that are provided by the application "auto-magically".

For anonymous users (based on IP address), we can simply evaluate click-stream or web log data to understand macro-level traffic patterns; for permission-based users (those providing information via cookies or server-side session state management), we can understand patterns of usage and repeat visits through more advanced analytics; and finally, authenticated users – those that give us some very specific information about who there are – we can provide very rich scenarios of behavior. Let us now examine these in turn.

Personalizing Content for the Anonymous User

For "un-authenticated user, we can indeed provide some form of personalization. From a technical perspective, we can do this through information provided from the browser, through cookies as well as through the use of forms.

Browser Sniffing: Who is that computer behind the visitor?

Because the web is digital, it may seem obvious that we ought to be able to get a lot of information from those that browse our sites. It is true that as you traverse the web, we do leave a virtual trail. That is, depending on how you access information on the web, what browser you use, what technologies sites have for you and the ISP you use all seem to leave our digital footprint – and in some cases, lead directly back to you.

Common examples of how we might use this information can include:

- Browser and Version (e.g., Internet Explorer, Netscape)
- E-mail address (if provided in your browser software)
- IP address (depending on how you connect, this may be yours, your ISP or even your firewall)

These are just a sampling of the types of information that can be obtained by using just standard, out of the box settings on a web server. More sophisticated web servers can provide a much more complete profile of who you are (as represented by your browser). A complete listing of the information that is available to most web servers can be found at Brian Lavoie's site (Lavoie, 2001).

[2] Systems that allow for the presentation of information based on what others have chosen.

Cookies: Unsolicited Personalization

Cookies are simply a method of storing basic content about a visitor to a specific web page (or site) such that the information can be retrieved at a later time. Cookies are used to store information such as the user's name, the last time they visited the site or demographic information (i.e., personal and technical information – such as the browser). Cookies are usually stored on a user's computer without their explicit permission.

There are two types of cookies that can be used to store information about a visitor, depending on how long you want to retain the information about the visitor, transient and persistent.

Transient, or a session level cookie, is placed in the users browser and lasts as long as the browser is open. This serves as a temporary ID for the browser and to any application servers that request the cookie. This method is used to associate data from click-stream and application server log processing.

Persistent cookies are similar to transient cookies, except these can be read by entire domain (e.g., sas.com) and can be used throughout applications across an organization if common conventions are used. Persistent cookies can be used to "persist" information across multiple sessions or visits.

There are many issues surrounding the use of cookies, but these are generally a safe and effective way to capture information about a visitor (computer). The main advantage of using cookies is that they are easy to program and can be implemented quickly.

Cookies also suffer from several disadvantages.

- There are issues about how and what information we collect about people – that is, the social/ethical/perception issues surrounding their use.
- Cookies have a limit to how much data they can hold.
- Most significantly, though, cookies lack portability: the statefulness of cookie data is tied to an individual computer, rather than an individual person. When that person visits your site from a different computer, they have no access to their personalization settings.

Transient cookies cannot be used to track multiple visits/ repeat visits. In addition, Users may turn off or destroy cookies. From an organizational perspective, enterprises that wish to use them across all web touch-points should provide some standards and management of cookie signatures. On some operating systems (e.g, Windows ME and 2000) that allow for multiple users, a clear understanding of who the user is and the interpretation of a *visitor* versus a *user* versus a *household* should be clearly understood. In addition, the same user may visit from multiple machines making it impossible to really track a specific person.

Forms: Anonymous Solicitation

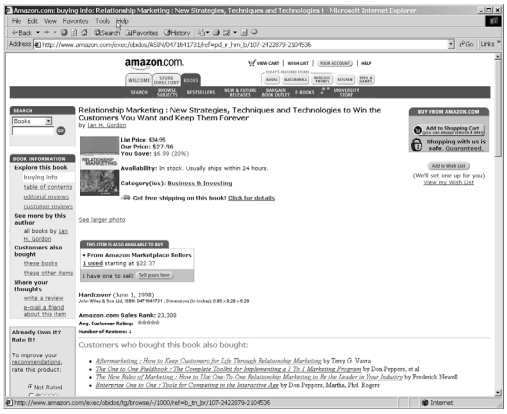
Forms, as you might suspect, gather specific information from anonymous visitors, and give something back – usually a request for information, a search or a subscription to content.

Many of us have seen this is action when we visit a web site to request product information and find that they have seemed to understand us by giving us recommendations of what others have purchased/ liked.

These sites have probably used some form of collaborative filtering. Collaborative filtering combines preferences and interactions of similar users and applies it to a new user/ request. This approach takes user built profiles in combination with system-generated models of how other "like" users look. The content is then provided to the user as part of their "personalized" view. There are at least three different types of Collaborative Filtering in use today:

- *Automatic Collaborative Filtering*, where a system modifies or customizes the interface and content offered based on understanding the preferences and usage patterns of other users within similar and behavioral characteristics.
- *Active Collaborative Filtering* is based on voluntary inputs from the community of users. ACF is based on permission-based marketing and can help determine which content is most relevant to users. As content is based on what users actually say they want versus modeled content.
- *Expert/ rules based filtering* is a highly sophisticated technique that allows the system to make deductions or inferences about a visitor based on built-in experiences and knowledge (usually stored in a database.)

The screen shot below from Amazon.com shows an example of collaborative filtering in action. Note the section displaying what others have purchased.



Like Amazon, most of the e*tailors on the market make use of third party application providers. Examples of vendors that provide collaborative filtering software include: Net Perceptions, Broadvision and Like Minds.

Anonymous Session Management

Because of the limitations of cookies for storage of data – either within a session or across sessions – vendors have developed server-side[3] solutions to manage this problem. Server-side solutions that allow for the persistence of information across multiple pages within a web site are referred to as session ids or session management solutions. Usually, these work in conjunction with cookies or authenticated forms of permission. When the visitor first

[3] Server side refers to the fact that the information about the session or user visit is stored/ generated on the server, not on the client (as in cookies).

visits a web site, the initial page sets a Session ID that allows the server to track session "state" or persistence across multiple pages or even multiple visits to a web site. Often these session ids are included as hidden fields within a page. As the information is requested from the server, the session id is validated and the page is generated on demand using a server side technology such as Micorosoft's Active Server Pages or Sun Microsystem's JavaServer pages.

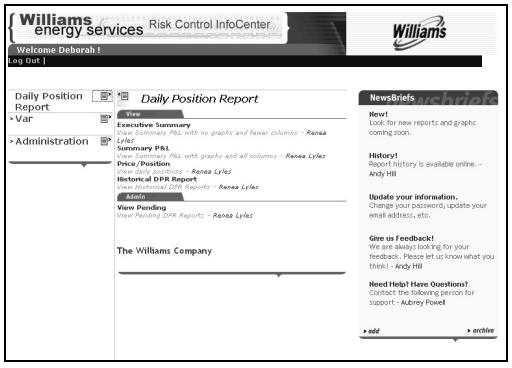
There are a few problems that developers may find when they wish to utilize these approaches. Namely, users that "bookmark" pages may find the page unusable at a later visit because it may have contained a hidden variable (sessionID) that has expired. In addition, caution must be taken to ensure that sessionIds can be used between applications in an organization/ web farm. Since many organizations use different tools and technologies – even within their own external web pages – users may find that they are not "remembered" even when directed from the customer service area of the web site to technical support.

Understanding the Authenticated User

Authenticated users are those that provide some sort of validation or verification of the individual. Authentication may be as simple as extracting the userid from the host (for example, the Microsoft NT userid is available to dynamic applications that request this information.) These applications represent the most reliable method for remembering users and providing content management services. These applications are often most appropriate within the firewall (intranet) or as secure connections between organizations (extranet).

Typically, authentication can come in several forms. We described one scenario of obtaining the domain userid from the operating system, but you may also have a custom logon screen, which requires a userid and/or password. In addition, SSL (secure sockets layer) can be used to authenticate a user to ensure that they are who they say they are.

We think of these as user supplied authentication where the user provides some level of information in order to receive content appropriate to them. Examples of these include commercial portals (myYahoo.com; myAOL.com); industry portals (ClinicalExchange.com); or corporate portals (Plumtree, Viador, Hummingbird). In each of these cases, the visitor is known and the content is based on their preferences (content subscription) or privilege (security). The screen shot bellows shows us an example of an Intranet page that offers conditional disclosure based on security roles.



Technologies for Authenticated Users

All of the approaches described earlier for managing and presenting content to the anonymous user can be used for the known – or authenticated user. For example, we can store information from page to page or even from visit to visit through the use of both client-side (browser, cookies) and server side (sessionID) solutions. In addition, since we know something about these people, we have the ability to store information about their likes, dislikes, demographic information, purchase history, favorite reports, etc. As a result, the possibilities for how we generate content for these audiences are endless. Most often these solutions rely on enterprise database management scenarios that allow for the integrated of corporate data and web commerce data (e.g., web log and click-stream data).

EXAMPLES OF PERSONALIZATION

As we discussed, there are a variety of approaches to personalization. In this section, we will examine these in the context of SAS. We now take a complete example from beginning to end – showing how we can use SAS to understand whom the user is – from anonymous, permission based and authenticated.

First, we will show how you can get a variety of information about the visitor just by using information cleaned from the browser. Second, we will show how we can set a cookie so that we can use that within a session (from page to page) and then again on a repeat visit. Finally, we show an example of how we might authenticate a user and then present information to them based on who they are.

Please note that these examples are simplified and are not meant to show how SAS can be used fully to exploit techniques for personalization.

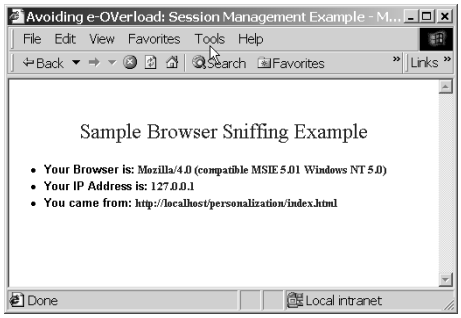
Reading Browser Information

At its simplest form, understanding a visitor in terms of their browser, gives us some really good information about who a user is – as represented by their computer/ browser. In this example, we set up a simple page that shows us some information like what browser they are using and their name if available.

The goal of this paper is not to show users how to set up and configure SAS/IntrNet. But a word about some of the variables that are available to us through the broker is important. The broker – or application dispatcher – is a CGI (common gateway interface) program that allows us to communicate with SAS through a web request. By default, many of these variables (about the browser) are commented out. Lets take a look at some of the browser/ client specific variables. This partial list is taken directly from the broker.cfg file.

```
# User's IP address
  Export REMOTE_ADDR        _RMTADDR
# Username if authenticated
  Export REMOTE_USER        _RMTUSER
# Browser name
  Export HTTP_USER_AGENT    _HTUA
# Referring page if known
#  Export HTTP_REFERER       _HTREFER
```

The following screen displays the results of a simple program that uses these variables in formatting the screen. Specifically, we have identified the name of the remote user and their browser version.

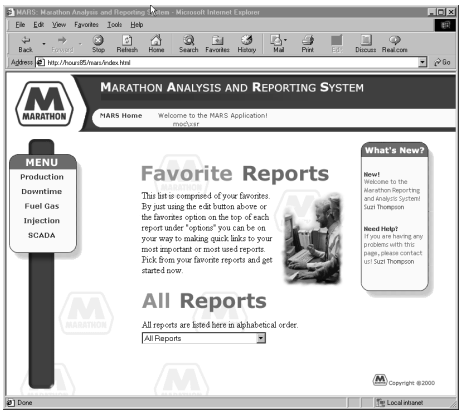


Of course, we could accomplish the same effect through the use of JavaScript:

```
<SCRIPT LANGUAGE="JavaScript">
document.writeln(parseInt(navigator.appVersion));
document.writeln(navigator.appName);
</SCRIPT>
```

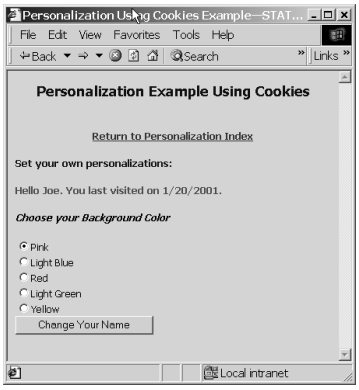
However, in dynamic SAS applications, there may be situations where getting this outside of the browser and passing it on application logic/ code, would be useful. A good example of this is for applications that might pull the current userid (from the Microsoft Windows NT login) is pulled from the operating system environment variable and identified in the first screen the user comes to.

In the following example, we show an example of personalization based on what we know about the user instead of what the user tells us. Here, we allow the user to save reports (a.k.a. My Favorites) within the application – but not force them to logon with yet another userid and password.



Forms, Cookies and JavaScript: Remembering People

In this next example, we want to provide some familiarity by setting a cookie when they first come to our site. Once visited, upon return visits, we either say Hello <name> (if the cookie exists) or simply "Hello" if there is no cookie set. We also show how you set the color of the background color and then remember that color when they return.



As we discussed previously, cookies can be very useful – especially when we combine it with more advanced techniques such as forms. In this example, we will have a user fill out a form, then remember this information so that it can be used the next time they visit. In this case, we take advantage of JavaScript to check whether or not the cookie is set, then depending up whether the value is available, we display information relevant to them.

Server-Side Session Management

Up to this point, we have relied on the browser and its ability to remember people from session to session by storing information on the client. There may be times, however, when you either cannot or do not wish to rely on the browser or cookies to store this information. Recently there have been numerous technologies that have been introduced that allow us to remember visitors from page to page and even from visit to visit. We can do this through the use of session management techniques.

Session management can be accomplished using a variety of technologies – some of these include: Microsoft's Active Server Pages, Sun Microsystems JavaServer Pages, Haht Software's HahtSite. Indeed, all of these server-based solutions could even exploit SAS data and compute services. For simplicity, we will focus on two methods that are provided for with SAS technologies: SAS/IntrNet sessionID and session management using Java through SAS' AppDev Studio. Note that with both of these approaches, we could use either client side (cookies) or server side (sessionID) – we have decided to limit our discussion here to just server-side approaches.

SAS/IntrNet: Application Dispatcher Session Management

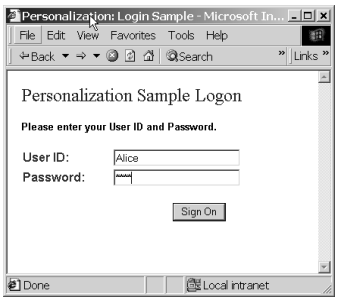
Like most web application servers, SAS/IntrNet affords us the luxury of being able to manage state within a web application by persisting information through a sessionID that resides on the server (or in cookies). Sessions allow us to save information such as temporary

datasets and variables just as we would typically save data in WORK libraries and in macro variables.

Macro variables can be saved for use in subsequent requests by prefixing them with the word SAVE_. Similarly, datasets can be persisted if you use the library SAVE instead of WORK.

Let's explore an example. In the series of screens that follow we perform the following actions:

1. The logon page passes the userid and password to the SAS application dispatcher through a POST method. (logon.sas)



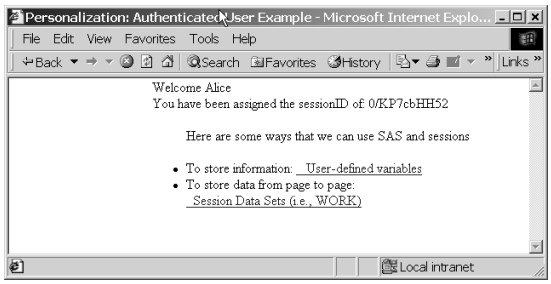
2. The logon.sas program reads the userid and subsets the sample dataset using the userid in the where clause. That dataset is saved using the SAVE.* method – which signifies to SAS that we want to save the data beyond this page.

```
Data save.mydata;
  set pdata.class;
  where upcase(name)=upcase("&userid");
```

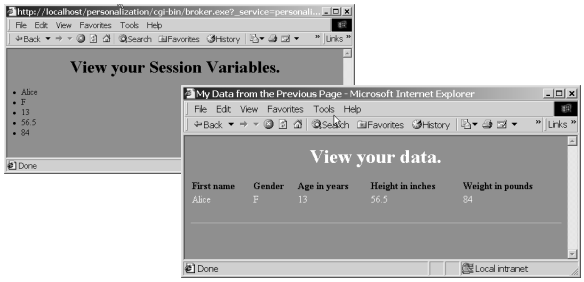
3. In that same step, we assign several macro variables based on the values being read in. These macro variables (SAVE_varname) are also available to subsequent calls.

```
call symput('SAVE_NAME',name);
call symput('SAVE_SEX',sex);
call symput('SAVE_age',age);
call symput('SAVE_height',height);
call symput('SAVE_weight',weight);
```

4. Finally, we build two hyperlinks dynamically so that we can see our sessionID in action. The sessionID is added to the link so that we have access to the SAVE. Datasets and the save. Macro variables.



And view the two subsequent pages where we pull the information based on the session information (as passed through the sessionID).



By take advantage of the server to manage session information, we have reduced he potential problems usually associated with using cookies. In addition, we have eliminated most of the effort in constructing construct complex URL's with parameters in order to pass information from page to page.

In addition, we can now also take advantage of the session to store information about what people have selected and/or saved through by intercepting each REQUEST and programmatically processing the information contained in the request – including writing the session information to a log. We would do this by modifying our appstart.sas program (the Application Dispatcher program that provides the service for our SAS/IntrNet programs). Here we would add a REQUEST statement – specifically an INIT program that would run each time a request was made.

Active Server Pages, JavaServer Pages and AppDev Studio

JavaServer Pages was introduced by Sun Microsystems in response to Microsoft's Active Server Pages as a technology to allow for the generation of HTML-based content in web applications from Java. JavaServer Pages is a special type of servlet (Java server program) that generates content on demand using the Java programming language. Similarly, Active Server Pages is a server side language that produces content from the server. Active Server Pages uses VBScript primarily as it's programming language. However, we can also take advantage of COM/DCOM for storing program logic on the server.

In each of these languages, we have the ability to maintain session state or persistence, using the session object. The session object, much like SAS/IntNet, allows us to persist information from page to page. We do this by establishing a sessionID when we first start out application. From there, we simply pass the sessionID from page to page and it "remembers" information despite the "connection-less" state that HTTP protocol provides.

THE PROBLEM WITH PERSONALIZATION

In the first part of this paper, we explored some of the business and technical reasons that "personalization" is now part of our every day vocabulary. Now we will discuss some of the challenges that we face in measuring the effectiveness of these strategies.

Earlier we spoke of these technologies from two perspectives. The first, our ability to generate content on demand so that visitors to our web sites have a much more personalized experience. The second is technologies that we use to understand what people do with that information. So now we turn to from technologies and approaches that deliver web content to those that help us understand its effect. That is, did we accomplish our objective with this web site? Do

people buy more? Do they stay longer? Do they come back to the same places? Will our servers survive the traffic?

The Need

Today's business and technology landscape is replete with reasons for personalizing content. In order to measure the effectiveness of these approaches, a methodology is needed, which allows us to determine the effectiveness of a site – in terms of the business value. Today's analysis techniques for reporting about site traffic, such as the most popular pages – tell us little or nothing about things that drive the bottom line. The business decision makers want reporting against profitability, customer satisfaction and return on investment.

e-Marketers who must make decisions about how to organize and present content on the web need to know such things as: What pattern of behavior by a web site visitor is followed by a high-margin product purchase. And which changes to the web site facilitate such behaviors.

Market for web activity analysis solutions, profiling data and auditing services has exploded. The business need requires an intelligent system capable of aggregating and analyzing activity on infinitely variable content in a manner that is meaningful.

Behavioral Indicators

The question of why people buy the things that they do has been at the forefront of the minds of researchers for decades. In the 1950's, folks like Louis Cheskin attributed the reasons to the way it made people feel. However, others found that attitudes and behaviors often are incongruent (see for example, Aizen & Fishbein, 1975). If we can really measure what people are doing, rather than merely getting their opinion on the subject, is much more reliable. So it may be that we measure people's traffic patterns through our web sites because we can, but it is also that is gives us a tremendous amount information about behavioral patterns which can then be used to make business decisions that directly affect the bottom line.

Web Log Analysis

One of the most popular techniques for analyzing web activity involves reporting on web logs. Web logs helped us understand such things as: Where did people (visitors) come from? Where did they jump off? Where did they go when they were there? How long did they spend at each place? Did they come back?

Web activity reporting consists of:

- Basic traffic statistics (hits, page views, visits)
- Navigation patterns (referrers, next-click, entrance and exit pages)
- Content requested (top pages, directories, images, downloaded files, stickiness - measuring depth and persistence)
- Visitor information (domains, browsers, platforms, time of day)
- Fulfillment of the site's objectives (purchases, downloads, subscriptions)

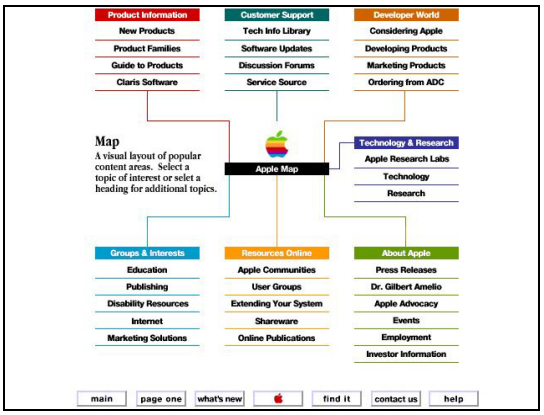
Answering questions such as these are critical as companies are pouring huge capital investments in the web-front experience. Help with decision making about content activity – gives companies quantifiable measurements as to the type of content that is being requested.

Analysis tells an organization about their visitors – just like as presenters, we try – as we may – to understand our audience so that we can tailor content to the audience – tracking helps us understand who is it that is requesting information. Questions like: audience composition, how do they match our models of a customer, how does it change over time, what improves visitor retention or session depth?

***e*-Intelligence**

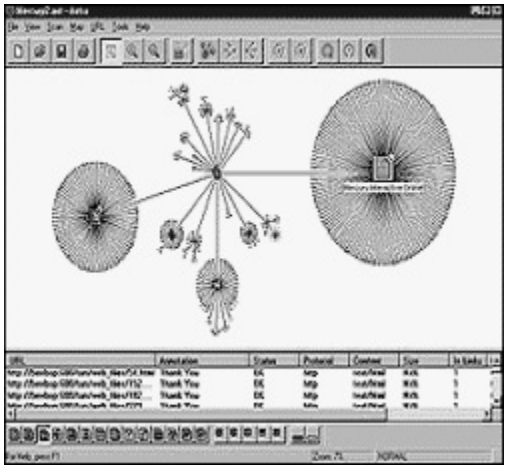
One of the most common techniques for understanding how people have interacted with a web site is through the use of web log analysis. That is, web servers produce a tremendous amount of data about who clicked on what, where. These techniques were adopted early in order to unearth the hidden mysteries of performance, traffic flow, on-ramps and off-ramps and buying patterns. However, in this context, we must move beyond traditional web log analysis and focus on how we can use the data coming from our web sites when the content is continually changing through personalization techniques. The holy grail of marketing is, after all, delivering your message in a manner that is compelling enough to create a personal relationship between an individual and the organization providing the message.

With these technologies, comes a price. That is to say, the more dynamic we make the content, the harder it is to track and monitor what content is being viewed. For example, given a typical web site, we are usually aware of the possible destinations.



Of course, the paths that one could take within this site could be enormously complex. However, technologies that allow us to map these pathways and make decisions about site design, buying patterns and visitor information are fairly straightforward. For example, WebHound™ (SAS Institute), provides compelling views about our web site, allowing us to determine which areas are most often visited, which areas lead to the purchase decision most often as well as the ability to understand complex patterns of traffic flow within our site.

Visualizing this content and the pathways are often the most useful technique. Below, is a sample screen from one of these tools (Astra Site Manager.)



These software tools are designed to aid their masters in the visualization and management of large, complex web sites. More often than not, however, these tools are designed to understand and capture information about data that is slowly changing.

The Challenges

Most often, the first stop on our way to understanding what people have done on our web sites begins with web log analysis. As it's root, web log analysis is a stream of data that is generated when a user does something on your site.

If we examine a typical web log, we see that it is simply a trail of activity. The unit of analysis of this data is a computer (represented by an IP address) and a request (for an object such as a page or an image).

```
202.188.93.150 - - [26/Dec/1999:04:31:32 -0500] "GET /images/1sdex.gif HTTP/1.1" 200 967 "http://ecommerce.nc
202.188.93.150 - - [26/Dec/1999:04:31:32 -0500] "GET /images/tac.jpg HTTP/1.1" 200 1596 "http://ecommerce.ncs
202.188.93.150 - - [26/Dec/1999:04:31:35 -0500] "GET /images/gvr.gif HTTP/1.1" 200 41 "http://ecommerce.ncsu.
202.188.93.150 - - [26/Dec/1999:04:31:35 -0500] "GET /images/spacе.gif HTTP/1.1" 200 37 "http://ecommerce.ncs
202.188.93.150 - - [26/Dec/1999:04:31:36 -0500] "GET /topics/images/thumbnails/tn_businessmodel.gif HTTP/1.1"
202.188.93.150 - - [26/Dec/1999:04:31:38 -0500] "GET /images/logos/wachovia.jpg HTTP/1.1" 200 8848 "http://ec
202.188.93.150 - - [26/Dec/1999:04:31:38 -0500] "GET /images/gvr350.gif HTTP/1.1" 200 57 "http://ecommerce.nc
202.188.93.150 - - [26/Dec/1999:04:31:39 -0500] "GET /images/lquote.gif HTTP/1.1" 200 147 "http://ecommerce.n
202.188.93.150 - - [26/Dec/1999:04:31:39 -0500] "GET /images/rquote.gif HTTP/1.1" 200 151 "http://ecommerce.n
202.188.93.150 - - [26/Dec/1999:04:31:40 -0500] "GET /images/vertical.gif HTTP/1.1" 200 390 "http://ecommerce
202.188.93.150 - - [26/Dec/1999:04:31:41 -0500] "GET /images/edge3.gif HTTP/1.1" 200 100 "http://ecommerce.nc
202.188.93.150 - - [26/Dec/1999:04:31:41 -0500] "GET /images/horizonal.gif HTTP/1.1" 200 256 "http://ecommerc
202.188.93.150 - - [26/Dec/1999:04:31:42 -0500] "GET /images/l_elbow.gif HTTP/1.1" 200 87 "http://ecommerce.n
202.188.93.150 - - [26/Dec/1999:04:31:42 -0500] "GET /images/small_logo.gif HTTP/1.1" 200 2278 "http://ecomme
202.188.93.150 - - [26/Dec/1999:04:31:44 -0500] "GET /images/ban_logo.gif HTTP/1.1" 200 813 "http://ecommerce
202.188.93.150 - - [26/Dec/1999:04:32:37 -0500] "GET / HTTP/1.1" 200 3476 "http://partners.wamma.com/SurfFast
202.188.93.150 - - [26/Dec/1999:04:32:50 -0500] "GET /images/ec.jpg HTTP/1.1" 200 0 "http://ecommerce.ncsu.ec
202.188.93.150 - - [26/Dec/1999:04:32:53 -0500] "GET /ec.html HTTP/1.1" 200 7762 "-" "Mozilla/4.0 (compatible
202.188.93.150 - - [26/Dec/1999:04:32:54 -0500] "GET /images/ec.jpg HTTP/1.1" 206 0 "http://ecommerce.ncsu.ec
```

There are several challenges that we face as we deliver dynamic content to users. These include:

- **The problem of the dynamic "path"** – One of the challenges that we face when we deliver personalized content to a user or a group of users is that our notion about the "path" that they have taken to get to a certain point has changed dramatically. Each visitor may have a completely different path that makes traditional analysis difficult, if not impossible.

- **Making complex URLs Meaningful** – You may have noticed that while visiting a web page, the URL (the address of the query string in your browser) that appears as a long string of made-up numbers, letters and special characters. Often these are used to pass critical information to the server as you request specific information. An example of this can be found in even the simplest web application (this one is from the SAS Multidimensional Viewer.)

  http://10.133.1.7/sascgi-bin/broker?metabase=SASHELP.MBEIS&_program=sashelp.webeis.mddbrpts.scl&_DEBUG=0&VMDOFF=y&_service=default&mddb=SASHELP.PRDMDDB&css=%2Fsasweb%2FIntrNet8%2FMRV%2Fcss%2Fdefault.css

  As requests like these get made – no matter how similar – they appear in the web log report as different pages even though they may be the same request with a slight variation. For example, in the following query, we are requesting a sales report for the North East Region, sorted by County. In another request, we ask for the same data, only sorted by State. As these get processed by the web reporting tool, they are reported on as two separate pages.

- **Multi-source data integration** – as data from multiple servers are aggregated to provide a single, consistent view of the web activity for an organization, several problems can arise. For example, there are issues such as time-synchronization (multiple time zones) or tracking session variables across servers that may have different sessionId management standards. In addition, because of the pure volume of data/ interactions, it may not be feasible to report on all of the data that we would like.

- **Integration of Application Server and Session logs** – Mapping what's possible to what the user actually experiences often mean combining web server information, session logs and application server logs to create a logical mapping of the user experience.

- **IntraPage Analysis** – In order to make a web page more dynamic and interesting to users, we have adopted the use of JavaScript, XML, Java applets, ActiveX controls and other plug-ins. However, since non-HTTP protocols are not tracked in the server logs we have no way of understanding these client-side interactions.

- **Changing Hierarchies and Product Taxonomies** – Because content on the web can be changed often and even for each user, we have to take into account how we are going to report on this data in reaction to changing requirements of the site (movement and restructuring of pages, reorganization of products and product categories, visual representation of products, etc.)

Traditional web tracking and reporting tools are not prepared to deal with these complexities. Web server logs are designed to track the query string – not session or intra-page activity. Most often, web log analysis tools use IP matching to report on web activity. IP Matching consists of mapping time-contiguous server log entries from the same host ID in a short period of time.

Since IP Matching can only provide gross statements about traffic patterns, there are three major issues with this approach. (1) This technique tends to be inaccurate with web sites that have a sizeable number of visitors since IP address can be reused within a short period of time. (2) IP addresses may be used by the same user within the same session for the same user. (3) Firewall protection yields single IP addresses for a group of users in an organization (e.g., AOL).

These approaches are good for sites were there are few dynamic components because we are analyzing information after the visitor has left (i.e., web logs), there are no special requirements on the web server or the browser (except that the web server can generate the logs.)

Dynamic sites, on the other hand, have complex reporting requirements if we want to understand the entire picture. For

example, we may find ourselves combining the information from a web server with the application log to get a true picture about what has happened.

CONSIDERATIONS FOR E-INTELLIGENCE

If we really want to understand what has happened on a web site, it is critical that we understand several things.

First: We need to understand the difference between an object requested, a page and a visit. We typically look at objects as individual components that have been requested on a page (e.g., image, html). However, more meaningful content for analysis includes both the "Page" and the "visit" or Session. For analysis of site flow and pattern analysis, aggregating information at a page level is key. Integrating information about the context in which a page is viewed is critical for truly moving beyond the "river of clicks" and onto descriptions of behavior. Here, we combine server log information with application level content such as session state information.

Data must be mapped and aggregated at a level that is meaningful in the context of the business: server typically logs clicks and keystrokes (object level) – but basic analysis begins with page level experiences – product experience.

Mapping what's possible to what the user actually experiences often mean combining web server information, session logs and application server logs to create a logical mapping of the user experience.

Sometimes, a group of pages should be aggregated to understand the "super-ordinate" goal of a promotion or campaign.

Data Mining

From a technology perspective, data mining has certainly blossomed in its relevance for helping us understand the vast amounts of click-stream data. By applying data mining techniques such as market basket analysis and other association techniques, marketers can find a virtual gold mine in their data. Second generation mining techniques, applied to the web, evaluate not only end of line tracking – composed of tracking behaviors of a person in a store/ web site – but also what to display next. Good examples of these include Amazon.com's recommendation engine. The movement from traditional analysis (what happened) to "what's going to happen next" has taken eIntelligence to new way of thinking about the question "why do people buy the things they do?"

CONCLUSION

In this paper, we have explored the world of personalization in terms of its business drivers, the technologies for generation and techniques for measurement. The world of dynamic applications – where content is delivered to users based on who they are, what they like and what they have access to – is quickly evolving. No doubt this world will continue to change – both as a result of the technology but also because of the social and political landscape that governs its acceptance. As technologies such as XML and wireless communication become commonplace, our ability to deal with the information overload will become more important.

Equally as important is how we will deal with the measurement issues as we see the heightened visibility of privacy. Clearly this evolving landscape of politics, business processes and technologies will continue to raise questions along the way. However, personalization coupled with tracking provides us with a powerful toolbox that allows us to understand people – not just the technology – of our "webscape". Personalization can help your find out what makes your audience "click", what works and what doesn't.

BIBLIOGRAPHY

Fishbein, M., & Ajzen, I. (1975). Belief, attitude, intention, and behavior: An introduction to theory and research. Reading, MA: Addison-Wesley.

Lavoie, B. (2001) "Web Characterization Project", http://wcp.oclc.org/

Ramsey, C. (1.15.2001) "Managing Web Sites as Dynamic Business Applications" The Internet Industry Portal, http://idm.internet.com/articles/200006/wm_d.html.

Rosencranz, L. (9.5.2000) "Amazon charging different prices on some DVDs," Computer World on-Line, http://www.computerworld.com/cwi/story/0,1199,NAV47_STO49569,00.html.

Barnes-Nelson, G.S. (2001) " Collaborative Commerce: Portals for Decision Support and Knowledge Management," *Proceedings of the Twenty-Sixth Annual SAS Users Group International Conference, Long Beach, CA*.

CONTACT INFORMATION

The authors may be contacted as follows:

Gregory S. Barnes Nelson
STATPROBE Technologies
117 Einburgh South, Suite 202
Cary, NC 27511
Internet: greg.barnesnelson@statprobe.com
Personal: gregbn@ix.netcom.com
Web: http://www.statprobetechnologies.com

For the latest version of this paper, please refer to:
http://www.statprobetechnologies.com/downloads

Knowledge Management Using an Expert System Written in SAS®

Anthony M. Dymond, Dymond and Associates, LLC, Concord, CA

ABSTRACT

Expert systems are a branch of artificial intelligence that encode the knowledge and performance of a subject matter expert. An expert system provides a powerful approach to managing and applying knowledge to the business. The paradigm here is to encapsulate knowledge in an automated "expert" who can utilize this knowledge for the company's benefit.

An expert system can shield a company from employees leaving with critical information. It can help make knowledge available throughout the enterprise and help solve the problem of "knowing what we know." It can implement business rules to bridge the gap between written policy and practical application. It also allows importing automated subject matter experts when needed.

This paper describes an expert system written in SAS and reviews some of the major design considerations. Several knowledge bases are demonstrated to show how an expert's knowledge is encapsulated in a knowledge base and applied at run-time.

INTRODUCTION

Knowledge as an Asset

Companies have spent time and money discovering important information about themselves, their customers, and their competition. They have invested in what SAS calls "The Power to Know." Once this knowledge becomes available, it has to be leveraged to affect the corporate bottom line. In a sense, data warehousing and data mining are "mid-game" activities. They help to make knowledge available. Knowledge management, where knowledge is applied to the company's business activities to produce tangible results, is the "end-game." This knowledge is more than an accumulation of facts. It is also the rules and procedures needed to make decisions.

One example is Customer Relationship Management (CRM), which continuously acquires information about customers and their behaviors. Knowledge is analyzed and used to sell products and services, often by tailoring offerings to individual customers. Knowledge can also be embedded in intelligent interfaces, allowing customers to be guided through personalized product exploration and customization.

It is also important to make some of this information available to stakeholders such as suppliers. This can improve supplier relationships by controlling prices, inventories, and deliveries. In fact, there may be many localized areas both within and outside the company where information needs to be packaged and presented.

There are many ways companies can utilize knowledge. Knowledge can be summarized in reports or procedures. It can be stored and utilized through the expertise of employees. It can also be captured in a computer program (Figure 1).

Each of these approaches presents problems. Reports are filed and forgotten. Employees leave and take the company's knowledge with them. Procedures can be changed without adequate documentation. Computer programs are often incomprehensible and inherently lack flexibility.

It would be advantageous if this knowledge could be stored in an automated system in a more flexible and understandable form than conventional programming. Some of the attempts to do this are referred to as "business process management" or "business rules." These rules are interpretable by computer applications and directly drive computation. They are also understandable by nonprogrammers. In the ideal case, business units themselves can mostly maintain these rules without IT support. The business units become empowered and are more responsive and adaptable. Efficiency improves since there is less overhead spent on complex code maintenance and documentation.

The next step after automating business rules might be to capture the knowledge and behaviors of a subject matter expert and place these into an automated system. This is referred to as an “expert system.” Interacting with the expert system provides an experience similar to interacting with the human expert. This approximates the human expert being continuously available. This intelligent agent then reflects the best expertise the company has in a given subject area. This can offer a significant competitive advantage when dealing with customers, suppliers, or other business issues. Automated experts can be made available

throughout the company as well as to suppliers and customers. Additional expertise can be brought into the company by acquiring expert systems from external sources.

Knowledge today is viewed as one of a company's most valued assets (O'Dell and Grayson, 1998; Stewart, 1997). It must be tended and enhanced like any other asset. Knowledge management is more than storing and exploring data. It requires finding, deploying, and applying both facts and rules to obtain competitive advantage. Embedding it in intelligent systems can significantly leverage knowledge.

Current Enterprise Computing

The current enterprise computing environment is not particularly supportive of knowledge management. Applications are comprised of focused commercial packages, locally developed programs, and large application suites covering entire areas such as finance or manufacturing. Many of these applications, whether in-house or commercial, are basically procedural and provide a set of predetermined interfaces to an underlying database. They tend to be oriented to batch processes and are often inflexible.

Isolated legacy systems continue to hide both data and procedures behind a wall of incompatibility. Corporate mergers have complicated the situation with a mix of software and hardware that is difficult for IT to manage and for the business units to utilize. Many of these applications do not integrate well at the corporate level.

At the same time, the size and complexity of projects are increasing. Development teams charged with these projects can have multiple subject matter experts and may include members from foreign offices who struggle with language and cultural barriers. Traditional programming tools such as system development life cycle design and procedural code are proving to be inadequate.

Emerging Enterprise Computing

Corporations are beginning to look at their applications from an enterprise-wide perspective, rather than depending on a piecemeal approach. They seek integrated systems that support data interchange and common standards. They recognize that software needs to be flexible to meet changing business requirements. Software should also provide both event-driven and batch-oriented features.

There is a shift to object-oriented programming for large or complex projects, especially projects that need to be addressed by multiple task groups. Object-oriented techniques effectively decompose a subject or process into modular components. These object modules can be designed to be compatible with applications across the enterprise. They improve reliability and maintainability. They support reuse at both the object and method level.

Separate programs within the objects are contained in labeled blocks of code called methods. Object-oriented systems have other valuable features such as abstraction and encapsulation that isolate object internals from the rest of the system. Of particular importance is method inheritance, which provides a set of rules for sharing methods throughout a system or the entire enterprise.

When built to be compatible with industry standards such as CORBA, the object-oriented model provides the means to execute methods in diverse applications throughout the enterprise and to support information interchange regardless of the underlying data structures.

Use of object-oriented systems sets the stage for intelligent systems. Intelligent applications drive visual and conversational interfaces with embedded intelligence, allowing the application to function as an assistant. Intelligent agents provide a semiautonomous application that acts on behalf of the user.

INTELLIGENT SYSTEMS

Procedural programming separates program code from data. Particular attention is placed on developing a complex control structure. The control structure code is intertwined with the knowledge and rules contained in the program, making programs difficult to maintain or modify. Procedural programming focuses on how things should be done, rather than what should be done.

Object-oriented systems decompose the problem to modules called objects that combine both programs and data. But object-oriented systems have little to say about control structures and provide a somewhat static view of a problem.

Intelligent systems today start with the object-oriented model but add a separate control structure. The advantages of the object-oriented model are kept, but knowledge and expertise are now separated from application control. The systems become easier to understand and faster to construct (Jackson, 1999).

A separate control structure encourages the use of a declarative approach that focuses on what should be done, rather than emphasizing the details of program control. Stating rules and other knowledge to the system accomplishes the programming. The dynamic run-time interaction between objects is determined by rules written into methods and by the separate control structure. Rules are a natural way to describe a decision process. We can say:

"IF the applicant has a good credit history AND the applicant has the resources to repay a loan THEN it is OK to give the applicant a loan."

When the control structure can be represented graphically, then the system becomes exposed and comprehensible to staff who are not programmers. By using templates and graphical tools, staff in the business units can maintain these systems in the field with minimal IT support. Intelligent systems now begin to deliver integration at the corporate level with adaptability and responsiveness in the business units.

Building an Intelligent System in SAS

An intelligent system using an object design model with a separate control system has been developed in Base SAS and SAS Component Language (SCL) (Dymond, 2000, 2001). This can be used as an independent agent or embedded into applications where it is transparent to an end user.

It uses a conventional expert system design with an inference engine, multiple knowledge bases, and a database (Figure 2). The inference engine provides both development and run-time tools. It supports the run-time environment, including the separate control structure.

The knowledge bases serve a role similar to separate applications and are executed under the control of the inference engine. A subject matter expert's knowledge is captured in the knowledge base as objects in a graph. The graph provides a mechanism to navigate between the objects, and it is implemented in this intelligent system as a search tree explored by well-known tree search algorithms such as depth-first search.

Declaring objects into the tree and requesting one of the tree search algorithms provides the designer with program control. The control system contained in the inference engine then has the information necessary to automatically provide control services. Some control fine tuning can be done through code in the methods.

The tree is a dynamic system object and system methods are available that allow detailed programmer-level control should this be necessary. Methods are available that allow one tree to load and execute another tree, providing the capability of chaining between multiple subject matter experts. The tree is also extensible at run-time, forming the infrastructure for an intelligent agent that can learn and remember.

The memory-resident database contains the data from a current session. This is a "blackboard database" that is read-write accessible by all objects in the intelligent system. Data is stored in an object-attribute-value triplet, plus an index column. The database is also a system object that can be accessed by its own system methods.

The tree, objects, methods, and database can all be saved to disk and reloaded at any time during a run. This captures the complete state of the system at any time and supports "what if" studies.

MakeLoan Knowledge Base

The MakeLoan knowledge base demonstrates the intelligent system used to implement business rules. Rules describe how a company processes a loan application. The system encapsulates the company's procedures for making loans and also embeds the best practices of a company expert.

The object decomposition and placement into a tree is shown in Figure 3. The decomposition is based on functions, and the tree is arranged based on functional dependencies and on the order objects should be encountered. The tree should be processed with a depth-first search algorithm that transverses the tree from top to bottom, and from left to right. With this in mind, it is possible to "read" the tree as follows:

Begin by gathering some information about the customer, such as demographics and the nature of the loan (StartConsultation). Decide if any loan will be made (MakeLoan). This decision will depend on the customer having satisfactory credit (CreditOK) and adequate resources to repay the loan (ResourcesOK). The resource requirement can be met by the customer having adequate income to repay the loan (IncomeOK) or by pledging sufficient collateral against the loan (CollateralOK). Once a decision has been made to make a loan, the amount to be loaned can be determined (LoanAmount). Finally, a report is prepared summarizing this loan application (Report).

The object ResourcesOK could be coded using Base SAS with system methods to read and write the blackboard database, or it could be coded by using one of the method proof templates available in the system. A code fragment from a method proof template would be:

```
*PROOF=ResourcesOKProof;
*;
*TARGET={ResourcesOK,resourcesScore};
*;
*ORCLS={IncomeOK,incomeScore}        eq 'true' ;
*     {CollateralOK,collateralScore} eq 'true' ;
*ENDCLS;
*;
*ENDPROOF;
```

which states that there is a method proof template named "ResourcesOKProof" that will attempt to find a value (either "true," "false," or "unknown") for the attribute resourcesScore in the object ResourcesOK. It will accomplish this using a logical OR test on the attribute incomeScore from the object IncomeOK, and the attribute collateralScore from the object CollateralOK.

One further programming issue should be mentioned. If the applicant's credit is not OK, then MakeLoan,makeLoanScore = "false" and the object MakeLoan and its descendents do not need to be explored further. This is communicated to the search engine by setting a search control flag attribute in the object MakeLoan. This flag would be set when the makeLoanScore attribute was set to "false," and the system would know not to further examine the object MakeLoan or its descendents.

This system can be rapidly constructed, easily understood, and maintained at least in part by nonprogrammers in the business units. The objects and their control are defined by declaring them into the tree. Within the objects, much of the programming can be replaced by filling out method proof templates. Other system methods make it easy to open pop-up windows to gather information from users.

Mammal Knowledge Base

The Mammal knowledge base contains the knowledge and behaviors of a subject matter expert who knows how to classify mammals. It demonstrates an intelligent system built by successively subclassing from a general root node (Mammal) to specific examples of animals in the leaf nodes. The objects and tree shown in Figure 4 essentially reflect a hierarchical decomposition of mammalian animals. The sense of the tree is as follows:

Suppose that one encounters a medium-sized brown animal with a marsupial pouch, hopping on large hind legs and carrying the Australian flag. To identify the animal, a search begins at the root node of the Mammal knowledge base. The search proceeds through successively lower levels in the tree and seeks increasingly detailed information. As information is gathered, some branches of the tree are found to be unproductive. Control flags set from the methods code block further unnecessary search in these branches. The focus of the search would soon be guided to the object Kangaroo.

Several points can be made about the tree and search engine:

- These searches are very efficient. Only the most relevant questions are asked in the order needed to solve the problem.
- The process is scalable and works well for trees with thousands of nodes.
- The tree can be entered at any node. For example, if it is already known that the animal is a marsupial, then a focused search can begin at the node MarsupialMammal.
- It is easy to steer the search to prove a goal node in the body of the tree or to allow it to work down to individual leaf nodes. For example, the goal might have been just to determine if the animal is a mammal without trying to identify the specific type.
- "Goal driven" or "backward chaining" search proceeds from the top toward the bottom of the tree. It is also possible to have "data driven" or "forward chaining" where data is placed into objects at the bottom of the tree and the consequences allowed to flow upward. For example, it could have been initially stated that the animal is a Kangaroo and this fact then allowed to prove MarsupialMammal and Mammal.

MonitorOK Knowledge Base

The MonitorOK knowledge base encapsulates an expert's knowledge from a service call center supporting monitors. The main point of presenting this knowledge base is to show that its design and construction is essentially the same as the Mammal knowledge base. Both progress by subclassifying from a general root node to specific instances at the leaf nodes.

In fact, MonitorOK and Mammal are essentially the same as MakeLoan in that they all demonstrate search within a bound knowledge space. In all cases, the answer is known to exist somewhere within the knowledge base, and the problem is to find it expeditiously.

The underlying commonality of seemingly diverse problems is one of the most interesting and important facts uncovered when remapping these problems to the knowledge representation shown here. They all belong to a family of similar problems, and a generic tool, the intelligent system, can address them all.

CONCLUSION

Gathering, retaining, transferring, and applying knowledge are of great importance for a company seeking competitive advantage. Knowledge management is significantly leveraged by intelligent systems, which can capture and implement business rules and the expertise of subject matter experts.

The intelligent system presented here utilizes an object-oriented programming model and adds a separate control structure. This allows problems to be entered in a declarative manner by using object decomposition and then placing the objects as nodes in a search tree. Methods in the objects can easily communicate with the search engine to provide a robust and simple control paradigm. System methods supporting template proofs and user query windows round out a system that is understandable and supportable by nonprogrammers in the business units.

A number of problems that seem to be very diverse are in fact remappable as search within a bound knowledge space. The intelligent system then becomes a generic tool that can be used to address these problems.

Building the system around object-oriented techniques allows it to be integratable and to share information with the emerging enterprise computing infrastructure.

REFERENCES

Dymond, A.M. (2000), "An Object-Oriented Expert System for the SAS Environment," *Proceedings of the Eighth Annual Western Users of SAS Software Regional Users Group Conference*, 8, 454-458.

Dymond, A.M. (2001), “Telling Aardvarks from Zebras with an Expert System Written in SAS,” *Proceedings of the Twenty-Sixth Annual SAS Users Group International Conference*, Long Beach, California. (CDROM paper 135-26).

Jackson, P. (1999), *Introduction to Expert Systems*, Harlow, England: Addison Wesley Longman Limited.

O'Dell, C.S., and C.J. Grayson (1998), *If Only We Knew What We Know: The Transfer of Internal Knowledge and Best Practice*, Free Press.

Stewart, T.A. (1997), I*ntellectual Capital: The New Wealth of Organizations*, Doubleday.

ACKNOWLEDGEMENTS

CONTACT INFORMATION

Anthony M. Dymond, Ph.D.
Dymond and Associates, LLC
4417 Catalpa Ct.
Concord, CA 94521

(925) 798-0129
(925) 680-1312 (FAX)
amdymond@dymondassoc.com

A SAS Institute Inc. Quality Partner

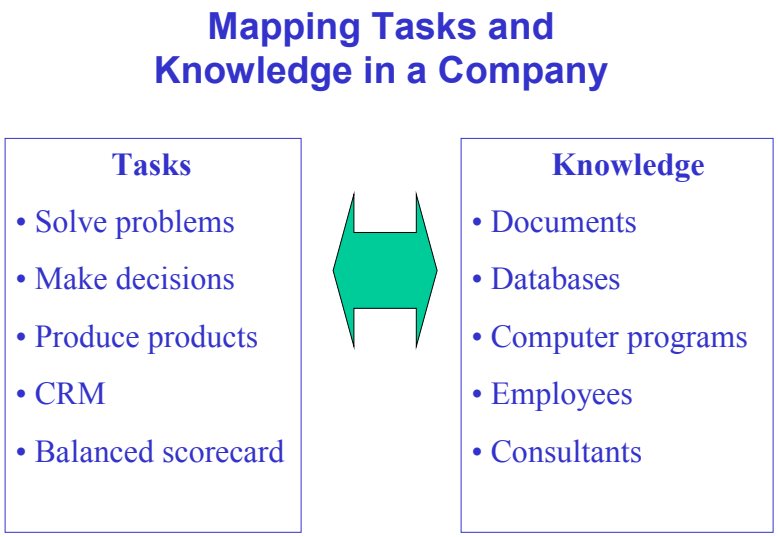


Figure 1: Knowledge used to solve enterprise tasks is stored in many forms.

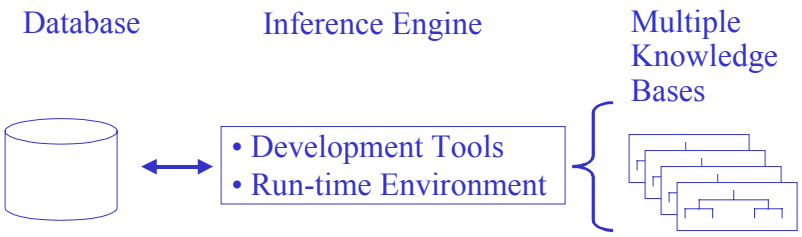


Figure 2: The intelligent system is composed of an inference engine, multiple knowledge bases, and a database

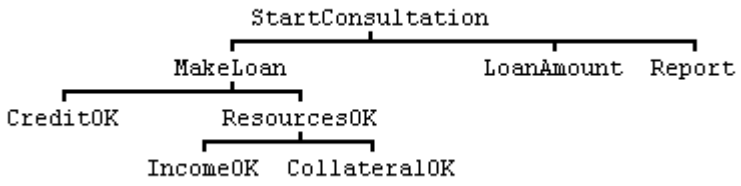


Figure 3: Tree for the MakeLoan knowledge base.

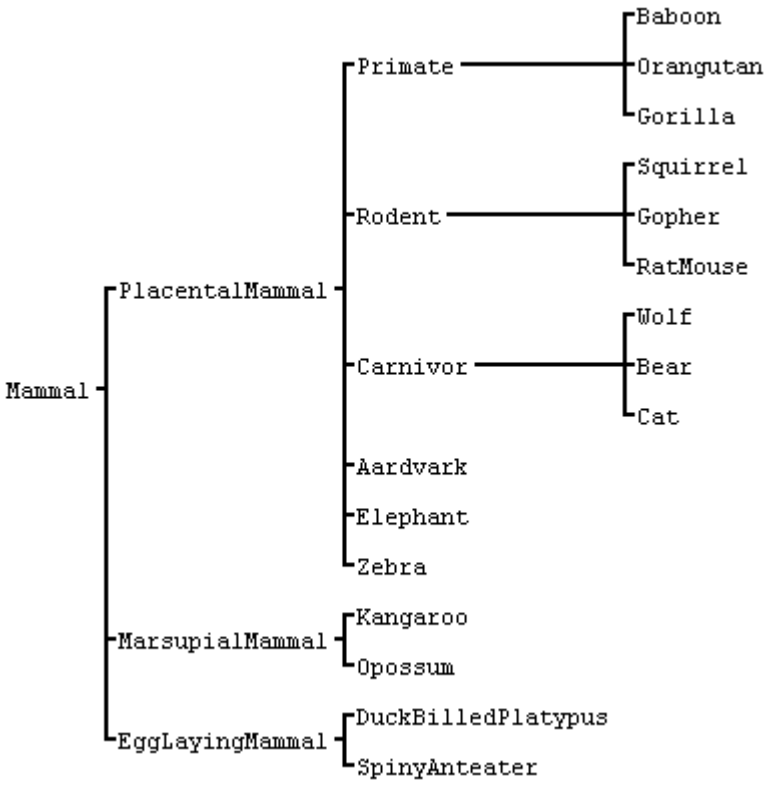


Figure 4: Tree for the Mammal knowledge base. (The tree is shown sideways with the root at the left.)

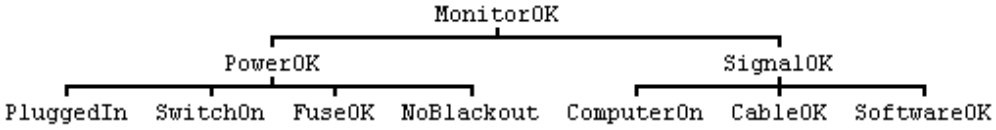


Figure 5: Tree for a service call center supporting monitors.

SAS and Electronic Mail:
Send e-mail faster, and DEFINITELY more efficiently

Roy Fleischer, Sodexho Marriott Services, Gaithersburg, MD

Abstract

With every new software package I install, I look for some application that will enable me to simplify or improve the efficiency of my day to day processes. After all, technology is supposed to make things better, more able, and somehow remove manual processes. This is exactly what happened when I was introduced to SAS' e-mail capabilities. I suddenly found I could, when completing a task, e-mail out the resulting file with the click of a button. I would not need to leave SAS to open e-mail, find the name of the person(s), attach the file(s) after searching for the correct subdirectory, then hit send. Instead, I was able to replace a lengthy administrative task with a single, repeatable one.

Introduction

Decision Support Systems (DSS), a division of Sodexho Marriott Services is responsible for making ***all data*** in the company useable. Because our use of SAS gives us the ability to parse static reports and read multiple formats (i.e., XLS, WK4, CSV, etc.), DSS is often called upon to convert some format to one of the user's preference. These preferences include paper reports, spreadsheets, text files, or converting data into SAS format to be used/merged with additional data.

One of DSS' primary objectives is to distribute financial data for each fiscal period (Sodexho Marriott is on a 12 period/5-4-4 calendar) to approximately 85 people across the company by 12:00 p.m. the Monday after financial close. These individuals are located throughout the United States and Canada, and their only common method of receiving information is via voice mail and e-mail. The people who receive the data are responsible for different areas of the company, and the files they receive include only those areas for which they are responsible. For example, the Vice President of Finance for Education Services receives all of the Education Services divisional information, and a Regional Controller receives ONLY data for his/her region.

The necessary tools required to accomplish this task remain the same for either a manual or systematic approach. The differences reside in the versions of SAS and Lotus cc:Mail that are used. Today's solution includes SAS Version 8 and Lotus cc:Mail Version 8.2. Previous versions of Lotus cc:Mail are not MAPI compliant. Microsoft Excel 97 is also used to maintain the specifics of the distribution list. SAS can read versions of Excel back to Version 4. The last requirement is an able body to submit the program.

Lotus cc:Mail Database Preparation

Lotus cc:Mail has a proprietary database format that SAS cannot read. To make the database readable, Lotus provides a utility. Figure 1 illustrates the cc:Mail utility to create the necessary file.

Figure 1:

wmcheck localpo password database location dirinfo >>path\ccmobile.csv

wmcheck	-	Lotus cc:Mail utility
localpo	-	keyword
password	-	users password
database location	-	local post office location
dirinfo	-	keyword
>>path\ccmobile.csv		
	-	path and name of export file

Mailing List Preparation

DSS maintains a list of recipients and specifics about the data they receive in an Excel workbook named **fastdist.xls**. In this workbook, multiple worksheets are defined for each division, change history, current period changes, divisional totals, and a comprehensive list (Tab: LIST) to import into the process. This list is updated every period with any changes and then saved to the designated place on the local area network (LAN).

Attachment Preparation

Every period, a process runs to create separate files relating to different areas of responsibility. These files are zipped, named FASTUPD.EXE, and attached to the resulting emails. The FASTUPD.EXE files range in size from <1 meg, for a region at period 1, to close to ~20

meg, for a full division at period 12 (year end). They are located on a dedicated machine within subdirectories designated by division (EUP=Education Services), period number, reporting hierarchy (S=SVP, D=DVP, R=RVP) and area of responsibility (051=Education Southeast). Figure 2 is an example of the directory structure after the process has been completed.

Figure 2:

c:\fsm$chgs\eup\pd10\d051\fastupd.exe

c:	-	Local drive
fsm$chgs	-	subdirectory
eup	-	division
pd10	-	period number (1-12)
d051	-	DVP 051
fastupd.exe	-	attachment

SAS Configuration

Figure 3 shows the lines that need to be added to the bottom of the SAS configuration file, SASV8.CFG.

Figure 3:

-EMAILSYS MAPI
-EMAILID "id"
-EMAILPW "password"

MAPI	-	type of email
id	-	MAPI compliant mail id
password	-	MAPI compliant mail pw

SAS Program

The shaded boxes below illustrate the complete SAS program used to import the two necessary databases, merge the databases together, ensure the attachments and their respective directory structure were created, and finally mail the complete e-mail to the users.

---Period Assignment---

Assign period number to macro PD.

```
%let pd = 10;
```

---Lotus cc:Mail---

Read the complete name from the Lotus cc:Mail database comma separated file created earlier in Figure 1. Delete blank records. Create a new variable, CCNAME, making all characters capital and left justified. Now this file is almost ready to merge with the distribution list. Make sure the attributes are the same in the files that will be merged.

```
data work.ccMail;
    keep CCNAME NAME;
    attrib          NAME            informat=$77.
                                    length=$77.
                                    format=$77.
                    CCNAME          length=$40.
                                    format=$40.
    ;

    infile 'path\ccmobile.csv' delimeter=','
          missover dsd firstobs=8;

    input NAME $;

    if NAME = " " then delete;
    CCNAME = upcase(trim(left(name)));
run;
```

The Lotus cc:Mail database created earlier in Figure 1 might have duplicate records. Sort data by CCNAME and use NODUPLICATES option to remove like records.

```
proc sort data=work.ccMail noduplicates;
     by ccname;
run;
```

---DSS Distribution---

Import the distribution list maintained in Excel. Using the option RANGE, read only the single tab needed.

```
proc import out=work.distr
            datafile="path\fastdist.xls"
     dbms=EXCEL2000 replace;
     range="List$";
     getnames=YES;
run;
```

Add the variable CCNAME to newly created WORK.DISTR dataset with the same attributes that were defined in WORK.CCMAIL.

```
data work.sasname;
     length CCNAME $40.;
     set work.distr;

     if NAME = ' ' then delete;
     CCNAME=upcase(trim(left(name)));
run;
```

Sort the dataset just created by CCNAME to assure data are in the same order as WORK.CCMAIL.

```
proc sort data=work.sasname;
     by ccname;
run;
```

---Exception Reports---

Merge the two datasets, SASNAME and CCMAIL, to check for exceptions. This ensures all names on the distribution list are actually on the e-mail database. If names don't exist in both extracts they fall into WORK.OOPSNAME.

```
data      work.checkname
          work.oopsname;

     drop CCNAME;

     merge          work.sasname (in=sas)
                    work.ccMail (in=ccMail);
     by CCNAME;

     if SAS and CCMAIL then output
          work.checkname;
     else if SAS then output work.oopsname;
run;
```

Print WORK.OOPSNAME in an exception report if any observations from the distribution file were not in the e-mail database. If any names are found on this report, check the distribution file for typos, then check with the individual for their correct email.

```
title1 '***ALERT***';
title2 'NAME NOT IN CC:MAIL';

proc print data=work.oopsname;
run;
```

Read the resulting dataset, CHECKNAME. Create the directory structure for the attachments. If the directory doesn't exist, it falls into WORK.OOPSFILE. If a directory wasn't created for a specific area, it will fall into WORK.OOPSFILE.

```
data      work.ready
          work.oopsfile;

     set work.checkname;

     fastupd =      (compress('c:\fsm$chgs'||pl||'\pd'||&pd
                    ||'\'||level||'fastupd.exe'));
     rc=fileexist(FASTUPD);

     if rc then output work.ready;
     else output work.oopsfile;
run;
-------------------------------------------------------
Note:     (variables found in dataset)
pl        -        division (EUP)
level     -        reporting level and are (D051)
```

Print WORK.OOPSFILE in an exception report if the correct directories weren't created. If any directories are found on this report, check the distribution file for typos, then check with the individual for his/her correct area of responsibility.

```
title1 '***ALERT***';
title2 'FASTUPD NOT ON LAN';

proc print data=work.oopsfile n noobs;
     var name fastupd;
run;
```

After the previous steps have run without any exceptions, output, the mail portion of the program can be executed. If exceptions occur and reports are produced in the mail section, the system will mail all records before the error and abort at the problem point.

---Initializing Mail---

Create a file reference for e-mail using a FILENAME statement. The EMAIL option is used as a keyword indicating the use of electronic mail.

```
filename reports email "FSMFAST";
```

---Mail Dataset---

Read the dataset, WORK.READY, created in the previous step. Create WORK.MAIL to produce a complete list that was mailed.

```
data work.mail;

      set work.ready;

      file reports;
```

---Mail Attributes---

SAS' mail functionality requires certain attributes to be assigned similar to those of any electronic mail package. The following shows the attributes that are assigned both through dataset variables and the body which is typed in the way it would show in the e-mail. The SAS log will contain all the e-mails sent.

```
      put  '!EM_TO!' name;

      put '!EM_SUBJECT! FSMFAST Period Updates';

      /***** BEGIN TEXT IN BODY OF E-MAIL*****/
      put   'BUSINESS AS USUAL / WHAT TO DO';
      put / 'Reminder:';
      put / '    Users on LAN:  Boot your machine as a
remote pc, NOT hooked to the LAN.';
      put / '    (Some users have had problems, some have
NOT – just a sageguard.)';
      put / '    Users with CD Rom Drives Attached:  Have
a readable CD in the drive or';
      put / '    unhook the CD drive before booting.';
      put / ' ';
      put / 'Processing Your Data:';
      put / '    1) Attached is your FASTUPD.EXE for
your Period updates.  Copy this';
      put / '       file to your FSM$FAST directory and
replace any previous versions.';
      put / ' ';
      put / etc...
    /***** END TEXT IN BODY OF E-MAIL*****/

    put '!EM_ATTACH!'  fastupd;

    put '!EM_SEND!';

    put '!EM_NEWMSG'!;

    put '!EM_ABORT!';

run;
------------------------------------------------------------
Note:     (variables found in dataset)
name     -          name off distribution list
fastupd  -          location and attachment
```

---Mail Report---

After completing the datastep and processing all the e-mails, WORK.MAIL is ready to produce a report containing a complete distribution list.

```
Title 1 'MAIL LIST';
Title 2;

proc print data=work.mail;
run;
```

Limitations

This version of SAS e-mail doesn't have the ability to do the following:

- change the priority from NORMAL to URGENT
- request RETURN RECEIPT.

Conclusion

SAS has created the ability to actually remove an lengthy, repeatable administrative function. With SAS' assistance modifying the SAS config file, this program was extremely easy to write and implement. The most difficult part of this is optimizing e-mail. With only two limitations, that I found, this functionality should be evaluated for any mass electronic mailing.

Acknowledgements

Sodexho Marriott Services is the largest provider of food and facilities management in North America, with $4.5 billion in annual sales. Sodexho Marriott offers a variety of innovative outsourcing solutions, including food service, housekeeping, groundskeeping, plant operations and maintenance, asset and materials management, and laundry services to corporations, health care facilities, schools, universities and colleges, and remote sites. Headquartered in Gaithersburg, MD, the company has 103,000 employees at 5,000 locations across the U.S. and Canada.

This solution was made possible by the challenges that Decision Support Systems faces every day and our need to answer and automate, if possible, as quickly as possible.

Contact Information

Roy Fleischer
Director, Decision Support Systems

Sodexho Marriott Services
9801 Washingtonion Blvd, Ste 1215
Gaithersburg, MD 20878

Phone: (301) 987-4309
Fax: (301) 987-4339
Email: rfleischer@sodexhomarriott.com

Paper P156-SSU2001

Advantages and Disadvantages of Using MDDBs, HOLAP, and SAS/IntrNet® in the Development of an Interactive System

Richard A. Denby, Lori A. Guido, United States Census Bureau, Washington, DC

ABSTRACT

The Housing and Household Economic Statistics Division (HHES) of the United States (U.S.) Census Bureau evaluated different SAS technologies to find ones best suited for the development of an interactive system to review the Census 2000 long form data. The three systems that were evaluated access data stored on centralized, large-scale Unix servers and the data are accessed through personal computers (PCs) running Windows 95. All the systems use multidimensional databases (MDDBs), and display the same data, but each system uses distinct SAS technologies.

- The first system uses a client-server approach, SAS v6.12 and the MDDB report object in SAS/EIS® to create more than 3400 commonly used reports.
- The second system uses a client-server approach, SAS v8.1's Hybrid On-Line Analytical Processing (HOLAP) techniques to build a "proxy" HOLAP cube in addition to the MDDB report object in SAS/EIS®, but only 74 reports had to be created.
- The third system uses a web-based approach, SAS v8.1's HOLAP and SAS/IntrNet® and other web products to display the data.

This paper describes the advantages and disadvantages of each technology used, including development, deployment, and performance issues, in addition to the space savings obtained via the use of formats.

INTRODUCTION

The U.S. Census Bureau is best known for conducting a national census in years ending in a zero. By December 31 of a census year, the Census Bureau must provide the U. S. President with population totals for each of the 50 states and the District of Columbia. This information determines the number of seats to which each state is entitled in the House of Representatives, which is fixed at 435 seats, with each state getting at least one representative. In this way, seats in the House of Representatives are "apportioned" among the states.

Census data are also used to delineate congressional and other election districts within each state. This processing is called "redistricting." States typically have tight deadlines for completing their redistricting work in time for the 2001 or 2002 primaries and elections. The Census Bureau is required to provide redistricting data to the states, at a very detailed level of geography, within one year of the census. Producing these data is a massive undertaking that involves tabulating the characteristics of more than 280 million people in about 116 million housing units assigned to 39,000 governmental entities in 7.5 million census blocks.

The apportionment and redistricting data are derived from the total universe of all people. Additionally, approximately one in six households received a "long form" questionnaire containing 53 questions covering 34 subjects. The remaining five in six households get a shorter form with only seven questions. Every question asked in Census 2000 was one that collected information required by law to manage or evaluate federal programs or that was needed to meet legal requirements stemming from U.S. court decisions such as the Voting Rights Act. Federal dollars for schools, employment services, housing assistance, highway construction, hospital services, programs for the elderly, and other programs are distributed based on census data. The Census 2000 Long Form Data Review system, which will be used to review the "long form" questionnaire data, is scheduled to be in production in the fall of 2001.

SELECTION CRITERIA

HHES used the following criteria to decide which technology would be best suited for the Census 2000 Long Form Data Review system.

- The resulting system must be easy to maintain and enhance and available for scheduled use by fall 2001. HHES will have to develop a review system in a very short amount of time. The analysts will have very little time to review the data, so the programmers will have to be able to fix any problem with the application in a short amount of time.
- The technology must handle data files containing more than five million observations.
- It has to allow for programming flexibility. The specifications for the system might change even after the system is in production.
- New data will have to be incorporated into it on a "flow" basis easily, without affecting the data already in the system.
- The technology must allow users to view several groupings of data at once.
- The systems developed with this new technology will have to be easy to deploy to several different user communities, each of whom have different PC environments.
- Of course, the data have to be displayed to the user as quickly as possible.

THE SAS V6.12 PROTOTYPE

The first prototype HHES developed uses a client-server approach, SAS 6.12 under Windows95 and resides on user PC desktops. Each PC has to have SAS 6.12, and the application's configuration and profile files installed on it. The data reside on Sun Unix servers running the Solaris operating system. The MDDBs and the underlying detail data sets on the Unix servers are accessed using "Remote library services" via SAS/CONNECT®. There is one detail data set, one MDDB, one set of SAS/EIS® reports and one set of application files for each of the fifty states, the District of Columbia, and Puerto Rico. One metabase file holds all of the metadata for all of the states. The application files and the metabase file reside on a Novell server.

The data are displayed via a gif file that contains a map that links each state's hot spot to an icon that lists the reports for each state. This configuration only allows access to one state's data at a time. For this system, more than 3400 reports would have to

be created, with one set of 74 reports having to be created for each state. In other words, there would have to be 51 copies of each report, since each report could access only one MDDB, or one state's data. If a problem with a label or title were found in a report, all the copies of that report would have to be updated. There is no way to mass-correct the reports. Also, the risk of making mistakes increases when developing such a large number of reports.

It takes seven seconds to display a summary report. It takes two seconds to display 557,000 detailed observations from a data set containing 29,700,000 observations. Most of the objects used for this system are "off the shelf." No override methods are used. Minimal knowledge of Screen Control Language (SCL) was needed for the creation of this system.

HHES identified a substantial limitation in the use of SAS formats with version 6.12. Analysts want to see display formats that are more meaningful than the underlying coded values. For example, the report displays the words "male" and "female" in place of the coded values of "1" and "2". The problem occurs when analysts use the "show detail" option from the EIS multi-dimensional report object. No detail records making up the cell total are found because during the "reach-through" process, internally SAS was searching for "male," when in reality all the data values for males were stored as "1."

So, for reports where data ranges or labels have to be displayed, a separate field was created and the appropriate range was stored in it. Using another example, the "Drilldown 3 Highest Degree" report, shown in Figure 1, displays the number of people who have completed certain levels of schooling (EHIGH) by the edit parameters that were used to allocate the data if the response was blank or inconsistent (FLHIGH).

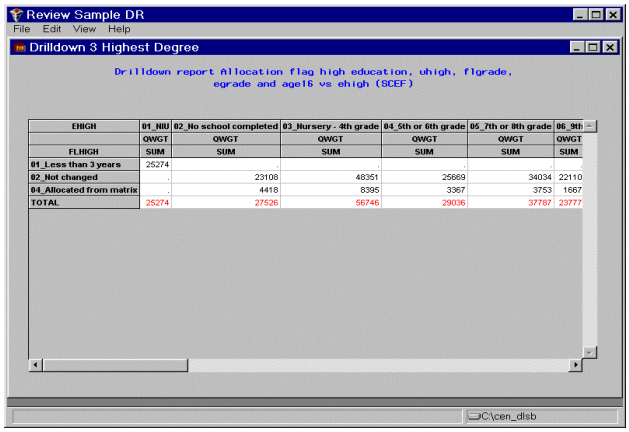


Figure 1. Drilldown 3 Highest Degree Report for the SAS v6.12 prototype. Please note, the data shown are from the Census Dress Rehearsal and do not represent actual Census 2000 data.

For each observation, a field called QHIGH contained a character code that stood for a certain level of education obtained. Data processing that takes place before the MDDBs are built created another field called EHIGH, which contained the character string corresponding to the character code. Figure 2 shows a partial listing of the QHIGH and EHIGH fields from the detail data set that goes into this report.

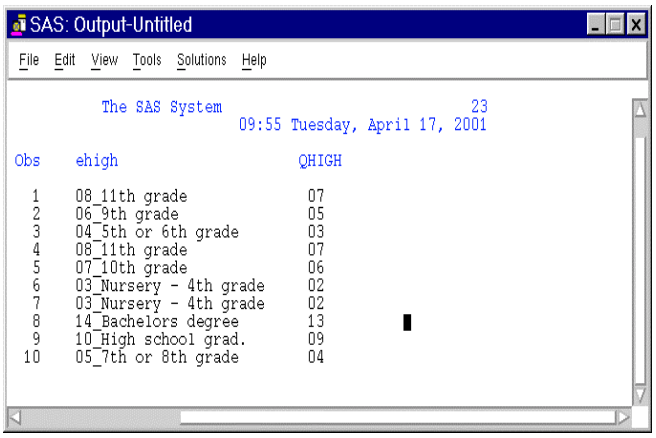


Figure 2. Detail observations used to create the Drilldown 3 Highest Degree Report for the SAS v6.12 prototype. Please note, the data shown are from the Census Dress Rehearsal and does not represent actual Census 2000 data.

THE HOLAP PROTOTYPE

HHES worked with SAS Consulting Services to develop the next two prototypes. The HOLAP prototype took advantage of new features found in SAS v8.1, but also retained all of the functionality contained in the SAS v6.12 prototype. The prototype's main feature is a new SAS technology called hybrid online analytical processing (HOLAP). This technology allows users to access data from multiple local and remote MDDBs and data sets as if the data were coming from a single data source.

A SAS view is created that allows concurrent access to the detail data sets. The view must be rerun each time a new state's data is added. This view is the input to a "template" MDDB. The template MDDB will store the hierarchies, formats, and base table attributes to pass to the resulting HOLAP cube. It does not have to be rerun every time a new state is added. It will need to be rerun only when the hierarchies, formats, or base tables change.

The HOLAP cube is stored in the central repository. It holds the location of each component of the logical data group and what it contains. It has to be updated each time a new state is added.

The use of the HOLAP technology decreases the number of reports that will have to be created, from the more than 3400 required in the SAS v6.12 prototype, to a single set of 74 reports that can be used for any state. The reports can be created in advance, decreasing the amount of time between file processing and file availability to the users. The HOLAP technology also allows users of the system to view data for multiple states and multiple years in one report.

The data are displayed via a modified version of the gif file used in the SAS v6.12 prototype. The file still contains the map, but for this prototype, there are hot spots for each state, regional groupings of states, and the entire U.S. The desktop frame class is overridden so new SCL code that allows multiple states to be selected will execute. It takes 12 seconds to display a summary report. Figure 3 shows how the "Drilldown 3 Highest Degree" report appears in the HOLAP prototype. The report still displays the number of people who have completed certain levels of schooling, but now the report shows data for multiple states. From this report, the data for any of the listed states can be expanded. Figure 4 shows the same report, but with data from both states expanded.

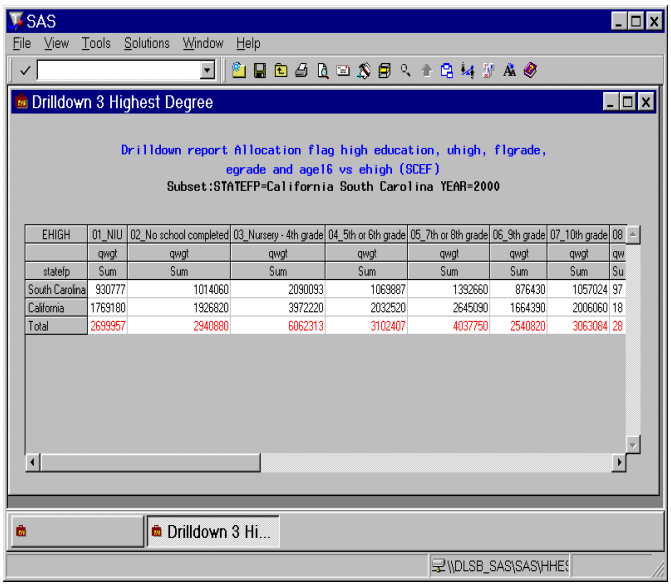


Figure 3. Drilldown 3 Highest Degree Report for the HOLAP prototype. Please note, the data shown are test data created specifically to be shown to non-Census employees and does not represent actual Census 2000 data.

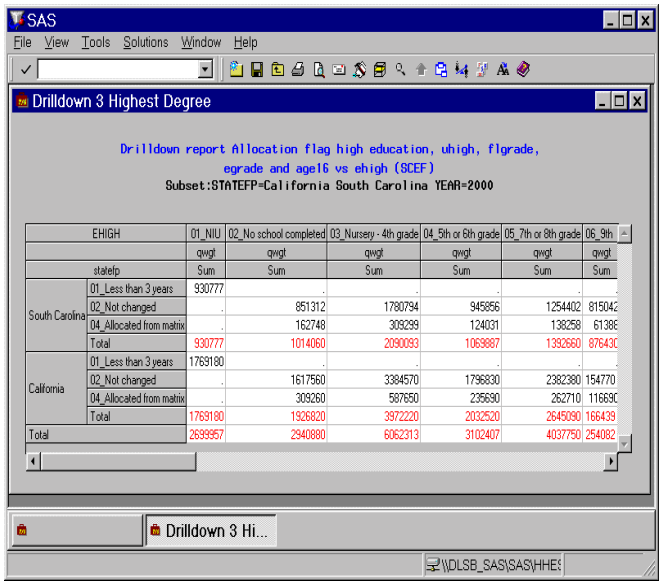


Figure 4. Drilldown 3 Highest Degree Report from the HOLAP prototype with state data expanded. Please note, the data shown are test data. It was created specifically to be shown to non-Census employees and does not represent actual Census 2000 results.

"Remote library services" are also used in this system to reach the MDDBs and the underlying detail data sets on the Unix servers. There is still one detail data set and one MDDB for each state. The central repository contains all the reports and the metabase registrations. However, the repository has to reside on each user's PC, which must have SAS 8.1, and the application's configuration and profile files installed on it.

Another override method is used to correct a potential performance problem that HHES discovered. While in the "Wage/Salary Income" report, when only one state was chosen, a "show detail data" took 40 minutes to subset 269,104 observations from the 18,366,027 observations available in a test data set. SAS believes this performance problem is due to the use of the HOLAP cube, which uses a view to point to all of the detail datasets.

SAS does not use the indexes on the data sets when accessing the data via a view. Even if only one state was selected, SAS is reading all the detail data sets in the data group sequentially to pull out the selected data. The prototype now uses an override method that captures the cell the cursor is sitting on when a "show detail data" is invoked. The SCL code determines what the cell represents, bypasses the HOLAP cube, goes directly to the data set that contains the detail data for the cell and subsets the data based on the cell's contents. This method allows the same 269,104 observations to be displayed in less than five seconds. A more in-depth knowledge of SCL is needed to maintain or update this prototype due to the use of these override methods.

The prototype also incorporated the use of reach-through of display formats to data set values so they did not have to be re-coded. This decreased disk space storage requirements with savings ranging from 32% to 63%. For example, the original education data set for Texas, which was created in SAS version 6.12 and had re-coded values stored in the data set, was 9 gigabytes. After the data set was converted to version 8.1 and the formats were applied, the file was 3.4 gigabytes, producing a 61% savings in disk space.

THE WEB-BASED PROTOTYPE

The third prototype takes advantage of new features used in the HOLAP prototype, retains most of the functionality contained in the SAS v6.12 prototype, but also takes advantage of the features of SAS/IntrNet®. This was the first attempt at the U.S. Census Bureau to build a web-enabled HOLAP system incorporating "reach-through" to the underlying detail data sets.

The Web-based prototype differs from the HOLAP prototype in how the data are displayed. The data are displayed via an HTML file that contains a map that has hot spots for each state, regional groupings of states, and the entire U.S. Javascript captures the mouse clicks and places the selections in the list box. A macro program takes the list of selected states and manipulates code to access the HOLAP cube that points to the selected data. The MDDB Report Viewer (MRV) that comes with SAS/IntrNet, is used to display the data in a report show, in figure 5.

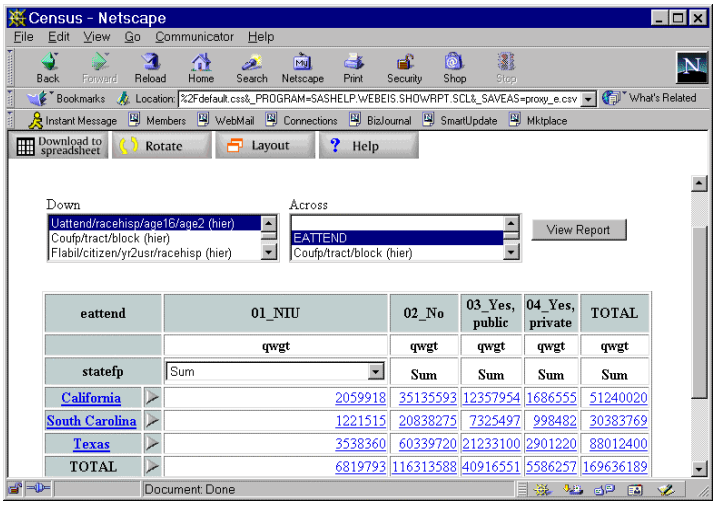


Figure 5. A Summary of education data from the MRV portion of the Web-based Prototype. Please note, the data shown is test data. It was created specifically to be shown to non-Census employees and does not represent actual Census 2000 data.

Expanding the reports is time consuming. Response time varies with the size of files. For the MRV portion of the prototype, it takes seven seconds to display a summary report. It takes 80 seconds to display the first set of 50 detailed observations.

Sun's Web server is used with SAS/IntrNet® to access the MDDBs and the underlying detail data sets on the Unix servers. There is still one detail data set and one MDDB for each state. The HOLAP technology is still used and all the reports and the metabase registrations are still contained in a central repository. However, SAS v8.1, the application's config file, the application's profile file, and the repository reside on the Unix server. Nothing resides on the user's PC.

Setting up the SAS/IntrNet® server is very tedious. The auto installer did not work for this product, so the installation had to be done manually. The instructions were lengthy and written in pieces. HHES's system administrator could not get an overall picture of all the steps necessary to complete the task. Many tasks have to be performed to complete the installation. The slightest deviation from the procedure can cause the installation to fail. Knowledge of Unix system administration, web server software (Apache or Sun web server), and general installation procedures are needed to do this task. Extensive knowledge of Javascript will be required to maintain or expand this prototype.

FINDINGS

HHES compared the prototypes with each other, reviewing the benefits and disadvantages of each. HHES also evaluated the prototypes against the criteria to be used to decide which technology would be suited for the Census 2000 Long Form Data Review system. HHES could find no clear-cut winner based on this analysis. No prototype met all of HHES's criteria. All three prototypes could handle data files containing more than five million observations in them. All three prototypes can incorporate new data "flow" basis easily, without impacting the data that were already there. Only the prototypes that use the HOLAP technology allow users to view several groupings of data at once. The SAS v6.12 prototype can display the data faster then either of the SAS v8.1 prototypes. Table 1 shows various statistics HHES used when evaluating the prototypes.

	SAS 6.12 Prototype	**HOLAP Prototype**	**Web-based Prototype**
# of states that may be displayed at once	1	50	50
Time to display a summary report	7	12	5
# of obs / # of seconds to display them	557,000 / 2	269,104 / 5	89,326 / 76
Can handle new data on a "flow" basis	Yes	Yes	Yes

Table 1. Miscellaneous Statistics Used for Prototype Evaluation.

The SAS v6.12 prototype was the least flexible because of the number of reports that would have to be created. The HOLAP prototype and the Web-based prototype offer more flexibility. Changes to the report labels and titles that would affect all the states can be made quickly in the prototypes that use the HOLAP technology, since each only has one set of reports that will access any of the states.

The SAS v6.12 prototype was the easiest to maintain, because it uses "off the shelf" SAS objects. The objects are well documented and the HHES programmers have a lot of experience using SAS v6.12.

The HOLAP prototype is the second hardest application to maintain, due to the use of the override methods and SCL code. Except for the override methods, this prototype is very similar to the SAS v6.12 prototype. The HOLAP creation process was very easy to understand, maintain and modify. There are objects in SAS v8.1 that would allow a user to choose which state to view. The use of this object would eliminate the need for the map and the need to override the desktop frame class. However, an override method had to be used to meet HHES's need for fast response time to display detail data. Organizations with more override method and SCL programming experience would be able to maintain the prototype with ease.

For HHES, the Web-based prototype is the most complicated and the hardest prototype to enhance or maintain. This assessment is based on HHES's lack of experience with Javascript, and the problems with the server installation. Organizations with more Javascript and SAS/IntrNet® server installation experience might be able to maintain the prototype with ease.

The Web-based prototype is the easiest application to deploy. Only a web-browser is needed on the user's PC. Both the SAS v6.12 and the HOLAP prototypes require that the appropriate version of SAS be installed on the user's PC. With a very slight modification, the prototypes could probably use a network version of SAS, but we have not tested this yet. The configuration and profile files needed by the SAS v6.12 prototype, can be copied to the user's PC via a network release package. HHES is not sure if the central repository in the HOLAP prototype can be deployed to the user's PC via a network release package. Another possible option for the HOLAP prototype deployment would be to have the central repository reside on a network server, but HHES has not yet tested the ramifications of this configuration.

CONCLUSION

HHES will have to conduct more research before drawing a final conclusion. HHES did not try tuning anything, nor did they fully load test the prototypes. HHES is leaning toward using the technology used in the HOLAP prototype for five reasons.

- There are fewer reports to create.
- SAS v8.1 is closer to the most current SAS version than SAS v6.12.
- The use of the HOLAP technology will allow the users to view groups of states instead of just individual states.
- The potential problems we might encounter with the override methods and SCL code in the HOLAP prototype will probably be overcome with additional training and internal mentoring from more experienced programmers. Unfortunately, the HHES programmers have the most SAS/IntrNet and Javascript experience of any programmers at the Census Bureau, so mentoring is not an option if we use the Web-based prototype technology.
- The benefits of using the HOLAP prototype technology outweigh the SAS v6.12 prototype's performance advantage.

Currently we are creating a SAS v8.1 HOLAP system using 1990 Census data to fully load test the technology before we go into production with Census 2000 data. It will allow us to try different tuning strategies. If these tuning strategies work on 1990 Census data, they should work on Census 2000 data. It will also allow us to research and test the other outstanding questions that we had with the prototype, while allowing the programmers to gain more override and SCL experience.

REFERENCES

CONTACT INFORMATION

Richard A. Denby
U.S. Census Bureau
4700 Silver Hill Road, 8500-3
Washington, DC 20233-1912
phone 301-457-6810 fax 301-457-3248
email richard.a.denby@census.gov

Lori A. Guido
U.S. Census Bureau
4700 Silver Hill Road, 8500-3
Washington, DC 20233-1912
phone 301-457-3204 fax 301-457-3499
email lori.a.guido@census.gov

Integrating SAS/Connect® with Java™

Randy Curnutt, Solutions Plus, Inc., Indianapolis, IN
Michael Pell, Solutions Plus, Inc., Indianapolis, IN
John LaBore, Eli Lilly And Company, Indianapolis, IN

ABSTRACT

SAS/Connect provides an easy and reliable method for connecting remote hosts and executing SAS® commands on the remote environment. This paper explores the use of SAS/Connect in conjunction with the robust Java language. Various methods of integration with SAS/Connect will be reviewed, including dynamic program generation to create and run a web-based application, and the dynamic generation and execution of SAS/Connect programs to be submitted and executed on remote systems, such as OS/390 mainframes, Unix servers, and NT servers. The following will discuss the advantages and disadvantages of each approach.

INTRODUCTION

THE PROBLEM

In today's IT environment many different hardware and software platforms are often thrown together in an attempt to meet diverse business needs. This hodgepodge approach forces the user to learn and work with each different system in order to perform the steps necessary to obtain the needed data or programs. Often users spend more time learning an operating system and/or protocol than doing the job they were hired to do. For example, if the specific operating system commands needed to logon and traverse a directory structure are very different, this significantly increases user knowledge/time requirements. However, even subtle differences between platforms can have an impact. For example, floating-point numbers are stored differently on various operating systems. This may result in discrepancies in the significant digits in high precision decimal numbers that may greatly discomfort a systems analyst (Klenz, 1992). Conversely, a statistician who understands the net effect of this type of difference may not perceive it as a problem.

A SOLUTION

One solution to the above problem is to develop an application that uses SAS, SAS/Connect and Java to provide functionality to the users so that they can accomplish their desired tasks without learning the different nuances of multiple hardware and software environments.

All of these multi-platform tasks can be performed by SAS/Connect itself; however anyone who chooses to do such a task must be fairly knowledgeable of the entire hardware and software environments in which they are working. This includes technology differences between the platforms being used, networking and communications issues between the platforms, operating systems, protocols, multiple languages, etc. Solving all of these issues at any one time can be difficult, even for an experienced SAS user. Optimally, the SAS user should not be sidetracked from their primary job with all of these diverse issues. Some of the tasks that are commonly performed on a routine basis yet may be difficult to complete include:

- Moving data to other platforms
- Converting data to other formats
- Analyzing and reporting
- Data manipulation
- Input/output
- Consistency, validation, repeatability
- Formally publishing results

The driving force for creating the suggested application is that it provides a custom solution for all users and prevents them from reinventing the wheel. Furthermore, with this approach the solution is built and validated once, instead of requiring retesting every time it is used.

THE APPROACH

The focus of this technique is to optimally integrate multiple hardware platforms using SAS/Connect with Java so that maximum convenience is provided to the end user. The application itself is developed in Java. Java also provides a user interface that can be either standalone or web-based. SAS programs and macros are embedded within the Java code, so that Java acts as a program generator for SAS programs and macros.

PLATFORM INDEPENDENCE

Java, SAS programs and SAS macros are highly standardized and portable. SAS programs, macros and datasets are not binary compatible across platforms but the programs themselves are highly compatible in terms of commands and syntax. There are minor distinctions in syntax for OS/390 SAS versus PC SAS, but for the most part the majority of programs can be used across platforms with only minor modifications.

STATIC OR DYNAMIC PROGRAMS

Combining Java and SAS provides an architecture capable of generating either static, predefined SAS programs, or dynamically generated custom programs based upon selections a user has made. To understand the static and dynamic generation of SAS programs, let's examine how this can be accomplished with Java. Java programs contain classes, a term describing a combination of attributes and methods that may be executed. One approach to generating SAS programs or macros is to have a Java class that contains methods, similar to functions in other languages, dedicated to concatenating a number of Strings together to build the desired program. Specifically, the method actually uses StringBuffer objects, since this is a far more efficient manner for concatenation within the Java programming language. Here's an example of appending two StringBuffers together.

```
StringBuffer sb1 = new StringBuffer(
  "Energizing Users with a ");
StringBuffer sb2 = new StringBuffer(
  "Slice of SAS and a cup of Java");
StringBuffer sb3 = new StringBuffer();
sb3.append(sb1);
sb3.append(sb2);

// will return "Energizing Users with a Slice
of SAS and a Cup of Java"
```

```
sb3.toString();
```

This approach can append together any number of predefined StringBuffer objects to build one long String that contains a whole SAS program. Here's an example of building a static SAS program which is run on the OS/390, and returns the output of a PROC CONTENTS in a text file:

```
StringBuffer sasPgm = new StringBuffer();
sasPgm.append("//USERID JOB (,8305,S),
   'PROC_CONTENTS',MSGCLASS=T\r\n");
sasPgm.append("//*\r\n");
sasPgm.append("//SAS     EXEC
SAS,SOUT=*,OPTIONS='LS=80 MPRINT SGEN'\r\n");
sasPgm.append("//SYSOUT    DD SYSOUT=*\r\n");
sasPgm.append("//SYSIN     DD *\r\n");

sasPgm.append("%MACRO CONTENTS(
   PRINTTO=,) ;\r\n");
sasPgm.append("%LET LIB=%UPCASE(&LIB);\r\n");
sasPgm.append("LIBNAME LNAME \"ABC_LIB\"
   DISP=SHR;\r\n");
sasPgm.append("FILENAME PRT \"&PRINTTO\"
    DISP=(OLD,DELETE);\r\n");
sasPgm.append("RUN;\r\n");

sasPgm.append("FILENAME PRT \"&PRINTTO\"
    DISP=(,CATLG) ");
sasPgm.append("SPACE=(CYL,(1,1))\r\n");
sasPgm.append("       RECFM=FB
   LRECL=80;\r\n");
sasPgm.append("RUN;\r\n");

sasPgm.append("PROC PRINTTO PRINT=PRT
   NEW;\r\n");
sasPgm.append("PROC CONTENTS
   DATA=LNAME._ALL_;\r\n");

sasPgm.append("%MEND CONTENTS;\r\n");
sasPgm.append("%CONTENTS(PRINTTO=
   USERID.TMP);\r\n");

sasPgm.append("/*\r\n");
sasPgm.append("//IFGOOD    IF (^ABEND &
   SAS.SAS.RC <= 4) THEN\r\n");
sasPgm.append("//RENAME    EXEC
   PGM=IDCAMS\r\n");
sasPgm.append("//SYSPRINT  DD
   SYSOUT=*\r\n");
sasPgm.append("//SYSIN     DD  *\r\n");
sasPgm.append("   ALTER USERID.TMP -\r\n");
sasPgm.append("   NEWNAME(USERID.TXT)\r\n");
sasPgm.append("//ELSE      ELSE\r\n");
sasPgm.append("//DELETE    EXEC
   PGM=IEFBR14\r\n");
sasPgm.append("//SASDEL    DD
   DSN=USERID.TMP\r\n");
sasPgm.append("//ENDIF     ENDIF\r\n");
```

The Java code above creates:

```
//USERID JOB (,8305,S),'PROC_CONTENTS',
   MSGCLASS=T
//*
//SAS     EXEC SAS,SOUT=*,OPTIONS='LS=80
   MPRINT SGEN'
//SYSOUT    DD SYSOUT=*
//SYSIN     DD *
%MACRO CONTENTS(PRINTTO=,) ;
%LET LIB=%UPCASE(&LIB);
LIBNAME LNAME "ABC_LIB" DISP=SHR;
FILENAME PRT "&PRINTTO" DISP=(OLD,DELETE);
RUN;
FILENAME PRT "&PRINTTO" DISP=(,CATLG)
   SPACE=(CYL,(1,1))
       RECFM=FB LRECL=80;
RUN;
PROC PRINTTO PRINT=PRT NEW;
PROC CONTENTS DATA=LNAME._ALL_;
%MEND CONTENTS;
%CONTENTS(PRINTTO= USERID.TMP);
/*
//IFGOOD IF (^ABEND & SAS.SAS.RC <= 4) THEN
//RENAME    EXEC PGM=IDCAMS
//SYSPRINT  DD  SYSOUT=*
//SYSIN     DD  *
   ALTER USERID.TMP -
        NEWNAME(USERID.TXT)
//ELSE      ELSE
//DELETE    EXEC PGM=IEFBR14
//SASDEL    DD DSN=USERID.TMP
//ENDIF     ENDIF
```

Dynamic generation is only slightly more complicated. It extends the previous approach by appending the value of variables between the StringBuffer objects. When the program is executed the method builds a single String that appends the StringBuffer value with the value of the given variable. For example, on a user interface a field is used to provide for the entry of a userId. In this situation presume that a value of "RandyC" was entered into the user interface as the value of userId. Following is a code snippet for doing dynamic generation:

```
public String buildSasMacro(String userId,
String password, String body)
{
  StringBuffer sasPgm = new StringBuffer(
   "%macro myMacro(userId=");
  sasPgm.append(", pswrd=);");

  sasPgm.append("\r\n\r\n");//2 returns
  sasPgm.append(body);
  sasPgm.append("\r\n\r\n");

  sasPgm.append("%mend myMacro;");
  sasPgm.append("%myMacro(userId=\"");
  sasPgm.append(userId);
  sasPgm.append("\",pswrd=\"");
  sasPgm.append(password);
  sasPgm.append("\");");

  sasPgm.toString();
}
```

The output of this method is:

```
%macro myMacro(userId=, pswrd=);

the body of the macro would be generated
here...

%mend myMacro;
%myMacro(userId="RandyC",pswrd="myPass");
```

The SAS code that is generated can easily be saved to a file. Doing this allows the code to later be edited by the user or it could be used to execute the program again without needing to regenerate the program.

BENEFITS

Not only can Java be used to generate the SAS scripts, it can also submit them to either a remote SAS server or to the local PC-SAS software.

Benefits of this approach include:

- A means to easily develop web-enabled platform independent applications
- A graphical user interface that provides an intuitive, easy to use interface for the user, shielding them from the behind-the-scenes details and protocols
- Enhanced security since the protocols and access methods are embedded within the application itself. The user can still be required to enter a password via the user interface but all of the secured functions are performed internally by the application, thus denying a would-be hacker the chance of altering the flow of the application

IMPLEMENTATION

LOCAL BATCH SUBMISSIONS

To submit a SAS program to PC SAS from Java, the SAS program is written to a temporary file on the local machine, then PC SAS is invoked using the temporary file as a parameter. This is equivalent to the following DOS command:

```
SAS.exe -NOSPLASH -ICON –SYSIN tempfile.sas
```

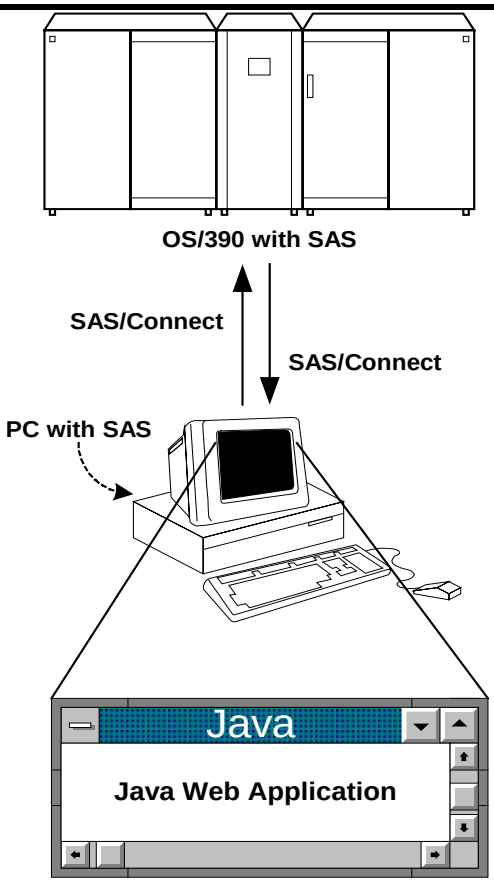
The appendix contains an example of how Java can submit the generated SAS program to PC SAS. See Example 1 for an overview.

There are a couple of issues to consider when using SAS/Connect. First, there must be an understanding of how PC-SAS connects to the remote server. It uses a sign on/sign off script file, such as tcptso.scr, to connect and sign on to the remote host. The networking environment and the desired remote host platform are the driving factors as to which script file should be used. The default scripts provided with the SAS software may need to be altered to accommodate the specifics of a remote host. One important change that is necessary when modifying the scripts is that there will be a need to develop a script which can be used for batch submissions by the application. The scripts come developed for an interactive session and are written to query the user for their user id and password. For the application to use such a script, it must be altered to pass in the user id and password and not query the user for these values (the querying is done by Java). Secondly, one must consider that a server might restrict a user to a limited number of sign-ons. This process will likely account for a sign-on, so it may be an important factor if the server's limit is one sign-on.

SAS/Connect provides the communication mechanism within a SAS framework to execute SAS commands on a remote host. SAS/Connect requires that the user login to the remote host. Once this occurs the user is free to write code to run on this remote host. This is done with the "`rsubmit`" and "`endrsubmit`" commands. When the user is ready to execute code on a remote host an "`rsubmit`" is issued in the SAS program. All SAS commands after the `rsubmit` are then run on the remote host until an "`endrsubmit`" command is issued.

For reporting errors or the SAS log back to the user via the Java code, the PC SAS program should use a PROC PRINTTO to output the errors or the SAS log to a temporary file:

```
PROC PRINTTO PRINT='C:\temp\saslog.log';
run;
```



Example 1. Completing a task on a remote platform via Java, PC SAS, and SAS/Connect

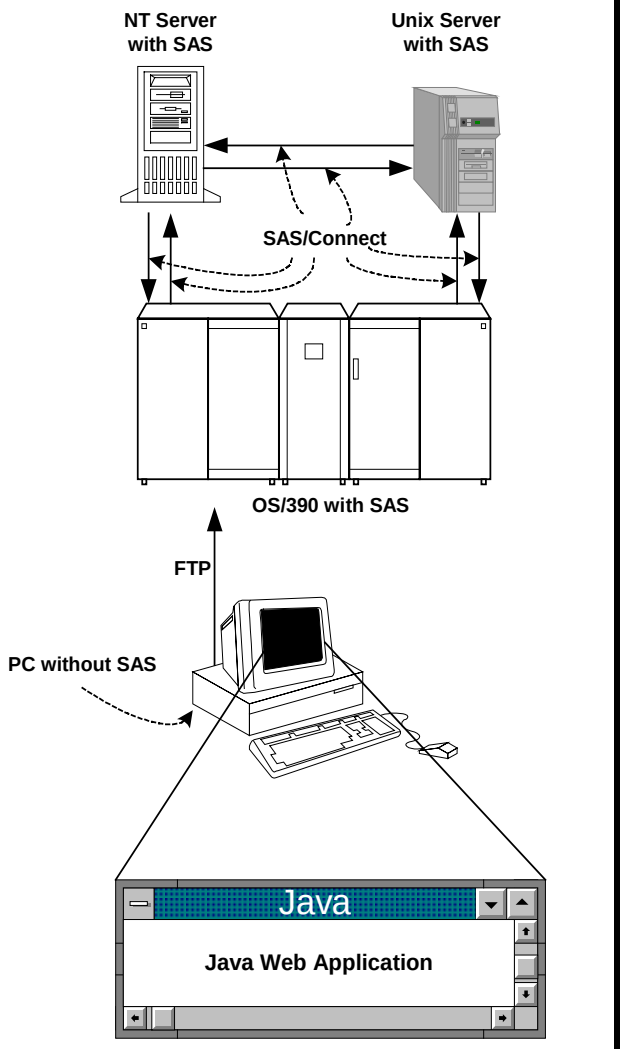
When the PC SAS program completes, the Java code will look for the temporary file, read its contents into memory, then delete the temporary file. It is important to remember that the output will not show up in the PC SAS Log window when developing and testing the SAS program with this PROC PRINTTO. It only shows up in the temporary file. Therefore the programmer may want to be selective about when that file is deleted. This will not be an issue once the SAS program is tested and validated.

REMOTE BATCH SUBMISSIONS

There are several approaches when submitting a SAS program to a remote host such as OS/390 or UNIX. This approach uses FTP to submit a SAS program via a JCL job. This can all be done in Java as well, however the process is a bit more involved than submitting a SAS program to PC SAS. (Java implementations of the FTP protocol are available for purchase or as freeware. For an example of developing an FTP class see "Hacking Java" by Mark Wutka).

The first step is to create a JCL program that runs a SAS job. The static SAS program shown in the 'Static or Dynamic

Programs' section above is a good example. See Example 2 for an overview.



Example 2. Completing a task on a remote platform Via Java and SAS using FTP and then using SAS/Connect to complete subsequent tasks on other remote platforms

The next step is to submit the JCL to the OS/390. In this application, a Java class is responsible for handling all FTP commands for communication to the remote server. Some of the commands are specific to the remote system's operating system, so it is important to design in this flexibility. The FTP Java class will use a Socket to connect to a port on the remote machine. In this case, set up the data connection using the PASV command rather than the PORT command. This causes the server to establish a listen socket. This is important because a Java applet may not allow an incoming socket connection from the remote server.

The FTP class will provide a user id and password (collected from the user) to establish a valid session. From there, this class performs typical FTP types of functions, such as list, get, put, delete, etc.

Once there is an FTP class that can communicate with the remote server, one must create a class that uses the FTP class to submit a JCL job. This class knows the required behavior of the specific remote server. For example, to submit a JCL job to the OS/390, this class would first prepare the OS/390 session by telling it that the next command it receives will be a specific file type, a JCL job.

```
"site filetype=jes"
```

It would then send the JCL job via the FTP class, using the STOR command. The STOR command tells OS/390 that a file is about to be sent. The file is actually sent from Java to the OS/390 through the socket using a Java OutputStream class. With each command sent to the remote server, the remote server will return a response code, so there will also be a need to check for valid response codes and handle any errors.

Now that the file is on the remote server, the next step is to run the job. This is also done by sending a command using the FTP class:

```
"site filetype=seq"
```

At this point, the JCL job should start and run. Per the JCL example above, the Java code should start monitoring the remote server for a file named 'USERID.TXT'. Once the file shows up, Java can then use the FTP class to GET the file. The file will be read in as a String, which is then be presented to the user.

Again, this is a very simplified summary of submitting a JCL job from Java. It is recommended that programmers find a book, such as "Hacking Java" to help get started. Additionally, there are FTP classes now available for download from various sites on the Internet.

ADVANTAGES & DISADVANTAGES

There are several advantages when using SAS/Connect on OS/390 to submit a job to a remote host (which also has SAS/Connect installed).

- User is not required to have PC SAS on their local machine.
- Data doesn't have to be moved; instead, SAS/Connect allows execution of the SAS program on the host where the data resides.
- This approach leverages the power of the remote host, which is likely far more powerful than the local machine. For example, if the SAS program runs a CPU intensive procedure that will process a large number of rows then it is far more efficient to run this program on the powerful remote host instead of the user's PC.

Some disadvantages of this approach are:

- The developer who codes this piece of the application must be knowledgeable of the FTP protocol, JCL, SAS and Java programming.
- The application will experience delays due to connection time if there are frequent connections and disconnections, depending on the architecture.
- Once a job is submitted the application must wait until the job is actually run by the remote host.

Minor obstacles are to be expected and addressed when bringing together so many different hardware and software technologies. Here are a few items worthy of consideration:

- Do set `options = nocaps` on OS/390 jobs. Problems may be experienced with SAS/Connect from OS/390 to Unix due to the password being uppercased by SAS/Connect, thus causing logins to fail. With Version 8 of SAS this may no longer be an issue, but watch out for it.
- Be aware of remote host connection limits. OS/390 only allows one direct user connection and this cannot be set by a parameter. Therefore, if a user is logged on via a telnet session they will be unable to connect via SAS/Connect. Note that FTP sessions are not counted in this area and a user can be connected to OS/390 via FTP and still successfully connect via SAS/Connect.
- Consider how to monitor remote jobs for completion or failure. One approach is to always FTP either a validation log or error log to the target destination. The Java code can wait until one of these files exists before continuing. See the JCL code of Appendix B for an example of this approach.
- Take the extra step of capturing error messages on the remote host and handle these in an appropriate manner.

SUMMARY

REVIEW OF THE CODE SAMPLE IN APPENDIX B

As stated earlier, this code sample was generated by a Java program based upon the selections made by a user. Note that in this case the file is an OS/390 JCL file ready to be executed on the mainframe. After the job statement is an `exec` command that launches SAS.

Notice that a SAS macro is used in the job and is passed in following the JCL statement:

```
//SYSIN DD *
```

In the macro, an options line is used to define the remote system that will be accessed:

```
options remote=unix comamid=tcp nocaps;
```

The next line to note is the filename command that identifies the location of the SAS/Connect script file that will be used. That is immediately followed by the `signon` command that will trigger SAS to connect to the identified remote server.

```
filename rlink
'syst.sas.sas608.ctmisc(datcpunx)';
signon unix;
```

Lastly, the code between the `rsubmit` and `endrsubmit` accesses the remote server. In the example, the first occurrence of the `rsubmit` then does `a PROC UPLOAD` to transfer the file from OS/390 “client” to the UNIX system. In this case, the remotely submitted code is running on the UNIX host, which is viewed as the server.

Additional tasks are also included in Appendix B.

CONCLUSION

This paper highlights some of the advantages and disadvantages of creatively combining and leveraging the unique features of SAS, SAS/Connect and Java. The techniques presented allow extremely powerful and flexible systems to be built and deployed. Useful aspects of Java and SAS are introduced, and various methods of integration with SAS/Connect are explored, including dynamic and static SAS program generation. Additionally, techniques are discussed for submitting the generated programs in both local (PC) and remote (OS/390 mainframe, Unix, and NT servers) batch jobs. Advantages and disadvantages of each approach are reviewed. The techniques presented allow the creative systems developer to build time-saving applications that may not be easily possible with any single one of the discussed languages.

REFERENCES

Buchecker, M. M. 1996. "%FLATFILE, and Make Your Life Easier," Proceedings of the Twenty-First Annual SAS Users Group International Conference, 178-180.

Curnutt, R., M. J. Pell, and J. M. LaBore. 2001. “Energizing Users with a Slice of SAS® and a Cup of Java™”, These Proceedings.

Klenz, B. W., 1992. “Handling Numeric Representation Error in SAS Applications,” Observations, Vol. 1, Number 3,

SAS Institute Inc. (1996), *SAS Companion for the Microsoft Windows Environment*, Cary, NC: SAS Institute Inc.

SAS Institute, Inc. (1996*), SAS Companion for the MVS Environment, Version 6, Second Edition*, Cary, NC: SAS Institute Inc.

SAS Institute Inc. (1993), *SAS Companion for Unix Environments: Language, Version 6, First Edition*, Cary, NC: SAS Institute Inc.

SAS Institute Inc. (1994*), SAS/Connect Software: Usage and Reference, Version 6, Second Edition*, Cary, NC: SAS Institute Inc.

Wutka, M.,1997, “Hacking Java: The Java Professional’s Resource Kit”, Que Publishing, Indianapolis, IN

ABOUT THE AUTHORS

Randy Curnutt, Solutions Plus, Inc. (http://www.sol-plus.com)
Randy Curnutt is the president of Solutions Plus, Inc., a software consulting company that specializes in applying leading edge technologies in order to provide comprehensive solutions to its clients. He focuses on client/server solutions, especially object oriented technology, and relational database management systems. He has experience with Java, Smalltalk, Visual Basic, C, C++, Oracle, MS SQLServer, and numerous other development languages.

Michael Pell, Solutions Plus, Inc. (http://www.sol-plus.com)
Michael Pell is a consultant at Solutions Plus, Inc., a software consulting company that specializes in applying leading edge technologies in order to provide comprehensive solutions to its clients. He focuses on the analysis, design, and implementation of object oriented technology client/server solutions. Michael has 3 years of Java development experience, and spent 6 years as an IBM consultant prior to joining Solutions Plus, Inc.

John LaBore, Eli Lilly And Company (http://www.lilly.com)
John LaBore is the SAS and JMP Coordinator for Eli Lilly and Company, a leading innovation-driven pharmaceutical corporation. He is responsible for supporting SAS and JMP use by Lilly staff worldwide. John has been a SAS software user for more than 20 years, and has authored numerous SAS technical papers for SUGI, PharmaSUG, SEUGI, and other SAS user group conferences.

ACKNOWLEDGMENTS

The authors appreciate the assistance of Fred Forst and Thomas H. Burger in reviewing technical content of this paper. We also appreciate the editing assistance provided by Linda E. LaBore.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors at:

Randy Curnutt
Solutions Plus, Inc.
10401 North Meridian, Suite 300
Indianapolis, IN 46217
(317) 848-3081
rcurnutt@sol-plus.com
http://www.sol-plus.com

Michael Pell
Solutions Plus, Inc.
10401 North Meridian, Suite 300
Indianapolis, IN 46217
(317) 848-3081
mjpell@sol-plus.com
http://www.sol-plus.com

John LaBore
Eli Lilly And Company
Lilly Corporate Center
Drop Code 6334
Indianapolis, IN 46285
(317) 277-6387
jml@lilly.com
http://www.lilly.com

TRADEMARK NOTICE

APPENDIX A

The generated PC SAS program can be written to a temporary file, then PC SAS can be run using the generated SAS program (all done in the Java method as shown below).

```
  /** Writes the PC SAS program to disk, then runs it.
  * @param sasProgramString String of the SAS program to be executed
  * @return int the return code returned by the executed SAS program
  */
public int runSasProgram(String sasProgramString) throws TransportFailedException
{
    int returnCode = 0;
    try
    {
            //Write the file to disk
            File sasProgramFile = new File("C:\\SAS\\GenProg.sas");
            BufferedReader buffReader = new BufferedReader(new StringReader(sasProgramString));
            PrintWriter sasProgOut = new PrintWriter(new FileOutputStream(sasProgramFile));
            printDIStreamOnPrintWriter(buffReader, sasProgOut);
            Runtime rt = Runtime.getRuntime();
            sasProgOut.close();

            //Now run the program using PC-SAS.  This is equivelant to starting PC-SAS from a
            //DOS command line
            StringBuffer programSb = new StringBuffer("C:\\SAS");
            programSb.append("\\SAS.exe -NOSPLASH -ICON -SYSIN \"");
            programSb.append(sasProgramFile.getPath());
            programSb.append("\"");
            Process myProc = rt.exec(programSb.toString());
            returnCode = myProc.waitFor(); //wait here until SAS is done running
            SasProgramFile.delete();  //be a good neighbor and clean up
    }
    catch (Exception e)
    {
            throw new TransportFailedException("Error communicating with PC-SAS software.");
    }
    return returnCode;
}
```

APPENDIX B

The following is an example of a JCL job that contains a SAS program that uses SAS/Connect to transfer a SAS dataset to a UNIX server. It also provides a summary report about the transferred data's integrity. If the job detects an error, then it will transfer an error log rather than the data requested.

```
//MYUSERID JOB (,8305,S),'DAT35TestSasToUnix',MSGCLASS=T,CLASS=T
//*-----------------------------------------------
//JS010      EXEC PROC=SAS609,SOUT=T
//*-----------------------------------------------
//MVSIN    DD DSN=ABC_DATA,
//            DISP=(SHR,KEEP,KEEP)
//VALLOG  DD DSN=MYUSERID.F0545615.VALLOG,
//            DISP=(NEW,PASS,DELETE),
//            SPACE=(10796,(250,500),RLSE),
//            UNIT=WKDISK,
//            BLKSIZE=0,
//            LRECL=120,
//            RECFM=FB
//ERRNAME  DD DSN=MYUSERID.F0545615.ERRORS.TXTT,
//            DISP=(NEW,PASS,DELETE),
//            SPACE=(10796,(250,500),RLSE),
//            UNIT=WKDISK,
//            BLKSIZE=0,
//            LRECL=120,
//            RECFM=FB
//SYSIN    DD *
%macro sasToUx(userId=,
               pswrd=,
           inclause=,
```

```
           rowconst=,
               cols=,
               file=,
           targHost=,
           targDir=,);
%let unix=&targHost;
%let userId=&userId;
%let pswrd=&pswrd;
%let inclause=%upcase(&inclause);
%let rowconst=%upcase(&rowconst);
%let cols=%upcase(&cols);
%let file=%upcase(&file);
%let targHost=%upcase(&targHost);
%let targDir=%upcase(&targDir);
options remote=unix comamid=tcp nocaps;
options sgen mprint;
filename rlink 'syst.sas.sas608.ctmisc(datcpunx)';
signon unix;

PROC PRINTTO log=ERRNAME;
proc sql;
create table WORK.INV as
    select &cols
    from MVSIN.INV &rowconst ;

* Create the first comparison dataset  ;
create table WORK.valid1 as
  select nobs,obslen,nvar
  from dictionary.tables
  where libname = 'WORK'
  and memname = 'INV';

* Go to UNIX ;
rsubmit;

libname unixOut "/home/myUserId";

*Move the MVS work.&dsn file to Unix  ;
proc upload
  data=WORK.INV
  out=unixOut.INV;
run;

*Move the 1st comparison file to Unix  ;
proc upload data=WORK.valid1
  out=WORK.valid1;
run;

* move the transferred file back to MVS ;
PROC download data=unixOut.INV
     out=WORK.xfer;
run;

endrsubmit;

*Must be done on MVS to get the dictionary.tables;
proc sql;
create table WORK.valid2 as
  select nobs,obslen,nvar
  from dictionary.tables
  where libname = 'WORK'
  and memname = 'INV';

* Go back to UNIX ;
rsubmit;

* move the validation dataset to Unix ;
proc upload data=work.valid2
  out=work.valid2;

* Perform the comparison ;
proc compare  base=WORK.valid1
```

```
     compare=WORK.valid2
     out=WORK.vallog noprint;
run;

data _null_;
    set WORK.vallog;

(DATASTEP CODE TO COMPARE NUMBER OF OBSERVATIONS, OBSERVATION LENGTH, AND NUMBER OF VARIABLES DELETED
FOR BREVITY)

run;
endrsubmit;

proc printto print=VALLOG;

* Perform the comparison ;
proc compare  base=WORK.INV
     compare=WORK.xfer
     outstats=WORK.vallog;
run;

* Perform the PROC contents ;
proc contents data=WORK.xfer;

proc printto;

signoff unix;
%mend sasToUx;
%sasToUx(userId="myUserId",
pswrd="mypass",
inclause=
    and name in ('CLINVNO', 'COUNTRYC', 'GLBINVNO'),
rowconst=WHERE clinvno>"804";,
cols=%STR(CLINVNO, COUNTRYC, GLBINVNO),
file=inv.ssd01,
targHost=unixhostnamee@mycompany.com,
targDir=/home/myUserId);
run;
/*
//*******************************************************************
//*  IF JOB DOES NOT ABEND AND RETURN CODE <= 4 THEN RENAME
//*******************************************************************
//IFGOOD   IF (^ABEND & JS010.SAS.RC <= 4) THEN
//FTP      EXEC  PGM=FTP,PARM='(EXIT'
//OUTPUT   DD SYSOUT=*
//INPUT    DD *
  PASCAL
  myUserId
  mypass
  RENAME /home/myUserId/invval.logT invval.log
  PUT 'MYUSERID.F0545615.ERRORS.TXTT' /home/myUserId/invERRORS.TXT
QUIT
/*
//*******************************************************************
//*  ELSE TRANSFER THE ERROR LOG
//*******************************************************************
//ELSE     ELSE
//FTP      EXEC  PGM=FTP,PARM='(EXIT'
//OUTPUT   DD SYSOUT=*
//INPUT    DD *
  PASCAL
  myUserId
  mypass
  PUT 'MYUSERID.F0545615.ERRORS.TXTT' /home/myUserId/invERRORS.TXT
  DELETE /home/myUserId/invval.logT
  DELETE /home/myUserId/inv.ssd01
  QUIT
//ENDIF    ENDIF
//
```

HANDS ON WORKSHOPS

SECTION CHAIR

Heidi Markovitz
Simply Systems

HANDS ON WORKSHOPS

Who Needs To Know Program Syntax When You Have Enterprise Guide?

Prepared by

destiny CORPORATION

International SAS® Training and Consulting
A SAS Institute Quality Partner™

100 Great Meadow Rd, Suite 601
Wethersfield, CT 06109-2379
Phone: (860) 721-1684 1-800-7TRAINING
Fax: (860) 721-9784
Web: www.destinycorp.com Email: info@destinycorp.com

Introduction

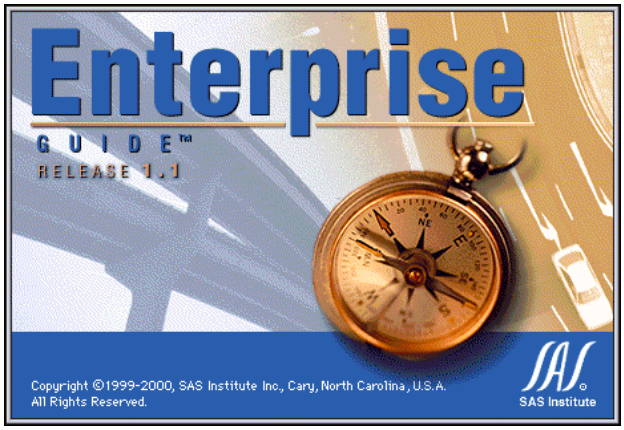
Why use Enterprise Guide?

Programming in SAS has always been the biggest obstacle for Analysts. This course will demonstrate an incredible new tool that will allow you to create reports, statistics, graphs and assemble data in seconds flat. This software will also build the program for you with perfect syntax. You don't have to code anything. This is the best tool SAS Institute has developed for day to day SAS user needs. You're programming and programs will never be the same again. Folks, this is better than sliced bread.

Starting Enterprise Guide

Locate the 

Icon or selection on your Start Menu. Click it to bring up the splash screen.

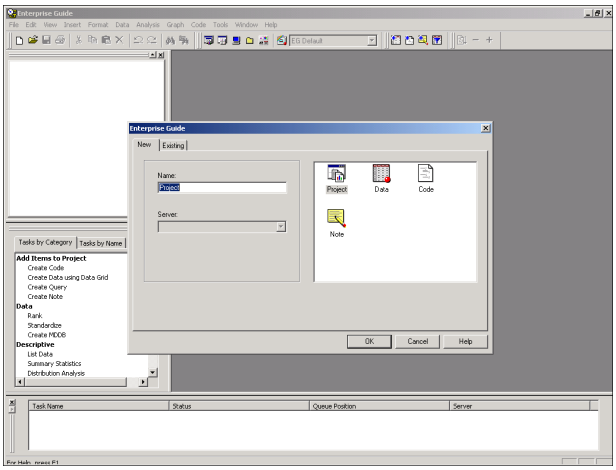


The application will load.

Projects

Everything you do in Enterprise Guide is done in a project. These are similar to folders that can contain many types of analysis. It is possible to have several project folders.

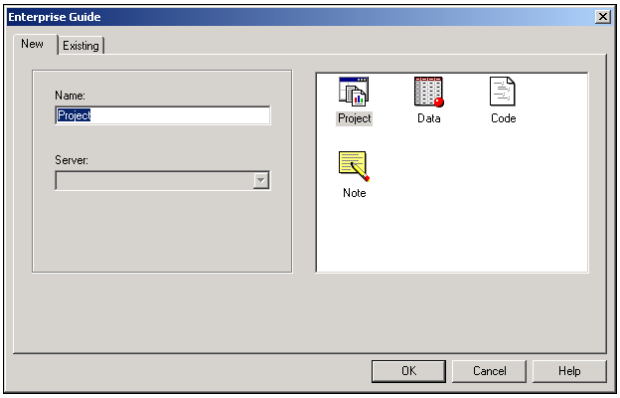
Many people don't just work on one project at a time. This software is designed to allow you to manage multiple analytical projects on your machine. You can switch between projects as required.



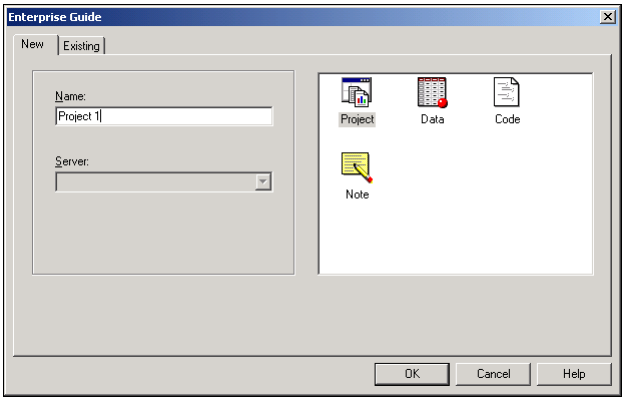
Notice Merlin. He is your personal assistant who can guide you through your use of Enterprise Guide.

Note: It is best to have the appropriate multimedia computer system to make use of this feature.

You will be prompted to enter a name for your Project.

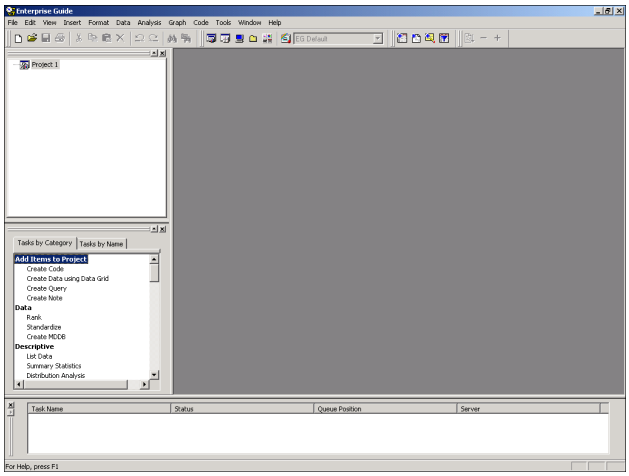


We will use Project 1 for this demonstration.

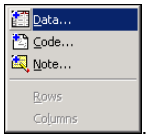


Click 

to enter the Enterprise Guide Desktop.

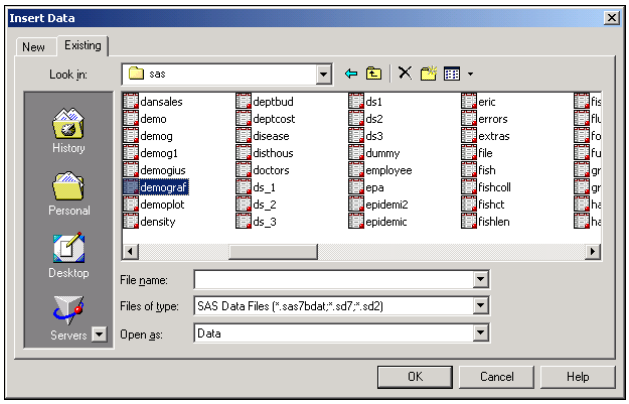


Now that we have a project, we must insert some data into the project so we can analyze it.

Select the Insert pull down menu and 

.

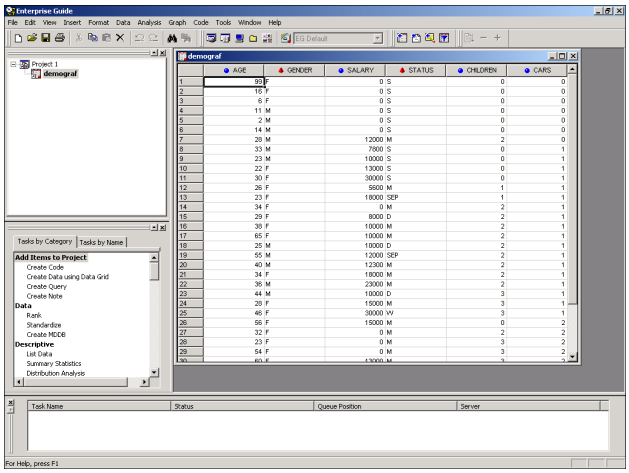
The following prompt window will appear, asking you to select data. Data can come from almost anywhere, including tables from relational databases stored on other operating systems.



Select the DEMOGRAF data set from the course library (see your instructor for specific locations) and click 

.

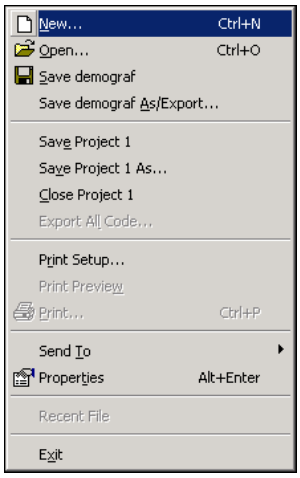
Enterprise Guide will now analyze the data, determining variables available and their types. When finished, the data will be loaded onto the desktop.



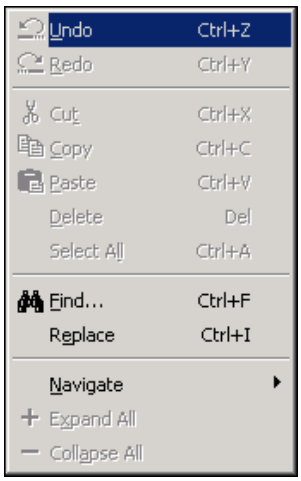
Notice the data set demograf listed in the upper left corner of the desktop as part of Project 1.

There are many pull down menus available to do all of your work in Enterprise Guide. No coding is necessary. An overview of all of the items will give you a better feel as to what the software will be able to do.

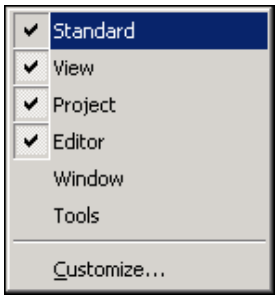
File Menu



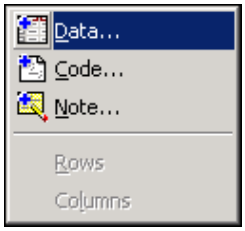
Edit Menu



View Menu

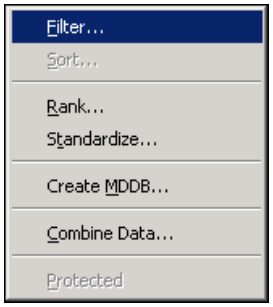


Insert Menu

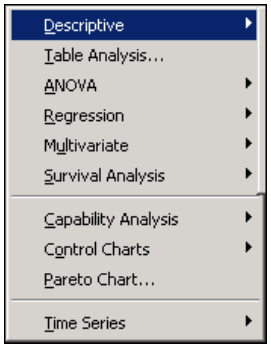


Format Menu

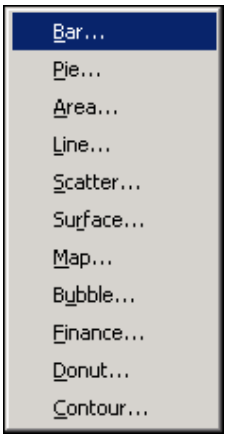
Data Menu



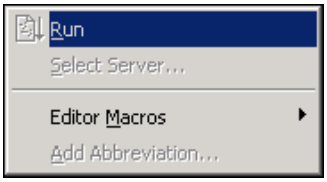
Analysis Menu



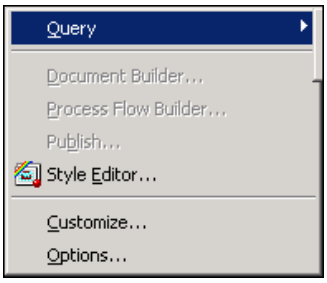
Graph Menu



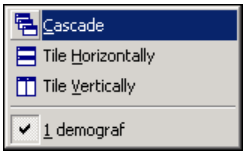
Code Menu



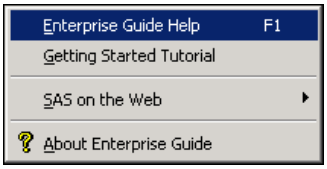
Tools Menu



Window Menu



Help Menu

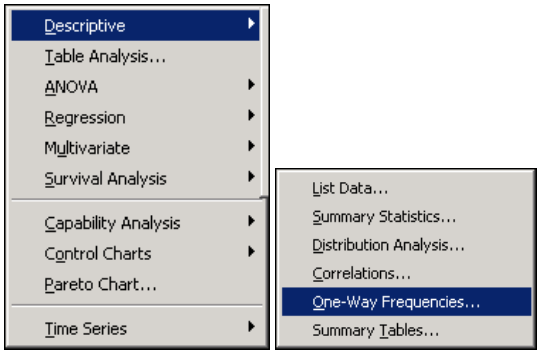


Basic Analysis On Your Data

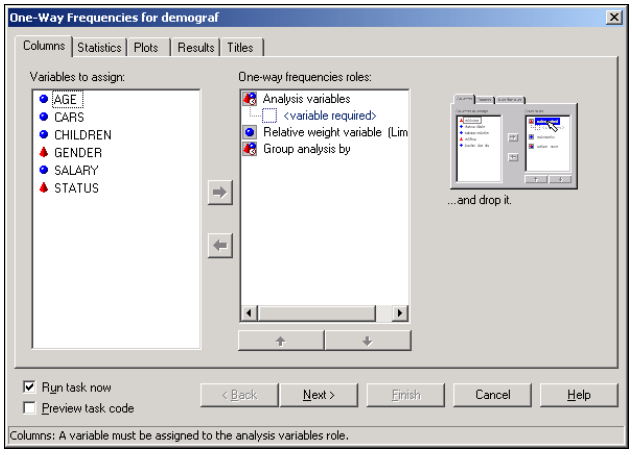
Let's perform a simple analysis on the data to demonstrate the power of this software.

Given the data we have loaded, we will create a simple Frequency Analysis on the variable Status.

Select the Analysis pull down menu



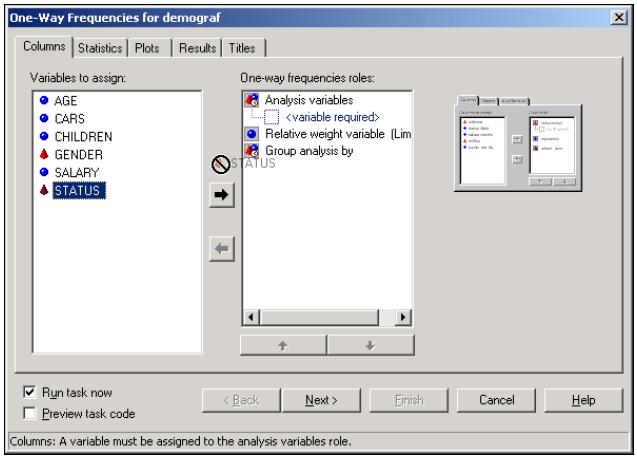
The following window will appear.



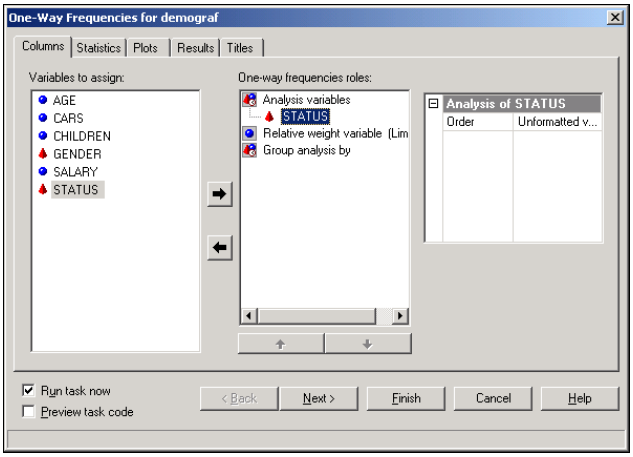
The variables available are listed on the left and are color and symbol coded based on numeric or character attributes.

The Frequency attributes are listed in the middle and some simple instructions are listed on the right.

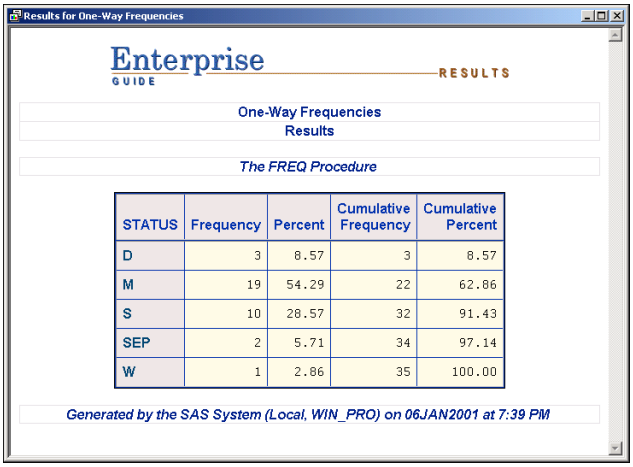
Drag the variable STATUS onto the Analysis variables attribute



to achieve the following.

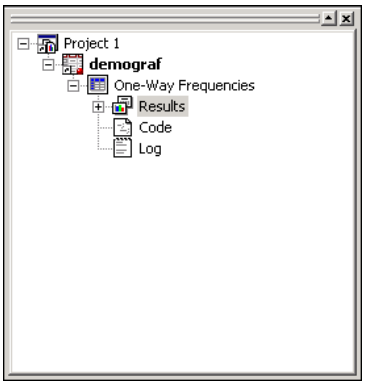


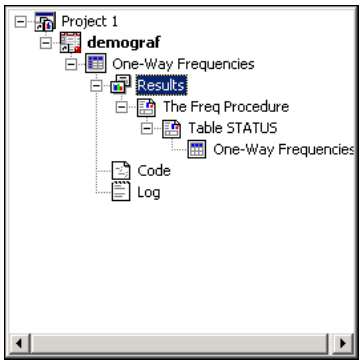
Click Finish when done and the following report will appear.



STATUS	Frequency	Percent	Cumulative Frequency	Cumulative Percent
D	3	8.57	3	8.57
M	19	54.29	22	62.86
S	10	28.57	32	91.43
SEP	2	5.71	34	97.14
W	1	2.86	35	100.00

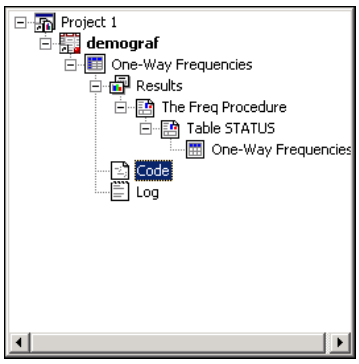
You can see the results in the upper left hand window. Click on the Results item to expand it.



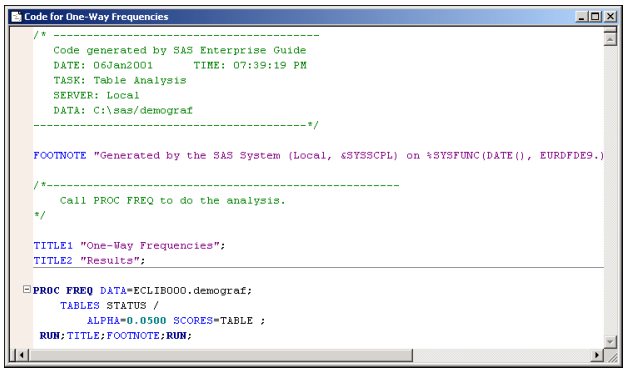


You will see the listing of the HTML report, a One-Way Frequency analysis.

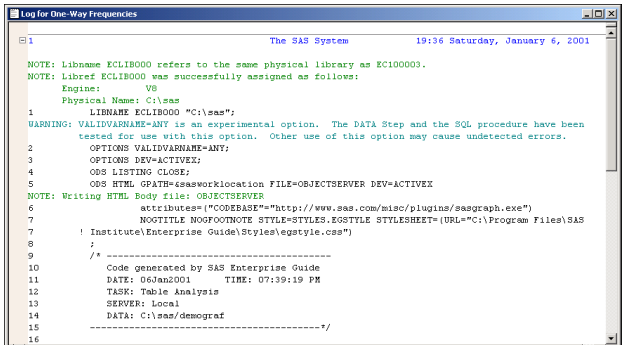
Let's examine the code that was created.

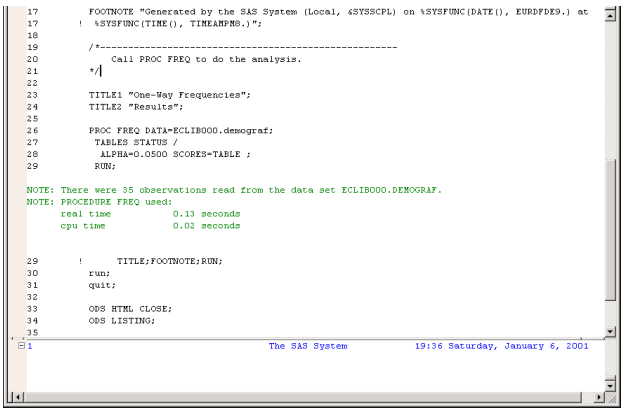


Double click on the Code item to view it in the Enhanced Editor.



Double click on the Log item to view it.

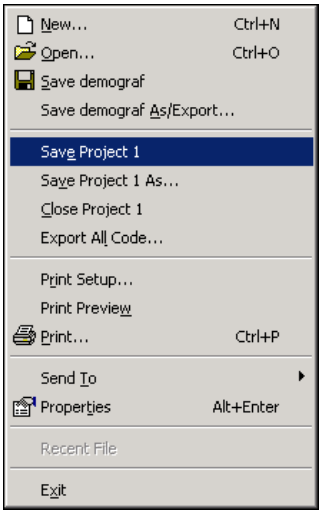




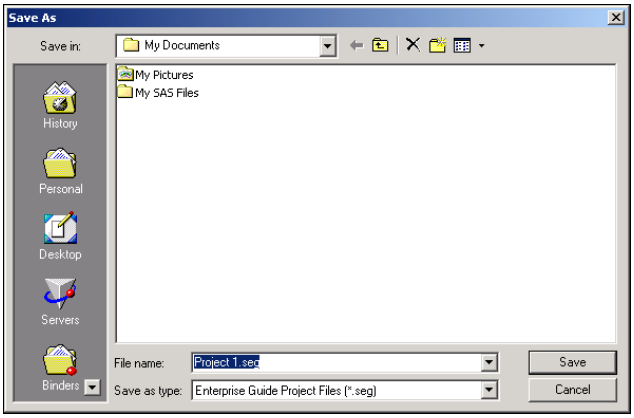
Saving Projects

To save the analysis created and stored in this project, it needs to be saved into a Project folder.

To do this, select the File pull down menu,



and specify the location for this project. You will need to know this location when you return to Enterprise Guide and want to use the analysis in this project.



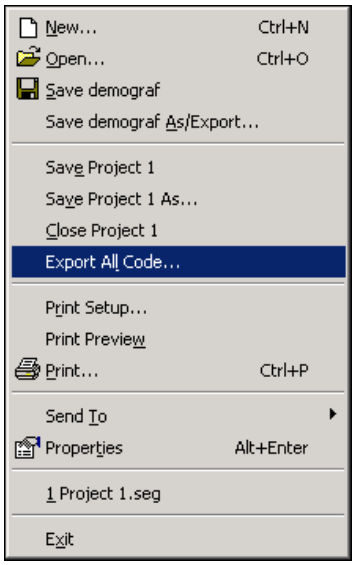
Type the name of the Project and click Save when complete.

Exporting Code

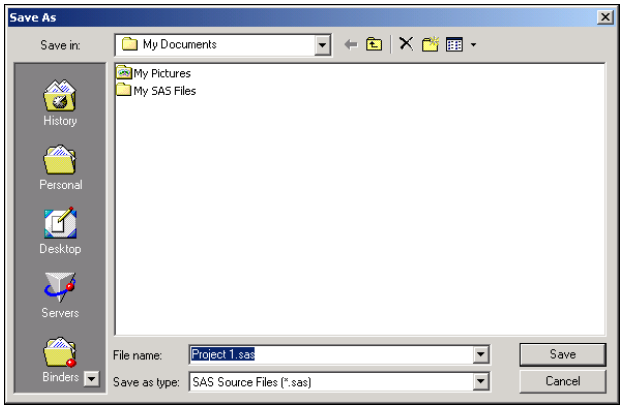
Many analysts and programmers use Enterprise Guide for quick creation of SAS programs that can be used in a standard SAS

environment. The code created to perform the analysis in each project can be exported for use outside Enterprise Guide.

To do this, from the File pull down menu, select

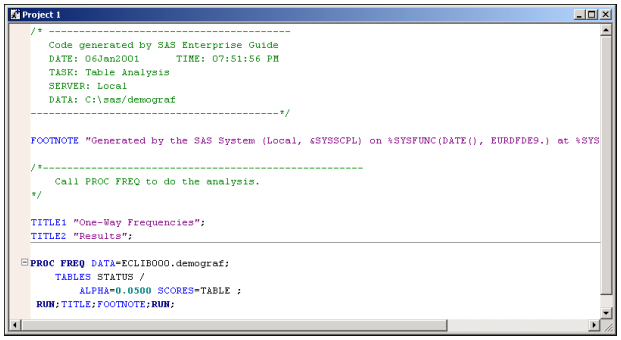


Choose a destination location.



Give the program a name and click Save when finished.

In SAS, when the file is opened, it looks like the following.



It is important to note that librefs need to be adjusted. This process is only designed for saving the code and not the libref pointers used inside the Enterprise Guide session.

Basic Macro Processing

Prepared by

Destiny Corporation

International SAS® Training and Consulting

Destiny Corporation
100 Great Meadow Rd Suite 601
Wethersfield, CT 06109-2355
Phone: (860) 721-1684 1-800-7TRAINING
Fax: (860) 721-9784
Email: info@destinycorp.com
Web: www.destinycorp.com

Macro variables

In this module we discuss the first of the two special characters - the ampersand (&).

When the SAS Supervisor sees an ampersand followed by a non-blank character, the macro facility is triggered. In turn, the macro facility, determines the value for the macro variable and passes the value back on to the input stack.

If the macro facility fails to find the current value for a macro variable, the following message appears on the SAS Log:

```
Warning: Apparent symbolic reference is not resolved.
```

In version 5 systems this message is accompanied by the number 1301:

```
Warning 1301: Apparent symbolic reference is not resolved.
```

Strings enclosed within single blip quotes (') are always assumed to be one continuous string. Therefore, macro variable references within single quotes will NOT resolve. Tokens enclosed within double blip quotes (") are treated separately and thus macro variable references WILL resolve.

In release 5.18 and below, the facility needs to be enabled by:

```
OPTION DQUOTE;
```

In version 6 the system works as though DQUOTE is always enabled.

Predict the results of the following statements. Assume you are working on a version 6 system.

PROGRAM EDITOR

```
title 'Macro facility involves % and &';
title2 'Macro facility came as a result of much R&D';
title3 "Macro facility came as a result of much R&D";
title4 "Report prepared on &sysdate";
title5 'Report prepared on &sysday';
```

Automatic Macro Variables

When SAS is invoked, a set of Automatic macro variables is created; some of these are read-only, others are read-write. We have encountered two of these so far -&SYSDATE and &SYSDAY.

There are certain variations in the variables available in different versions of the systems as shown below. Note that these are the automatic macro variables available with the base product only.

Macro variables and their current values are stored in internal work areas called Symbol Tables. When the macro processor is trying to resolve a macro variable reference it will scan its symbol tables for a macro variable of that name and either retrieve its value and place that value upon the input stack, or the message

```
Warning: Apparent Symbolic reference&macro-variable not resolved.
```

will be issued.

When the SAS system is invoked (and assuming system options enable the macro facility) the GLOBAL SYMBOL TABLE is built. This holds most of the automatic macro variables.

Later we shall see other symbol tables created and deleted dynamically, how we can force variables into different tables and how the tables have a prescribed search order.

Creating macro variables

Within the macro facility and Data step language there are no fewer than 9 ways of creating a macro variable. We shall see all 9 methods throughout this course.

Let us begin with the simplest method, the %LET statement. The presence of a % followed by a non-blank character triggers the macro facility. A %LET tells the macro facility that a macro variable is to be defined. %LET can appear anywhere within a SAS program to define one macro variable at a time.

The form of the %LET statement is:

```
%LET macrovariable = value;
```

- value is optional:

```
%LET price=;
```

The macro variable price is created but takes a null value.

- trailing and leading blanks are ignored:

```
%LET cost=  very cheap  ;
```

creates a macro variable called cost with the value of very cheap. Note the embedded blank is included but no blanks before very or after cheap are included.

- excepting the case above, all characters are included as part of the macro variable including any quotes used.

```
%LET cost='   "very cheap"   ';
```

creates a macro variable called cost with the value

```
'   "very cheap"   '
```

including all quotes and spaces. Note that you cannot do

```
%LET cost='   "very cheap"   ;
```

This will give unbalanced quotation mark errors.

- inclusion of special characters in a macro variable value (except & or %) or leading or trailing blanks can be achieved by use of the %STR function:

```
%LET code=proc print;

%LET code2=%str(&code;run;);
```

code1 takes the value of proc print
code2 takes the value of proc print;run;

- to include & or % as part of the string use %NRSTR:

```
%LET code=proc print;

%LET code2=%nrstr(&code;run;);
```

code1 takes the value of proc print
code2 takes the value of &code;run;

- to inspect the variable's value simply use the %PUT statement to write the value of the Macro variable out to the log. Remember to reference it with an ampersand:

```
%PUT &code;
```

Writes the value of the macro variable code to the SAS Log.

PROGRAM EDITOR

```
%let station1=Paddington;
%put &station1;

%let station2=Clapham Junction;
%put &station2;

%let prefix1=St;
%let name1=Pancras;
%let station3=&prefix1 &name1;
%let suffix1=Street;
%let name2=Liverpool;
%let name3=Cannon;
%let station4=%str(   Victoria   );
%put &station4;

%let station5=&name2 &suffix1;
%put &station5;

%let station6=              &name3 &suffix1;
%put &station6;

%let station7='Tottenham Court Road';
%put &station7;

%let station8=' Tottenham Court Road;Oxford Circus ';
%put &station8;

%let station9=%nrstr(Highbury&Islington);
%put &station9;

%let stationa=%str(Chalfont&Latimer);
%put &stationa;

%let stationb=%str(Harrow&Wealdstone);
%put &stationb;

%let stationc="&station7";
%put &stationc;
```

These %LET and %PUT statements write the following Log:

LOG

```
NOTE: The PROCEDURE PRINTTO used 0.11 seconds.


MPRINT(PROGRAM):   DM 'clear log; clear out';
SYMBOLGEN:  Macro variable N resolves to m234
SYMBOLGEN:      Macro variable STATION1 resolves to
Paddington
Paddington
SYMBOLGEN:  Macro variable STATION2 resolves to Clapham
Junction
Clapham Junction
SYMBOLGEN:  Macro variable PREFIX1 resolves to St
SYMBOLGEN:  Macro variable NAME1 resolves to Pancras
SYMBOLGEN:  Macro variable STATION4 resolves to   Victoria
SYMBOLGEN:  Some characters in the above value which were
          subject to macro quoting have been unquoted for
          printing.
  Victoria
SYMBOLGEN:  Macro variable NAME2 resolves to Liverpool
SYMBOLGEN:  Macro variable SUFFIX1 resolves to Street
SYMBOLGEN:  Macro variable STATION5 resolves to Liverpool
Street
Liverpool Street
SYMBOLGEN:  Macro variable NAME3 resolves to Cannon
SYMBOLGEN:  Macro variable SUFFIX1 resolves to Street
SYMBOLGEN:  Macro variable STATION6 resolves to Cannon
Street
Cannon Street
SYMBOLGEN:      Macro variable STATION7 resolves to
'Tottenham
          Court Road'
'Tottenham Court Road'
SYMBOLGEN:    Macro variable STATION8 resolves to '
Tottenham
          Court Road;Oxford Circus '
' Tottenham Court Road;Oxford Circus '
SYMBOLGEN:  Macro variable STATION9 resolves to
          Highbury&Islington
SYMBOLGEN:  Some characters in the above value which were
          subject to macro quoting have been unquoted for
```

```
        printing.
Highbury&Islington
WARNING: Apparent symbolic reference LATIMER not resolved.
SYMBOLGEN:  Macro variable STATIONA resolves to
Chalfont&Latimer
SYMBOLGEN:  Some characters in the above value which were
        subject to macro quoting have been unquoted for
        printing.
WARNING: Apparent symbolic reference LATIMER not resolved.
Chalfont&Latimer
ERROR: Symbolic variable name WEALDSTON must be 8 or
fewer
     characters long.
SYMBOLGEN:  Macro variable STATIONB resolves to
        Harrow&Wealdstone
SYMBOLGEN:  Some characters in the above value which were
        subject to macro quoting have been unquoted for
        printing.
ERROR: Symbolic variable name WEALDSTON must be 8 or
fewer
     characters long.
Harrow&Wealdstone
SYMBOLGEN:  Macro variable STATION7 resolves to
'Tottenham
        Court Road'
SYMBOLGEN:  Macro variable STATIONC resolves to
"'Tottenham
        Court Road'"
"'Tottenham Court Road'"
MPRINT(PROGRAM):  PROC PRINTTO;
MPRINT(PROGRAM):  RUN;
```

Notes on the log:

- %LET does not allow leading or trailing blanks - for them use quotes or the %STR function.

- Quotes remain part of the macro variable value. Note that blanks can be included and other special characters too - see Station8.

- Macro variable references can themselves be used within %LET statements.

- Macro variable names conform to usual SAS naming conventions. Note that although WEALDSTONE is 10 characters long, the Log reports WEALDSTON - 9 Characters. Once the limit of 8 characters is exceeded SAS reports the error.

- %STR( ) hides the meaning of blanks and special characters, but not & and %. A macro variable reference will resolve within the function.

- %NRSTR( ) hides the meaning of the % and& as well as blanks and other special characters. A macro variable reference within the argument to the function will NOT resolve.

We have seen that the Global Symbol Table is built at SAS invocation time. All the %LETs in the example above will also be written to this table, and after they have executed the table will contain the following:

Definition of a Macro

As the module title implies, a macro is simply 'a bundle of code'. In its simplest form, the regular job you submit to backup your data sets or produce your graphics can all be bundled up into a macro and then 'invoked' using one word - the name of the macro. This is known as making a macro 'call'.

But as you would expect, the macro facility can do far more than this. For example, we shall see how to pass values to the macro, and, in Module 5, how to take decisions within the macro.

All this simply leads to the inclusion of text (normally ordinary SAS code - DATA and PROC steps) on to the top of the input stack, to be submitted to SAS in the normal way.

Example

Consider a regular job to age a series of time-related data sets; there are four data sets in the group. A new data set forms the first, newest member, the former first data set becomes the second and so on:

PROGRAM EDITOR

```
data mylib.new;
  ...
  ... data step statements ...
  ...
proc datasets lib=mylib;
  age new first second third fourth;
run;
```

new	is renamed	first
first	is renamed	second
second	is renamed	third
third	is renamed	fourth
fourth	is deleted	

How can we bundle this code up into a macro?

Macro definition

- A macro must be defined first before it can be called.

- Macro definitions start with the %MACRO statement which defines the name of the macro...

- ... and the definition is deemed to continue until the %MEND statement. Including the macro name is optional (but if it is used must correspond with the name of the macro):

```
%macro macroname;
    .
    macro programming statements;
    .
%mend macroname;
```

The above example thus becomes:

PROGRAM EDITOR

```
%macro age; /*definition of macro called age*/   proc
datasets lib=mylib;
    age new first second third fourth;
  run;
%mend age; /*completion of macro definition*/
```

N.B. The %MEND is critical. The macro processor takes control when a %MACRO statement is seen. Should the %MEND be missing, all the input stream is regarded as being part of the open macro definition. There are occasions when all the submitted code is seen to be written to the log and nothing else - the code appears to be disappearing into a black hole! Upon such occasions, check for the absence of a %MEND statement.

What can a macro contain?

- Data and Proc step code
- Macro programming statements and functions

We shall see these in Module M5. Some macro programming statements must be within the bounds of a macro definition, others are totally global.

- Macro variable references
  i.e. things starting with the &
- Other macro calls and definitions
  i.e. things starting with the %

In the example we have just seen, the name of the data sets are fixed. We shall see how to make these variable shortly.

What can a macro be called?

- Any valid SAS name (CMS users are limited to a maximum of 7 characters)
- Anything other than one of the following reserved words:

Reserved words in the macro facility (Release 6.06 and higher):

ABEND	GOTO	QUOTE
ABORT	IF	QUPCASE
ACT	INC	RESOLVE
ACTIVATE	INCLUDE	RETURN
BQUOTE	INDEX	RUN
BY	INFILE	SAVE
CLEAR	INPUT	SCAN
CLOSE	KEYDEF	STOP
CMS	LENGTH	STR
COMANDR	LET	SUBSTR
COPY	LIST	SUPERQ
DEACT	LISTM	SYSEXEC
DEL	LOCAL	SYSGET
DELETE	MACRO	SYSRPUT
DISPLAY	MEND	THEN
DMIDSPLY	METASYM	TO
DMISPLIT	NRBQUOTE	TSO
DO	NRQUOTE	UNQUOTE
EDIT	NRSTR	UNSTR
ELSE	ON	UNTIL
END	OPEN	UPCASE
EVAL	PAUSE	WHILE
FILE	PUT	WINDOW
GLOB	QSCAN	
GO	QSUBSTR	

An attempt to call a macro by one of the above reserved names will result in a warning message; the macro will neither be compiled or available for use.

The compiled macro

- A compiled macro is an entry in a utility catalog in the WORK library

- the system does not support the renaming or copying of entries of member type of macro

The once-defined macro is stored in the WORK data library in compiled form. In versions prior to 6.03 it was stored as a special type of data set; in version 6.03 and above, it is stored in the WORK.SASMACR catalog with an entry type of MACRO. As it is held in 'compiled' form, it is, of course, not browsable or editable.

The macro could also be part of an autocall library. This topic will be discussed at length in a later module.

The Macro Call

Once a macro has been defined it can be called anywhere in a SAS job. The call is simply the name of the macro preceded by the % sign:

PROGRAM EDITOR

```
%macro age; /*definition of macro called age*/
  proc datasets lib=mylib;
    age new first second third fourth;
  run;
%mend age; /*completion of macro definition*/
data mylib.new;
  set...;
  if....;
run;
%age              /*the macro call*/
```

Notice that the macro call, %age , does not include a semi-colon. There is no need here as a semi-colon has been generated by the macro call; the code within the definition is complete, so no extra semi-colon is required.

The macro call sees the execution phase of the macro, whereupon the macro processor executes the macro in sequential form, placing the resulting open code (i.e. simple DATA and PROC steps) upon the input stack.

Passing Parameters

In the %age example the data set names were fixed. How could a macro be written such that the procedure is invoked with any names for the library and data sets involved? I.e. to generate:

PROGRAM EDITOR

```
proc datasets lib=mylib;
age new first second third fourth;
run;
```

at the first invocation and to generate

PROGRAM EDITOR

```
proc datasets lib=newlib;
age next ds_0 ds_1 ds_2;
run;
```

at the next?

The way to do this is to pass parameters to the macro call.

To use this method, the macro must first be defined as requiring parameters in the call. There are two ways of doing this:

Positional Parameters

PROGRAM EDITOR

```
data new;    x=5; run;
```

```
data first ; x=1; run;
data second; x=2; run;
data third ; x=3; run;
data fourth; x=4; run;

%macro age(library,newds,ds1,ds2,ds3,ds4);
proc datasets lib=&library;
  age &newds &ds1 &ds2 &ds3 &ds4;
run;
quit;
%mend;

%age(work,new,first,second,third,fourth)
```

Here the macro age has been defined with six positional parameters to take the variation in library and data sets - in order.

The macro is invoked by:

```
%age(sasdata,latest,prod1,prod2,prod3,prod4)
```

to generate:

PROGRAM EDITOR

```
proc datasets lib=sasdata;
age latest prod1 prod2 prod3 prod4;
run;
```

What would the call

```
%age(prod1,prod2,prod3,prod4,sasdata,latest)
```

generate?

Keyword Parameters

This method does exactly the same job as defining a macro with positional parameters, except:

- it gets over the requirement to define and pass parameters in the same order
- it allows default values to be attached to the parameter

PROGRAM EDITOR

```
data new;    x=5; run;
data first ; x=1; run;
data second; x=2; run;
data third ; x=3; run;
data fourth; x=4; run;

%macro
age(library=work,newds=new,ds1=,ds2=,ds3=,ds4=);
proc datasets lib=&library;
  age &newds &ds1 &ds2 &ds3 &ds4;
run;
quit;
%mend;

%age(ds1=first,ds2=second,ds3=third,ds4=fourth)
```

The call

```
%age(ds1=first,ds2=second,ds3=third,ds4=fourth)
```

generates...

PROGRAM EDITOR

```
proc datasets lib=mylib;
age work1 first second third fourth;
run;
```

the definition of the macro providing default values for &library and &newds.

The call

```
%age(ds2=april,ds4=june,ds1=march,ds3=may,library=y
earlib)
```

generates

PROGRAM EDITOR

```
proc datasets lib=yearlib;
age work1 march april may june;
run;
```

Where the macro is defined with parameters the parentheses MUST be used. For example, where all the parameters are given default values, the minimum invocation is:

```
%age()
```

Null values

With positional parameters, null values can be passed by using a comma as a 'placeholder':

PROGRAM EDITOR

```
%macro rr(datads,setds,condval);
  data &datads;
    set &setds;
    if category="&condval";
  run;
%mend rr;

%rr(work9,saved.epidemic,E)
%rr(,saved.epidemic,E)
```

```
%rr(,work3,F)
```

will generate:

PROGRAM EDITOR

```
data;
set work3;
if key="F";
run;
```

and SAS will choose the name of the temporary output data set.

```
%rr(,,M)
```

will generate:

PROGRAM EDITOR

```
data;
set;
if key="M";
run;
```

and SAS will choose the name of the temporary output data set and use the last updated data set as input.

With keyword parameters, the parameter is simply omitted.

Combination of Positional and Keyword parameters

If the methods of positional and keyword parameters are mixed, the positional parameters must come first.

PROGRAM EDITOR

```
%macro tt(proc,dataset=_last_);
  proc &proc data=&dataset;
  run;
%mend tt;
```

The call...

```
%tt(print,dataset=first)
```

...generates

PROGRAM EDITOR

```
proc print data=first;
run;
```

Variable Numbers of Parameters

Sometimes you may want to write a macro to contain variable numbers of parameters. For example, the %age macro in its forms defined so far can only age 5 data sets; what if we wanted to write a utility macro so we could age any number of data sets. A way around this is to use the PARMBUFF option.

- Define the macro in the normal way except for the /PARMBUFF option.

```
%macro age/parmbuff;

  macro programming statements

%mend age;
```

Here, all supplied parameters, including any special characters used are assigned to the automatic local macro variable SYSPBUFF which is then manipulated in the macro by macro programming statements.

The call

```
%age(library=mylib,new,gdg_0,gdg_1,gdg_2)
```

gives a value to &syspbuff of
library=mylib,new,gdg_0,gdg_1,gdg_2

Parameters may also be included in the definition

```
%macro age(posparm)/parmbuff;

  macro programming statements

%mend age;
```

In the above example, a different number of parameters can be supplied as long as there is at least one.

The call

```
%age(mylib,new,gdg_0,gdg_1,gdg_2)
```

gives a value to &syspbuff of
mylib,new,gdg_0,gdg_1,gdg_2

and

&posparm the value mylib.

Version 8 ODS (Output Delivery System)

Prepared by

destiny CORPORATION

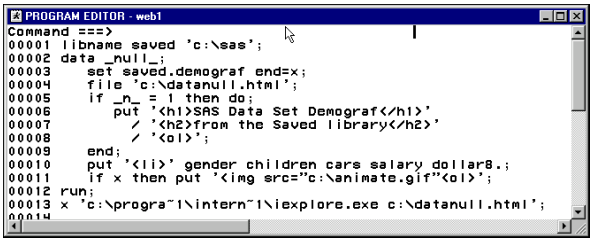
International SAS® Training and Consulting
A SAS Institute Quality Partner™

100 Great Meadow Rd, Suite 601
Wethersfield, CT 06109-2379
Phone: (860) 721-1684 1-800-7TRAINING
Fax: (860) 721-9784
Web: www.destinycorp.com Email: info@destinycorp.com

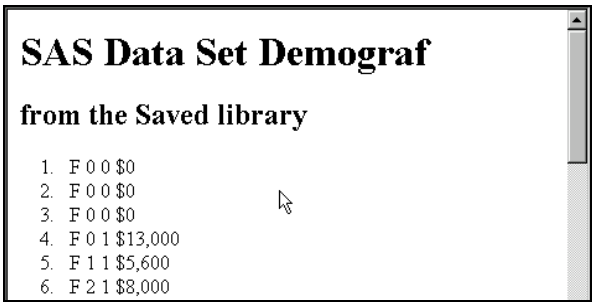
This presentation is designed to demonstrate the different possibilities we use when we build web based applications for clients and teach them the web publishing techniques we use with SAS Software. Actual examples are available on our web site.

Basic HTML Publishing

The most basic form is brute force HTML generation using the traditional data step. This tends to be the most flexible, but the most time consuming and requires complete knowledge of how HTML works. FILE and PUT statements are used extensively.



```
PROGRAM EDITOR - web1
Command ===>
00001 libname saved 'c:\sas';
00002 data _null_;
00003    set saved.demograf end=x;
00004    file 'c:\datanull.html';
00005    if _n_ = 1 then do;
00006       put '<h1>SAS Data Set Demograf</h1>'
00007         / '<h2>from the Saved library</h2>'
00008         / '<ol>';
00009    end;
00010    put '<li>' gender children cars salary dollar8.;
00011    if x then put '<img src="c:\animate.gif"<ol>';
00012 run;
00013 x 'c:\progra~1\intern~1\iexplore.exe c:\datanull.html';
00014
```



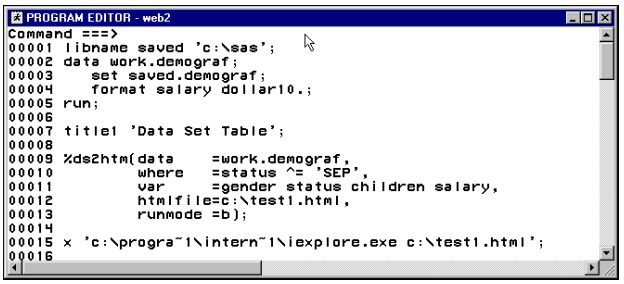
SAS Data Set Demograf

from the Saved library

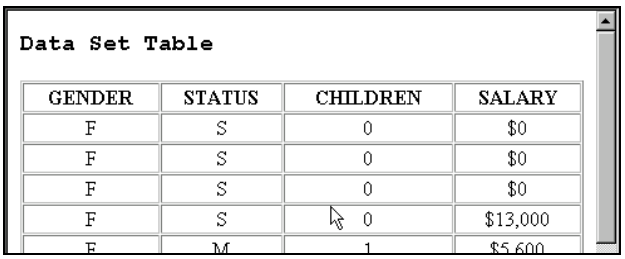
1. F 0 0 $0
2. F 0 0 $0
3. F 0 0 $0
4. F 0 1 $13,000
5. F 1 1 $5,600
6. F 2 1 $8,000

Data Set Publishing with DS2HTM

SAS offers a macro that allows one to take a SAS data file and present it in an HTML form.



```
PROGRAM EDITOR - web2
Command ===>
00001 libname saved 'c:\sas';
00002 data work.demograf;
00003    set saved.demograf;
00004    format salary dollar10.;
00005 run;
00006
00007 title1 'Data Set Table';
00008
00009 %ds2htm(data    =work.demograf,
00010         where   =status ^= 'SEP',
00011         var     =gender status children salary,
00012         htmlfile=c:\test1.html,
00013         runmode =b);
00014
00015 x 'c:\progra~1\intern~1\iexplore.exe c:\test1.html';
00016
```

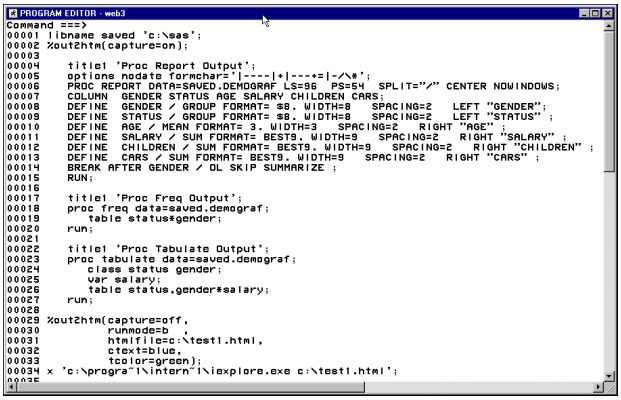


Data Set Table

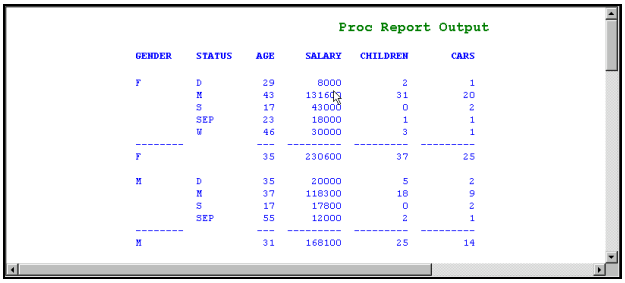
GENDER	STATUS	CHILDREN	SALARY
F	S	0	$0
F	S	0	$0
F	S	0	$0
F	S	0	$13,000
F	M	1	$5,600

SAS Output Publishing with OUT2HTM

Any procedure's output can be captures and published with the OUT2HTM macro.

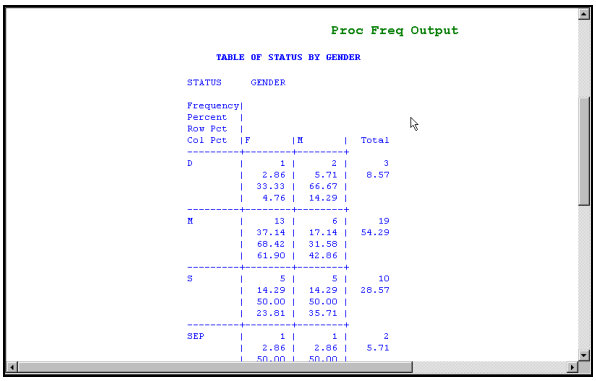


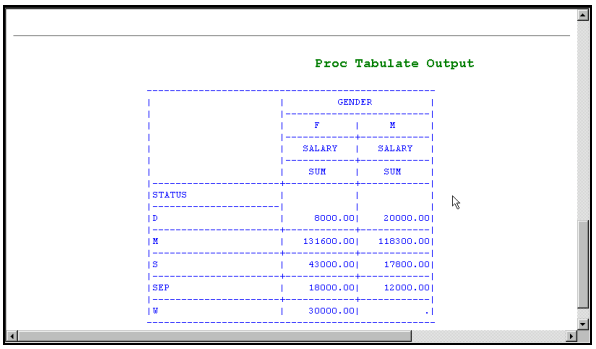
```
PROGRAM EDITOR - web3
Command ===>
00001 libname saved 'c:\sas';
00002 %out2htm(capture=on);
00003
00004    title1 'Proc Report Output';
00005    options nodate formchar='|----|+|---+=|-/\*';
00006    PROC REPORT DATA=SAVED.DEMOGRAF LS=96  PS=54  SPLIT="/" CENTER NOWINDOWS;
00007    COLUMN  GENDER STATUS AGE SALARY CHILDREN CARS;
00008    DEFINE  GENDER / GROUP FORMAT= $8. WIDTH=8   SPACING=2   LEFT "GENDER";
00009    DEFINE  STATUS / GROUP FORMAT= $8. WIDTH=8   SPACING=2   LEFT "STATUS" ;
00010    DEFINE  AGE / MEAN FORMAT= 3. WIDTH=3        SPACING=2   RIGHT "AGE" ;
00011    DEFINE  SALARY / SUM FORMAT= BEST9. WIDTH=9   SPACING=2   RIGHT "SALARY" ;
00012    DEFINE  CHILDREN / SUM FORMAT= BEST9. WIDTH=9   SPACING=2   RIGHT "CHILDREN" ;
00013    DEFINE  CARS / SUM FORMAT= BEST9. WIDTH=9   SPACING=2   RIGHT "CARS" ;
00014    BREAK AFTER GENDER / DL SKIP SUMMARIZE ;
00015    RUN;
00016
00017    title1 'Proc Freq Output';
00018    proc freq data=saved.demograf;
00019       table status*gender;
00020    run;
00021
00022    title1 'Proc Tabulate Output';
00023    proc tabulate data=saved.demograf;
00024       class status gender;
00025       var salary;
00026       table status,gender*salary;
00027    run;
00028
00029 %out2htm(capture=off,
00030          runmode=b  ,
00031          htmlfile=c:\test1.html,
00032          ctext=blue,
00033          tcolor=green);
00034 x 'c:\progra~1\intern~1\iexplore.exe c:\test1.html';
00035
```



Proc Report Output

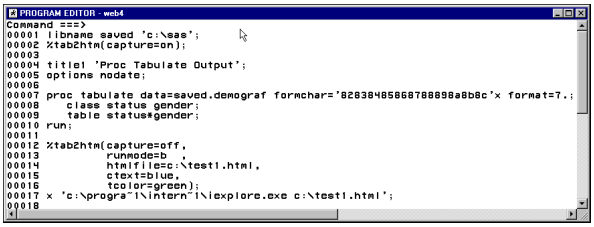
GENDER	STATUS	AGE	SALARY	CHILDREN	CARS
F	D	29	8000	2	1
	M	43	131603	31	20
	S	17	43000	0	2
	SEP	23	18000	1	1
	W	46	30000	3	1
F		35	230600	37	25
M	D	35	20000	5	2
	M	37	118300	18	9
	S	17	17800	0	2
	SEP	55	12000	2	1
M		31	168100	25	14

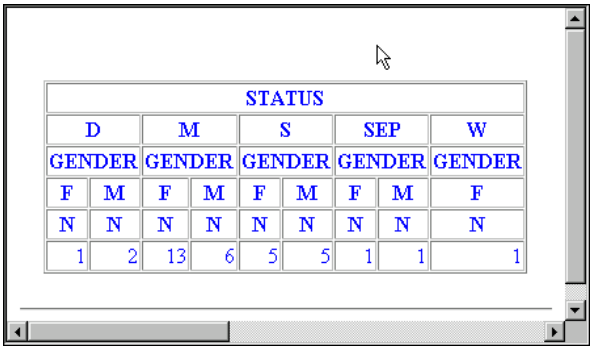




Tabulate Publishing with TAB2HTM

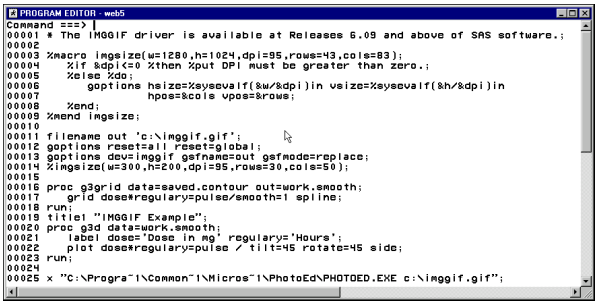
A special macro has been designed to capture the output of a PROC TABULATE and create an HTML table. This is a better presentation for a PROC TABULATE than using OUT2HTM.

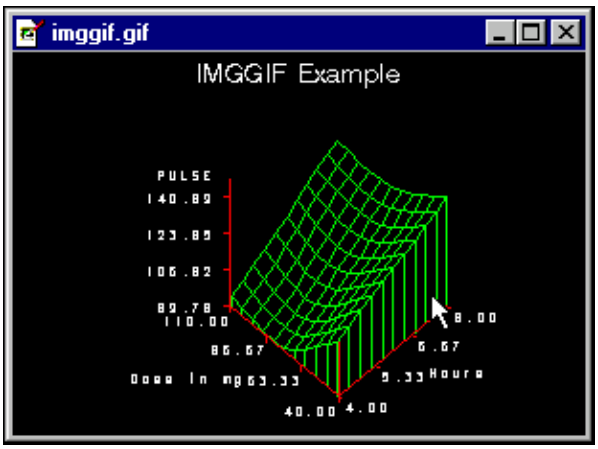




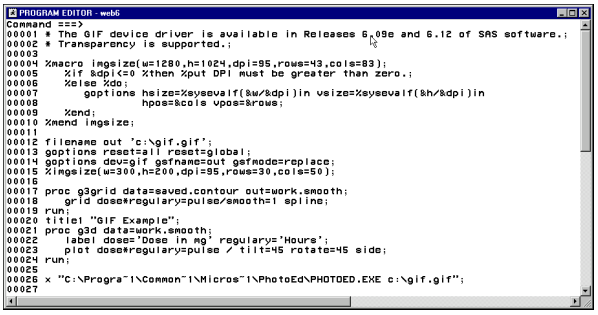
The following drivers are designed to create graphical output used on the web with SAS/Graph software.

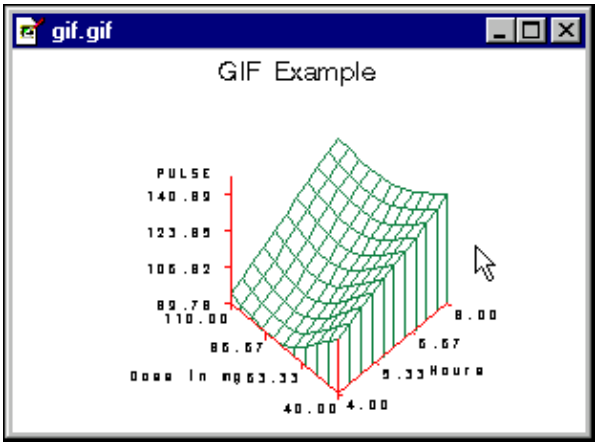
Graphical Publishing with GIFs 6.09



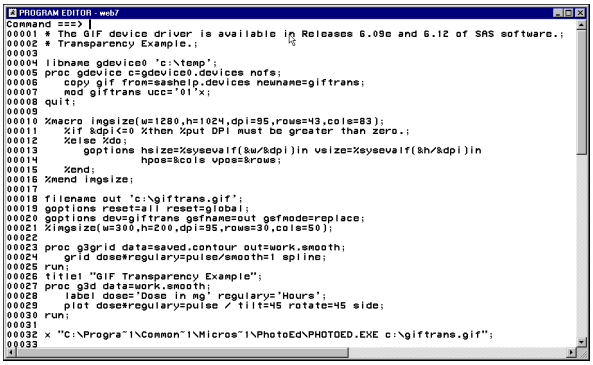


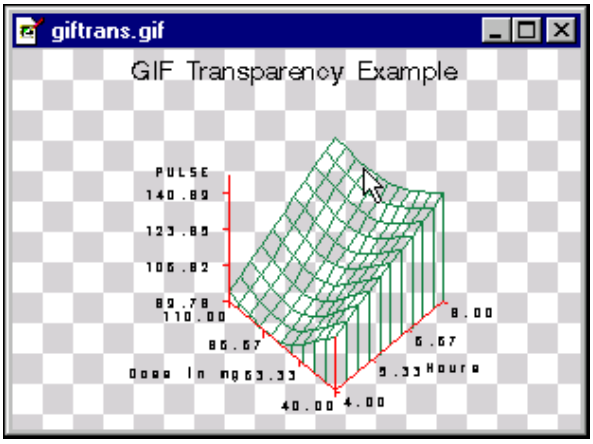
Graphical Publishing with GIFs 6.09e and 6.12



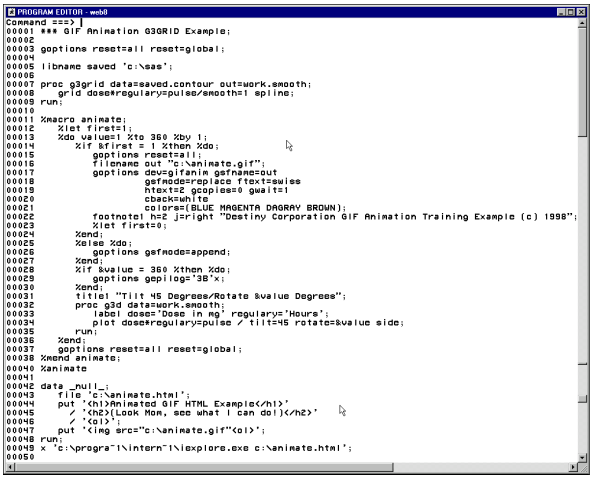


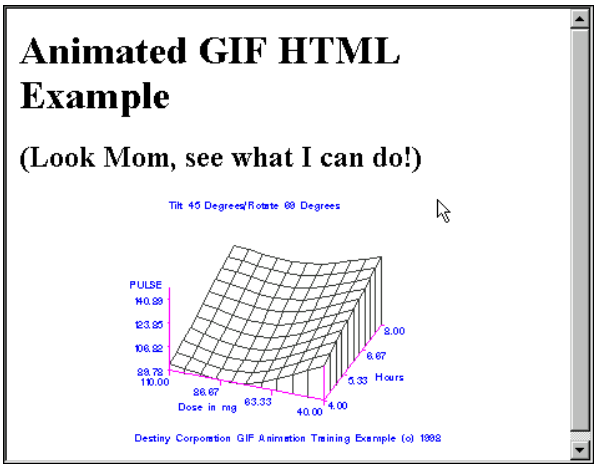
Graphical Publishing with GIFs 6.09e and 6.12 Transparency





Animation with GIFs





Version 8 and the Output Delivery System

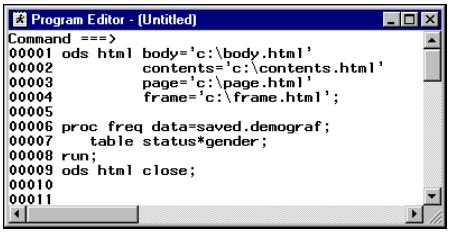
When ODS was introduced in Version 8 of SAS software, a whole new way of publishing was created.

The simplest form of output can be seen by the following example.

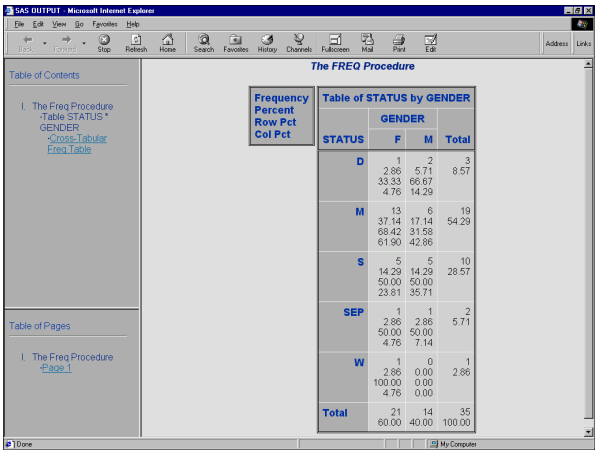
First, notice the typical style of using ODS.

1. The ODS 'capture style' utility is turned on.
2. The process executes.
3. The ODS utility is turned off.

The general ODS syntax can look like the following.

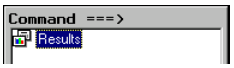


This syntax routes HTML output anywhere you desire. It creates four files. The FRAME.HTML file pulls them all together.

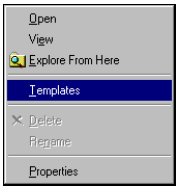


Styles in HTML

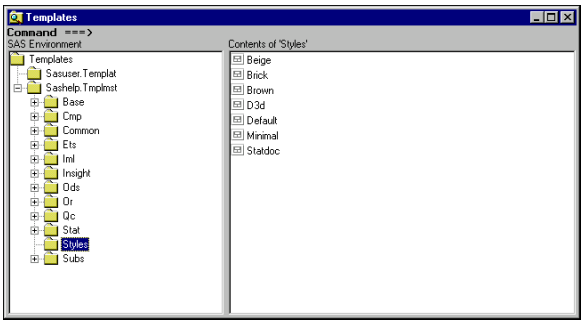
There are several HTML styles that come with SAS. They are designed to allow the user to create an HTML standard of colors, fonts and more across all HTML output. Any of the default ones can be used, or they can be modified to suit your needs. To see the existing styles, go to the Results window and select.



Right click and select Templates.



Select SASHELP.TMPLMST, Styles to see the available styles.



To examine a style, execute Proc Template with the Style name. Default is the default style used.

```
Program Editor - (Untitled)
Command ===>
00001 proc template;
00002    source styles.default;
00003 run;
00004
```

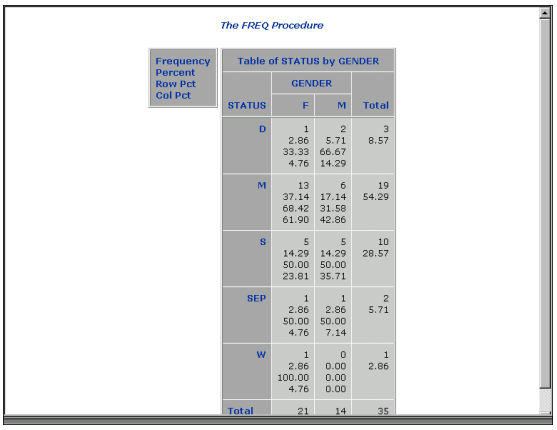
```
Command ===>
254  proc template;
255     source styles.default;
define style Styles.Default;
   style fonts /
      'TitleFont2' = ('Arial, Helvetica, Helv',4,Bold Italic)
      'TitleFont' = ('Arial, Helvetica, Helv',5,Bold Italic)
      'StrongFont' = ('Arial, Helvetica, Helv',4,Bold)
      'EmphasisFont' = ('Arial, Helvetica, Helv',3,Italic)
      'FixedEmphasisFont' = (Courier,3,Italic)
      'FixedStrongFont' = (Courier,4,Bold)
      'FixedHeadingFont' = (Courier,4)
      'FixedFont' = (Courier,4)
      'headingEmphasisFont' = ('Arial, Helvetica, Helv',4,Bold Italic)
      'headingFont' = ('Arial, Helvetica, Helv',4,Bold)
      'docFont' = ('Arial, Helvetica, Helv',3);
   style color_list /
      'fgB2' = cx0066AA
      'fgB1' = cx004488
      'fgA4' = cxAAFFAA
      'bgA4' = cx880000
      'bgA3' = cxD3D3D3
      'fgA2' = cx0033AA
      'bgA2' = cxB0B0B0
      'fgA1' = cx000000
      'bgA1' = cxF0F0F0
      'fgA' = cx002288
      'bgA' = cxE0E0E0;
   style colors /
      'headerfgemph' = color_list('fgA2')
      'headerbgemph' = color_list('bgA2')
      'headerfgstrong' = color_list('fgA2')
      'headerbgstrong' = color_list('bgA2')
```

Cut and paste the code for modification. Rerun it with a new name and your own style can be created.

To select a style, use the ODS statement

```
Program Editor - (Untitled)
Command ===>
00001 ods html body='c:\body.html' style=styles.statdoc;
00002 proc freq data=saved.demograf;
00003    table status*gender;
00004 run;
```

It produces the following HTML. (Looks different than styles.default)



The FREQ Procedure

Table of STATUS by GENDER

Frequency / Percent / Row Pct / Col Pct	GENDER F	GENDER M	Total
D	1 2.86 33.33 4.76	2 5.71 66.67 14.29	3 8.57
M	13 37.14 68.42 61.90	6 17.14 31.58 42.86	19 54.29
S	5 14.29 50.00 23.81	5 14.29 50.00 35.71	10 28.57
SEP	1 2.86 50.00 4.76	1 2.86 50.00 7.14	2 5.71
W	1 2.86 100.00 4.76	0 0.00 0.00 0.00	1 2.86
Total	21	14	35

See the online help to learn more about changing styles and what the values mean.

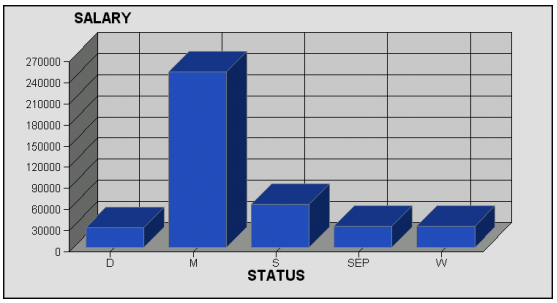
Graphical Support

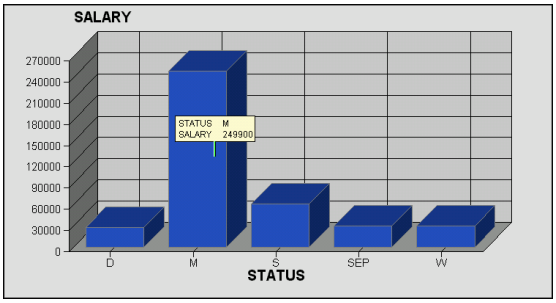
There are several other drivers available for outputting graphical objects. Consider the following.

Java Support

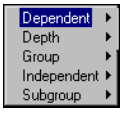
There is Java support. Consider the following driver.

```
Program Editor - java
Command ===>
00001 libname saved 'd:\sas\data8';
00002 filename odsout 'c:\';
00003 ods listing close;
00004 goptions reset=global reset=all;
00005 goptions device=java transparency;
00006 ods html body='demograf.html' path=odsout;
00007 title1 'Marital Status';
00008 proc gchart data=saved.demograf;
00009    vbar3d status / sumvar=salary;
00010 run;
00011 quit;
00012 ods html close;
```





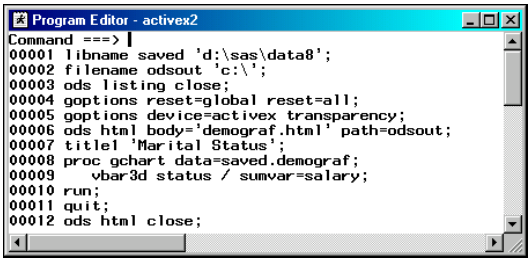
There are several menus available from the right click of a mouse.

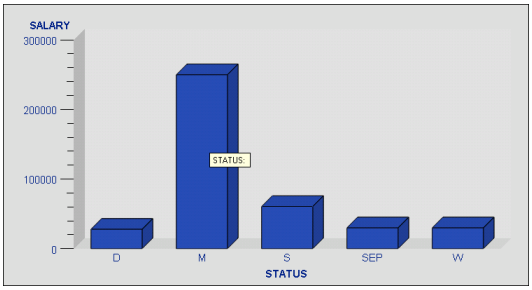


Experiment with each to understand the capabilities.

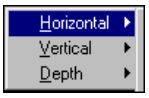
Active X Support

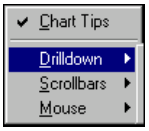
SAS now supports Active X controls. Consider the following code with the Active X Driver.





A right click yields many possibilities.





Consider testing out the different options.

Using Styles in Procedures

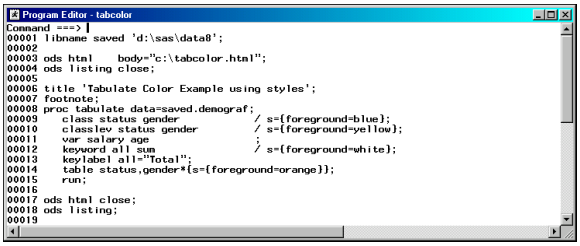
Style options are available in SAS. Their syntax is added to procedures when formatting them for output. This is available on the procedure level in the following way. Use the syntax

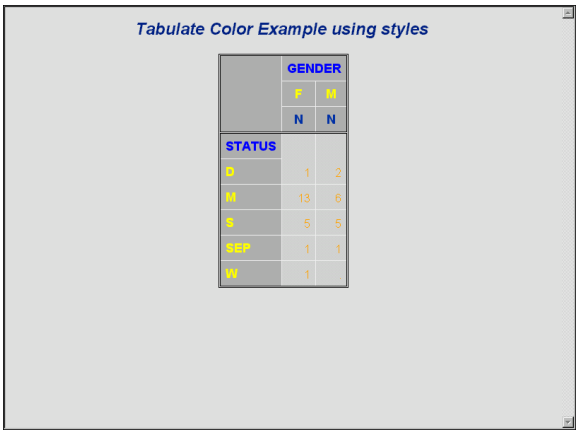
S={foreground=blue}

At various locations to specify colors.

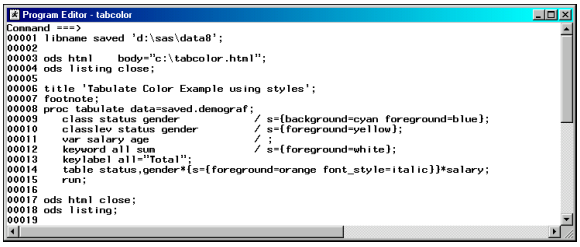
Current supported Procedures are Tabulate and Report.

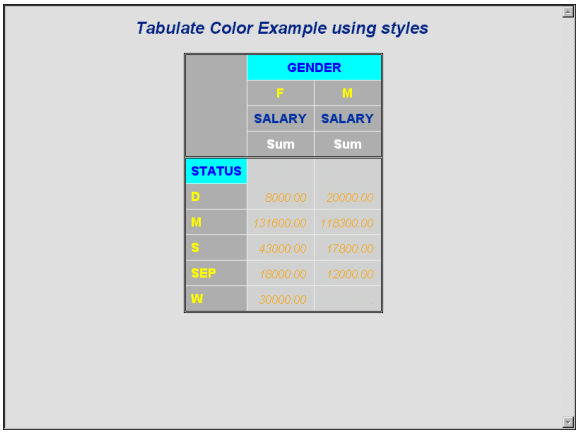
Styles in Proc Tabulate



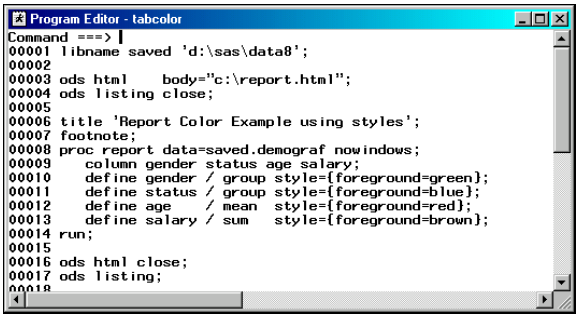


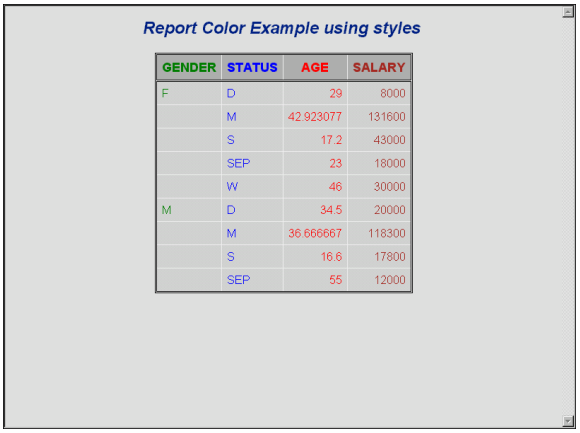
Additional examples use background and font_style.





Styles in Proc Report





Traffic Lighting

This is a technique that employs styles with formatted values to allow changing the color and presentation of cells as the value changes. Examine the following code.

Program Editor - traffic

```
Command ===>
00001 ods html    body="c:\traffic";
00002 ods listing close;
00003
00004 title 'Traffic Lighting Example';
00005 proc format;
00006    value agefmt
00007    1-18  = 'cx4DAE6E'
00008    19-30 = 'Blue'
00009    31-45 = 'Green'
00010    46-65 = 'Red'
00011    other = 'Gray';
00012 run;
00013
00014 proc tabulate data=saved.demogius style={foreground=agefmt.};
00015    class status gender    / s={foreground=black};
00016    classlev status gender / s={foreground=red};
00017    var age ;
00018    table status,gender*age*mean;
00019 run;
00020
00021 ods html close;
00022 ods listing;
```

Traffic Lighting Example

	GENDER	
	F	M
	AGE	AGE
	Mean	Mean
STATUS		
D	26.00	40.75
M	41.90	43.68
P	44.00	26.00
S	23.67	28.13
W	47.00	.

Hyperlinks in Graphs and Reports

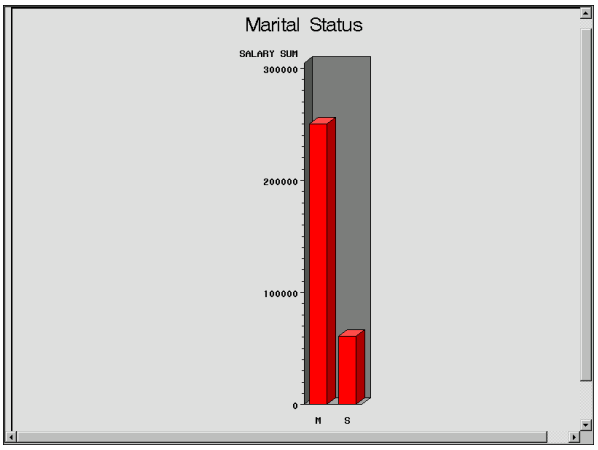
Hyperlinks in Graphs

In Version 8 of SAS Software, HTML examples that link to each other are possible. Wouldn't it be great to create a graph and then be able to click on the items in the graph to branch to appropriate detail about those items?

The following code could create that result.

Program Editor - graphhref

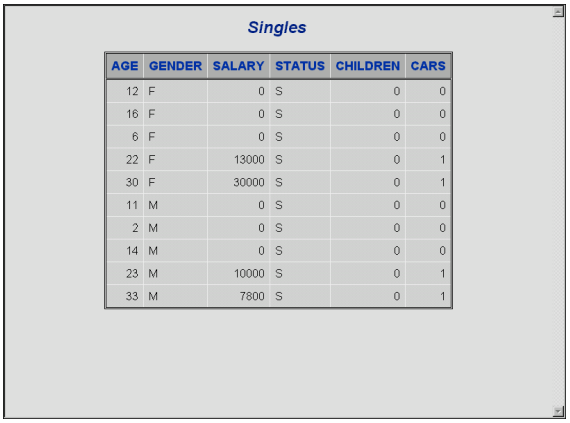
```
Command ===>
00001 libname saved 'd:\sas\data8';
00002 filename odsout 'c:\';
00003 ods listing close;
00004 goptions reset=global;
00005 data work.demograf;
00006    length linkme $ 40;
00007    set saved.demograf;
00008       if status='S' then linkme = 'href="single.html"';
00009    else if status='M' then linkme = 'href="married.html"';
00010 run;
00011 goptions device=gif transparency;
00012 ods html body='demograf.html' path=odsout;
00013 title1 'Marital Status';
00014 proc gchart data=work.demograf;
00015    vbar3d status / sumvar=salary
00016    html=linkme;
00017    where status in ('S','M');
00018 run;
00019 quit;
00020
00021 ods html body='single.html' path=odsout;
00022 title1 'Singles';
00023 proc print data=work.demograf(drop=linkme) noobs;
00024    where status='S';
00025 run;
00026
00027 ods html body='married.html' path=odsout;
00028 title1 'Married';
00029 proc print data=work.demograf(drop=linkme) noobs;
00030    where status='M';
00031 run;
00032 ods html close;
00033 ods listing;
00034
```



Click on the appropriate bars to see the detail.

Married

AGE	GENDER	SALARY	STATUS	CHILDREN	CARS
52	F	15000	M	5	2
28	F	15000	M	3	1
23	F	0	M	3	2
56	F	30000	M	3	2
54	F	0	M	3	2
60	F	13000	M	3	2
32	F	0	M	2	2
34	F	18000	M	2	1
34	F	0	M	2	1
38	F	10000	M	2	1
65	F	10000	M	2	1
26	F	5600	M	1	1
56	F	15000	M	0	2
48	M	8000	M	5	2
34	M	40000	M	4	3



Singles

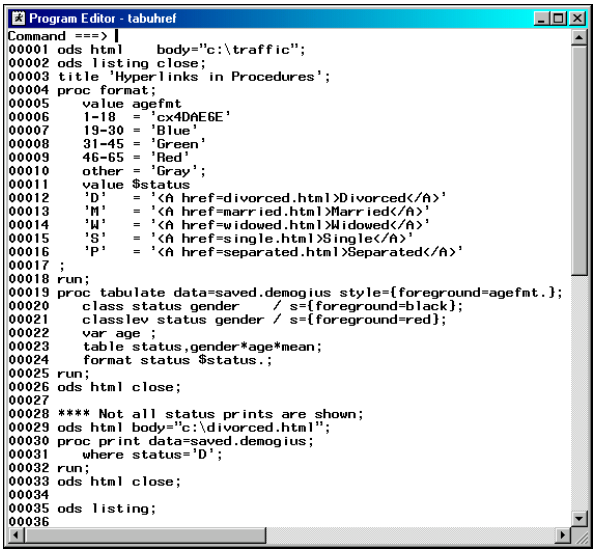
AGE	GENDER	SALARY	STATUS	CHILDREN	CARS
12	F	0	S	0	0
16	F	0	S	0	0
6	F	0	S	0	0
22	F	13000	S	0	1
30	F	30000	S	0	1
11	M	0	S	0	0
2	M	0	S	0	0
14	M	0	S	0	0
23	M	10000	S	0	1
33	M	7800	S	0	1

The key items to get this to work are the following:

1. Use the GIF driver for graph output.
2. Create a variable attached to the grouped data item that includes a valid hyperlink. Here we use LINKME.
3. Use the HTML= option on the charting statement.

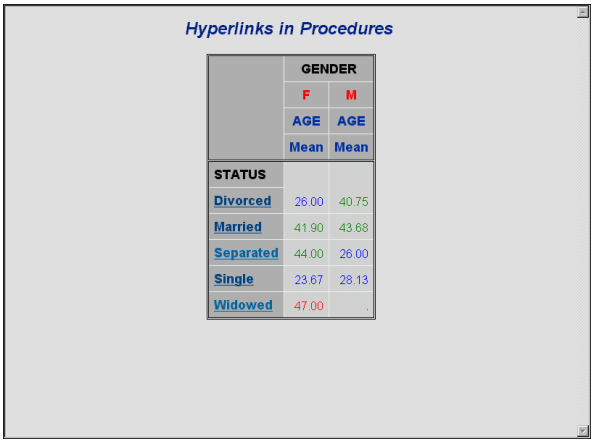
Hyperlinks in Reports

This is an easy task by formatting the values with predefined hyper links. Examine the following code:



Program Editor - tabuhref

```
Command ===>
00001 ods html    body="c:\traffic";
00002 ods listing close;
00003 title 'Hyperlinks in Procedures';
00004 proc format;
00005    value agefmt
00006    1-18  = 'cx4DAE6E'
00007    19-30 = 'Blue'
00008    31-45 = 'Green'
00009    46-65 = 'Red'
00010    other = 'Gray';
00011    value $status
00012    'D'   = '<A href=divorced.html>Divorced</A>'
00013    'M'   = '<A href=married.html>Married</A>'
00014    'W'   = '<A href=widowed.html>Widowed</A>'
00015    'S'   = '<A href=single.html>Single</A>'
00016    'P'   = '<A href=separated.html>Separated</A>'
00017 ;
00018 run;
00019 proc tabulate data=saved.demogius style={foreground=agefmt.};
00020    class status gender    / s={foreground=black};
00021    classlev status gender / s={foreground=red};
00022    var age ;
00023    table status,gender*age*mean;
00024    format status $status.;
00025 run;
00026 ods html close;
00027
00028 **** Not all status prints are shown;
00029 ods html body="c:\divorced.html";
00030 proc print data=saved.demogius;
00031    where status='D';
00032 run;
00033 ods html close;
00034
00035 ods listing;
00036
```



Hyperlinks in Procedures

	GENDER	
	F	M
	AGE	AGE
	Mean	Mean
STATUS		
Divorced	26.00	40.75
Married	41.90	43.68
Separated	44.00	26.00
Single	23.67	28.13
Widowed	47.00	.

SQL Processing

Prepared by

International SAS® Training and Consulting

Destiny Corporation – 100 Great Meadow Rd Suite 601 - Wethersfield, CT 06109-2355
Phone: (860) 721-1684 - 1-800-7TRAINING - Fax: (860) 721-9784
Web: www.destinycorp.com
Email: info@destinycorp.com

SQL Statements

PROC SQL consists of TEN statements:

1.SELECT	to perform queries
2.VALIDATE	to validate the syntax of a query
3.DESCRIBE	to show how a view has been defined
4.CREATE	to create a table (SAS data set), index or view
5.DROP	to delete a table, index or view
6.UPDATE	change the values in a table
7.INSERT	add rows to a table
8.DELETE	to delete rows from a table
9.ALTER	to add, delete or modify columns in a table
10.RESET	add, change or deletion of options

Of these, by far the most important is the **SELECT** statement.

The Select Statement

PROGRAM EDITOR

```
*Q02E01  The Select statement;
proc sql;
  select *
  from saved.computer;
```

** The Proc SQL statement is loaded into memory and remains resident until another Data or Proc step is run or a QUIT; statement is executed. Therefore, subsequent queries or other SQL statements can be run without the need to re-submit the Proc SQL statement.

** each SQL statement is processed individually. No RUN; is required.

** Proc Print style output is automatically produced. Note there is no observation number and there is a line below the variable names. This output can be suppressed with the NOPRINT option on the Proc SQL statement:

```
proc sql noprint;
select....
```

Usage

Invoke Proc SQL and then use a **SELECT** statement:

PROGRAM EDITOR

```
proc sql;
```

PROGRAM EDITOR

```
  select *
  from saved.computer;
```

PROGRAM EDITOR

```
quit;
```

Note that while Proc SQL is running, the RUN; statement itself simply produces an interesting note on the Log:

Basic Examples

The SQL procedure uses the SELECT statement to perform a wide variety of queries. Within the SELECT statement are different Clauses:

Order by Character Variable

Select input (Char 3.) during order by during;

Ordering the Result

The order by clause determines the order in which the rows are displayed in the resulting table:

PROGRAM EDITOR

```
*Q02E02  The Order By clause to determine order for rows;
  select *
  from saved.computer
  order by disk;
```

This code produces the same result as:

PROGRAM EDITOR

```
proc sort data = saved.computer;
  by disk;
proc print data = saved.computer;
run;
```

Advantages of SQL:

** No physical sort has to be performed

Notice that no physical output file has been produced from this query, simply a listing as output.

Selecting Rows

Selecting Rows is done with the *WHERE* clause:

PROGRAM EDITOR

```
*Q02E03  The subsetting rows with where;
proc sql;
   select *
   from saved computer
   where supplier = 'KETCHUP COMPUTERS'
   order by disk;
quit;
```

The WHERE clause syntax:

(These variants also apply to the *Having* clause. **Notes:**

** ¬= is normally the symbol used for Not Equals on an IBM mainframe.

** ^= is normally the symbol used for Not Equals on ASCII based machines.

** NOT= and NE can also be used for Not Equals.

** The 'Sounds Like' operator uses the Soundex algorithm. This is normally used to search for variants in names, for example telephone directory applications, but is not perfect.

** The BETWEEN operator includes values defined in the range.

** IS NULL and IS MISSING are equivalent and match with all missing values including special missing values.

** Computed columns cannot be referred to by name in a WHERE clause. They have to be recalculated for the WHERE clause.

** SAS functions are supported except for LAG, DIF and SOUND. These 'ordinary SAS' functions should not be confused with SQL summary functions seen in Q2.3.

Controls and Enhancements

Selecting columns

Using the column names from the SAS file allows choice of the columns in the report:

PROGRAM

```
*Q02E04  Selecting columns to display in select statement;
proc sql;
   select type,disk,retail
   from saved.computer
   where supplier = 'KETCHUP COMPUTERS'
   order by disk;
```

This is the equivalent of:

PROGRAM EDITOR

```
proc sort data = saved.computer
   by disk;
run;
proc print data=saved.computer
   (where=(supplier='KETCHUP COMPUTERS'));
   var type disk retail;
run;
```

Again, the SQL advantage is the lack of the sort step.

Note that the list of column names require to be delimited by commas:

select type, disk, retail

All lists of column names and table names require commas in SQL syntax.

Calculating Columns

Columns can be calculated by assigning an expression to an item name:

PROGRAM EDITOR

```
*Q02E05 Assigning an expression to column name;
proc sql;
   select type,
      disk,
      retail,
      retail * 7/47 as vat
   from saved. Computer
   where supplier = 'KETCHUP COMPUTERS
   order by disk;
quit;
```

Notice the structure: expression AS variable

as opposed to the traditional SAS: variable = expression;

Notice that the traditional equivalent would now require an additional step:

PROGRAM EDITOR

```
data new;
   set saved.computer;
   vat = retail * 7/47;
proc sort data = new;
   by disk;
proc print data = new
   (where=(supplier='KETCHUP COMPUTERS'));
   var type disk retail vat;
run;
```

Formatting Values - A Column Modifier

The format option appears after the column name, not as a separate statement:

PROGRAM EDITOR

```
*Q02E06  Formatting values with a column modifier;

  title 'Ketchup Computers, VAT element of prices';
  select type,
    disk,
    retail,
    retail*7/47 as vat format=7.2
  from saved.computer
  where supplier='KETCHUP COMPUTERS'
  order by disk ;
```

This is a SAS enhancement to a standard SQL.

Labeling Columns - a Column Modifier

Column headings can be changed with the LABEL option:

PROGRAM EDITOR

```
*Q02E07  Labeling columns with a column modifier;

proc format;
  picture pound low-high='000,000,009.99'(prefix='$');
proc sql;
  title 'Ketchup Computers, VAT element of prices';
  select type   label='Computer Type',
    retail label='Retail Price' format=pound9.2,
    retail*7/47 as vat format=pound7.2
  from saved.computer
  where supplier='KETCHUP COMPUTERS'
  order by disk ;
```

This is a SAS enhancement to standard SQL.

Using Functions

Ordinary 'Data Step' functions can be used in these expressions as noted in Q2.1:

PROGRAM EDITOR

```
*Q02E08   Using functions;

options nodate nonumber;
title 'Charges raised for car hire';
proc format;
  picture pound low-high = '000,009.99' (prefix= '$');
  value $model 'F' = 'Fiesta'
               'E' = 'Escort'
               'O' = 'Orion'
               'S' = 'Sierra'
               'G' = 'Granada';
proc sql;
  select substr(carkey,2,1) format=$model.,
       custkey label='Customer Code',
       daychg label='Daily Charge',
       (endate - stdate +1) as duration
            label= 'Days Hired',
       (endate - stdate +1)*daychg
            format = pound7.2,
       round(((endate - stdate +1)*daychg*7/47),0.01)
            as vat format = pound7.2
  from saved.carhire
  order by 5 desc;
```

Notes:

Functions can be used to derive calculated columns as shown by the SUBSTR function above and can be used to change calculate values as shown by the ROUND function. Derived columns need not be given aliases using the AS syntax.

Aliases cannot be used in further calculations. For example, if the expression in line 17 above:

(endate - stdate +1)*daychg
were replaced by
duration*daychg
an error would result, duration not being found.

The order by clause can use the ordinal position of the column. In the example above the 5th column in used. Ordering can be done a calculated column with no alias.

Distinct Values

A useful keyword is *DISTINCT,* which allows selection of unique values of a column:

PROGRAM EDITOR

```
*Q02E09  Selection of unique key values with Distinct;

  title 'Which disk types are sold by Ketchup?';
  select distinct disk
  from saved.computer
  where supplier='KETCHUP COMPUTERS'
  order by disk ;                    <-- not required
```

OUTPUT

```
Which disk types are sold by Ketchup?

DISK
  20
  40
  60
 100
 120
 200
```

Traditional SAS programming would involve the following:

PROGRAM EDITOR

```
title 'Which disk types are sold by Ketchup?';
proc sort data=saved.computer out=sorted;      <-----
required
  by disk;
data unique(keep=disk);
  set sorted(keep=disk supplier);
  by disk;
  if last.disk;
  where supplier='KETCHUP COMPUTERS';
proc print data=unique;
run;
```

Selecting data with DISTINCT

The *DISTINCT* keyword can be used to select data for all combinations of columns:

PROGRAM EDITOR

```
*Q02E10  Distinct can apply to unique combinations;

  title 'All combinations of disk and type';
  select distinct disk label='Hard Disk Size',
       type label='Computer Type'
  from saved.computer
  where supplier='KETCHUP COMPUTERS'
  order by disk ;
```

The DISTINCT keyword applies to all the column names and each unique combination of values is returned.

Data/Proc Step methods would use two BY variables. Note the effect on the inner variable *(type*) when the outer variable *(disk*) changes value:

PROGRAM EDITOR

```
title 'Which disk types are sold by Ketchup?';
proc sort data=saved.computer out=sorted;
  by disk type;
data unique(keep=disk type);
  set sorted(keep=disk type supplier);
  by disk type;
  if last.type;
  where supplier='KETCHUP COMPUTERS';
proc print data=unique;
run;
```

Syntax Checking

Use the VALIDATE statement, before the SELECT statement, to check the SQL statements without executing them:

PROGRAM EDITOR

```
*Q02E11  Checking select statement syntax without executing;

  validate
  select distinct disk label = 'Hard Disk Size'.
                  Type label = 'Computer Type'
  from saved computer
  where supplier = 'KETCHUP COMPUTERS'
  order by disk ;
```

LOG

```
Proc SQL has valid syntax
```

This facility can only be used with a *Query*-expression i.e. to qualify the syntax of a SELECT.

Syntax Errors

This syntax error is caused by the ORDER BY option:

PROGRAM EDITOR

```
*Q02E12   Where is the syntax error here?;

  validate
  select distinct disk label = 'Hard Disk Size',
     type label = 'Computer Type'
  from saved.Computer
  order by disk
  where supplier = 'KETCHUP COMPUTERS'  ;
```

LOG

```
  validate
  select distinct disk label = 'Hard Disk Size' ,
                  type label = 'Computer Type'
  from saved. Computer
  order by disk
  where supplier = 'KETCHUP COMPUTERS'  ;
--------
22
202
ERROR 22-322:     Expecting one of the following: (, **, *, /, +, -, !!,
||,
          <,<=,<>,=,>,>=,?,CONTAINS,
                          eq, ge, gt, le, lt, ne, ^=, ~=,
                          &, AND, !, or, ¦, OR, ¦, ',
 The statement is being ignored.

ERROR 202-322:    The option or parameter is not recognized.
Make sure the ORDER by statement is the last option on the
SELECT statement.
```

Analysis on Groups

Summary Functions

A series of functions are provided to work 'down the columns'. A complete list of these functions is given in Q2.5.

PROGRAM EDITOR

```
*Q02E13  Analysis down a column for groups;

  select mean(retail) as avprice
  from saved.computer;
```

OUTPUT

```
                    AVPRICE

                    1929.167
```

This is the equivalent of:

PROGRAM EDITOR

```
proc means data = saved.computer mean;
  var retail;
run;
```

With more than one argument, the function performs for each row:

PROGRAM EDITOR

```
*Q02E14  More then one argument to analyze each row;

  select retail format= pound10.2,
    retail * 7/47 as VAT format = pound8.2,
    sum(retail,retail*7/47) as gross
                format =pound10.2
  from saved.computer;
```

With a single argument, but with other selected columns, the function gives a result for all the rows, then merges the summary back with each row:

PROGRAM EDITOR

```
*Q02E15  Merges summary value onto each row of output;

  select cpu,
       disk,
       (retail -wholesal)
            as profit label='Profit',
       mean(retail-wholesale)
            as avprofit label = 'Average Profit',
       (retail-wholesal) - mean(retail -wholesal)
            as diff label = 'Difference'
  from saved.computer
  where supplier contains 'FLOPPY';
```

To accomplish the same thing in Data/Proc step either requires use of Proc Means/Summary to create a one-observation, one-variable data set which is then read into the data step alongside saved.computer or two passes of the data in the same data step:

PROGRAM EDITOR

```
data new;
  retain avprofit;
  if _n_ = 1 then do;
    do until(finish);
      set saved.computer end = finish
                  nobs = numobs;
      profit=retail-wholesal;
      totprof+profit;
    end;
    avprofit = totprof / numobs;
  end;
  set saved.computer;
  profit = retail - wholesal;
  diff = (retail - wholesal) - avprofit;
run;
proc print data=new;
  var cpu disk  profit avprofit diff;
  label profit='Profit'
  avprofit='Average Profit'
  diff = 'Difference';
 run;
```

An important function is *COUNT(*)* which gives the number of rows:

PROGRAM EDITOR

```
*Q02E16  The count function supplies the number of rows;

  select count(*) as no_rows
  from saved.computer;
  select sum(retail)/count(*) as average
  from saved.computer;
quit;
```

Running SAS® Applications on the Web
Prepared by Destiny Corporation

Starting The Web Server

A Web Server is required to execute SAS/Intrnet software. It must be up and running for a web browser and SAS to be able to interact.

A Web Server and SAS can run under any supported platform, including Unix, Windows NT, etc…

For the purposes of the following demonstrations, we will use a Personal Web Server available through Apache. (Apache for Windows 1.3.4) See www.apache.org for more details.

Configuring A Local Web Server

We must determine the Internet Protocol Configurations available and choose the appropriate one as our IP address.

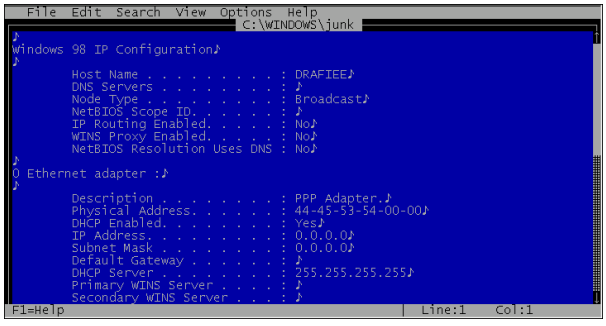
To do this, type IPCONFIG /ALL > junk at the command line.

```
C:\WINDOWS>ipconfig /all > junk_
```

Edit the junk. file.

```
C:\WINDOWS>edit junk.
```

Determine the IP address to use.



We will use IP address 0.0.0.0

The HOSTS. file under the C:\WINDOWS subdirectory must list the IP address, along with the servername, so it can be found easily by the local browser.

Change the HOSTS. file under C:\WINDOWS to read:

Loading The Web Server

We load the Web Server with the following command:

```
C:\Program Files\Apache Group\Apache>apache -C "ServerName drafiee"
```

The name of the server is ***drafiee***.

The following message appears.

```
Apache/1.3.4 (Win32) running...
```

Minimize the window.

Loading the Application Dispatcher and Broker

The Application Dispatcher is an integral part of executing SAS/Intrnet Software. It:

- Is a Gateway that connects Web browsers with SAS software
- Allows for executing SAS programs from a Web browser.

Note: Installation of SAS/Intrnet Software is required prior to loading it.

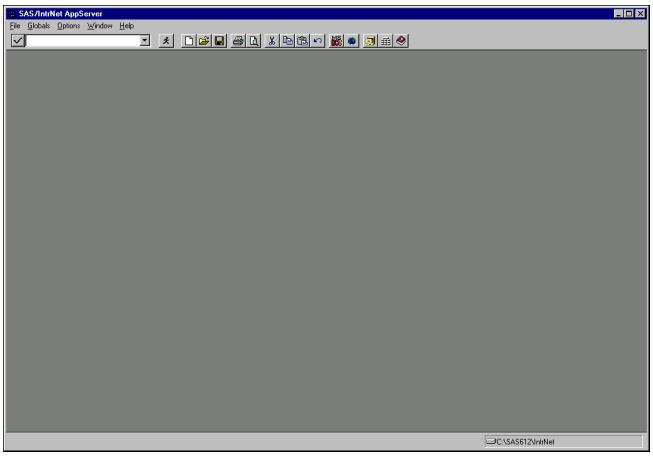
To load it, simply double click on the appropriate Icon:

Its properties are as follows:

```
C:\SAS612\sas.exe -config
C:\SAS612\config.sas -dmsbatch -
initcmd "af c=sashelp.web.appstart.scl
port=5001 srvroot='C:\SAS612\IntrNet'"
-altlog C:\SAS612\IntrNet\appsrv.log -
splashlocation
C:\SAS612\IntrNet\splash.bmp -awstitle
'SAS/IntrNet AppServer'
```

Double clicking produced the following splash screen:

And the following non-interactive SAS session.



Minimize it.

Testing the Broker

Now, we can test the broker.exe file.

The following SAS supplied files must live in the *cgi-bin* subdirectory of the Web Server.

- BROKER.EXE
- BROKER.CFG

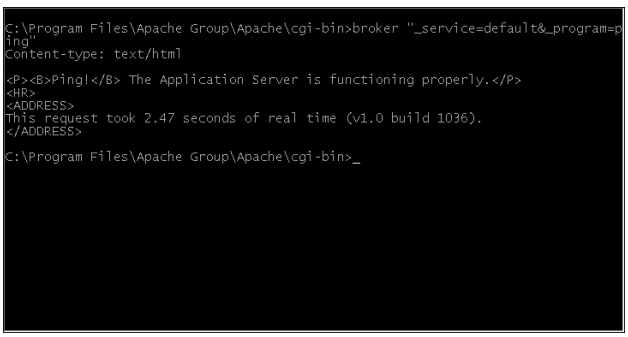
On this demonstration machine, these files live in:

C:\Program Files\Apache Group\Apache\cgi-bin

Test them with the following commands at the command prompt:

```
broker
"_service=default&_program=ping"
```

The following message should appear.



For more debugging, specify:

```
broker
"_service=default&_program=ping&_debug
=2305" > junk.
```

Returns the following in the junk. file.

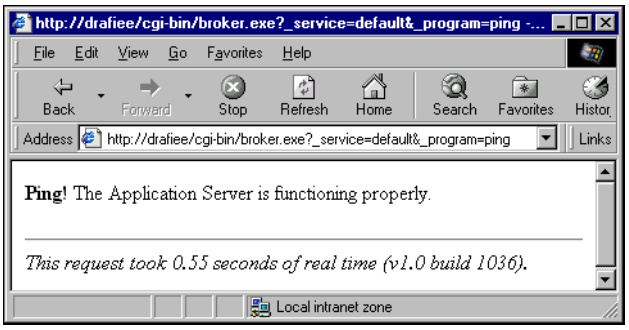
```
Content-type: text/html

<H2>Symbols passed to SAS</H2>
<PRE>
#symbols: 13
   "_RMTHOST" = ""
   "_RMTADDR" = ""
   "_RMTUSER" = ""
   "_HTCOOK" = ""
   "_HTUA" = ""
   "_HTREFER" = ""
   "_service" = "default"
   "_program" = "ping"
   "_debug" = "2305"
   "_VERSION" = "1.0"
   "_URL"     =     "http://drafiee/cgi-
bin/broker.exe"
   "_ADMIN" = "Dana Rafiee"
   "_ADMAIL"                              =
"drafiee@destinycorp.com"
</PRE>
Using timeout: 60
<BR>
rcv gethostname...ok
rcv gethostbyname...ok
rcv socket...ok
rcv bind...ok
rcv getsockname...ok (1041)
rcv listen...ok
Trying...drafiee:5001 (1 of 1)
gethost...ok
socket...ok (56)
bind...ok
connect...ok
write...ok
shutdown...ok
accept...ok
select...ok (1)
   recv...24
select...ok (1)
   recv...69
Content-type: text/html

<P><B>Ping!</B> The Application Server
is functioning properly.</P>
select...ok (1)
   recv...0
accept...ok
select...ok (1)
   recv...129
select...ok (1)
   recv...0
```

Notice the valid **URL** value.

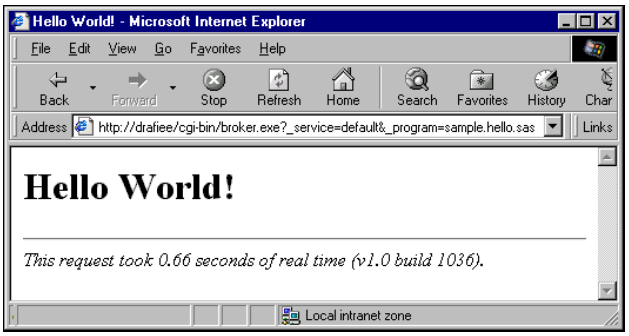
Testing the Browser

Load the browser. For this demonstration, we will use Microsoft Internet Explorer version 4.

Specify something similar to the following in the web address line to PING the system and the following should be returned.

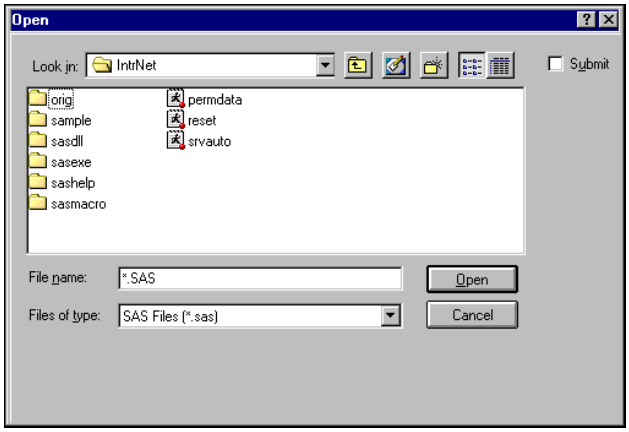


To run a sample program supplied with SAS Software, type:



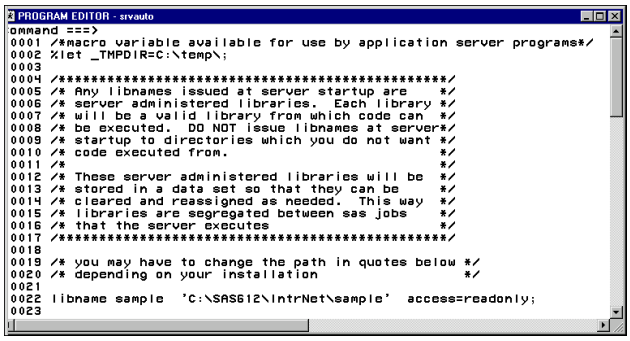
Sample Programs

Several Sample programs come with SAS/Intrnet Software. They are typically located under the *sas\intrnet* subdirectory.



Server Auto File

This file is designed to reference locations for SAS programs. It automatically loads when the Application Dispatcher loads.



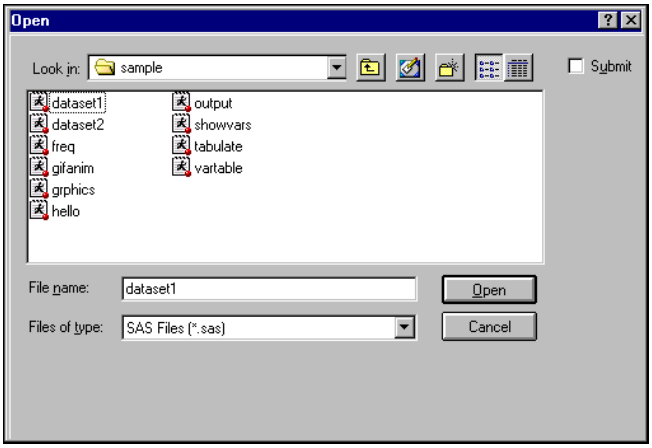
It can be tailored to fit your needs. See the reference to *sample*. This is why the call for the SAS program has a three level name.

sample.hello.sas

means

libref.programname.filetype

The following is a listing of the sample programs that are shipped with SAS/Intrnet software.



Let's open each one and examine the design. Features displayed here will be discussed in upcoming modules.

Prepared by:

Destiny Corporation
100 Great Meadow Rd., Suite 601
Wethersfield, CT 06109-2355
Phone: (860) 721-1684 1-800-7TRAINING
Fax: (860) 721-9784
Web: WWW.DESTINYCORP.COM
Email: info@destinycorp.com

Creating Java Based Applications

Prepared by

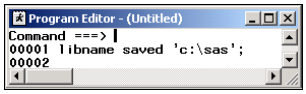
International SAS® Training and Consulting
A SAS Institute Quality Partner™

100 Great Meadow Rd, Suite 601
Wethersfield, CT 06109-2379
Phone: (860) 721-1684 1-800-7TRAINING
Fax: (860) 721-9784
Web: www.destinycorp.com Email: info@destinycorp.com

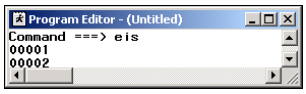
Introduction

This presentation is designed to demonstrate the use of Multi Dimensional Databases in SAS Software and their use in a business environment. We will demonstrate their creation to enablement in a webEIS application. This presentation is designed for intermediate to advanced SAS users with some background in web decision support.

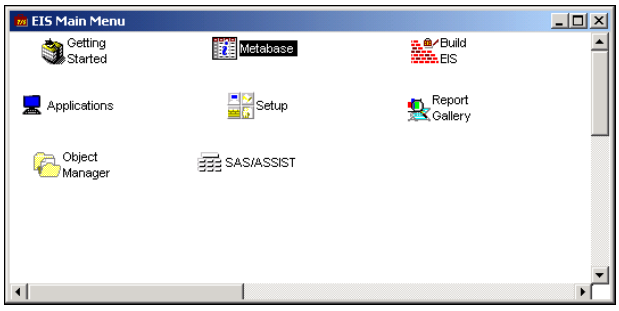
First, we need to create a libref to access our data. We'll issue code similar to the following. A SAS data set called BP1 will be located in this location.



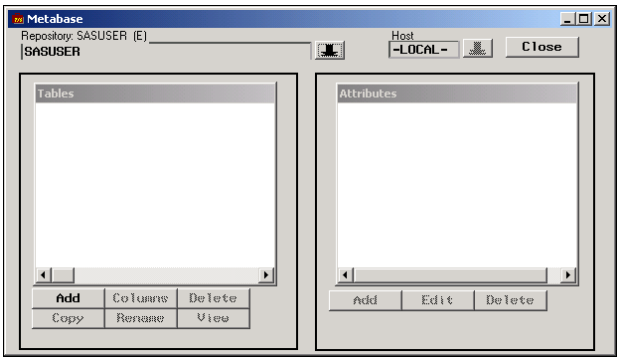
Now, we'll start EIS software.



Select Metabase.

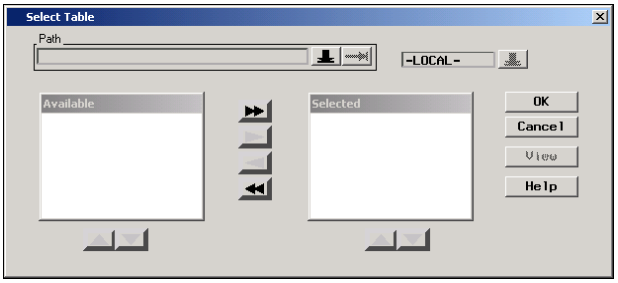


This will prompt us for a location to store our metadata. This is information about the actual data we will publish. The original data can be in detail form.

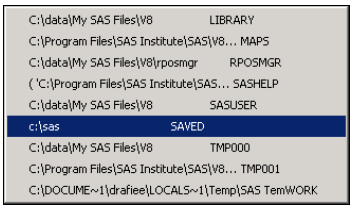


The repository will be SASUSER for this demonstration. Select Add

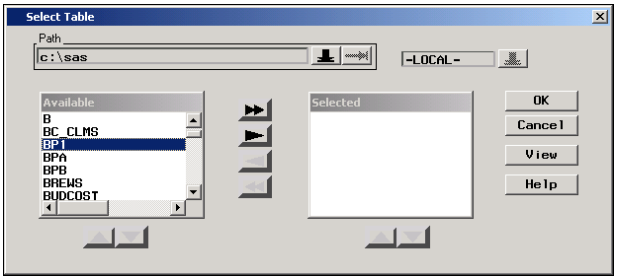
We will add the libref SAVED that we just defined to this metabase repository.

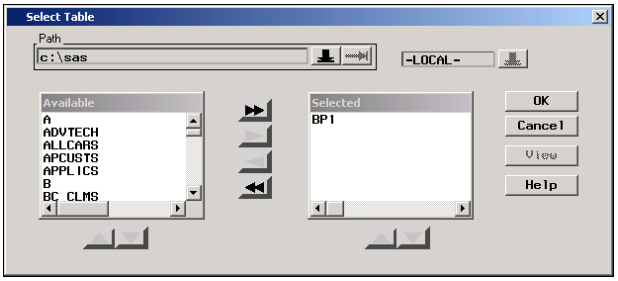


Select the  to see the following.

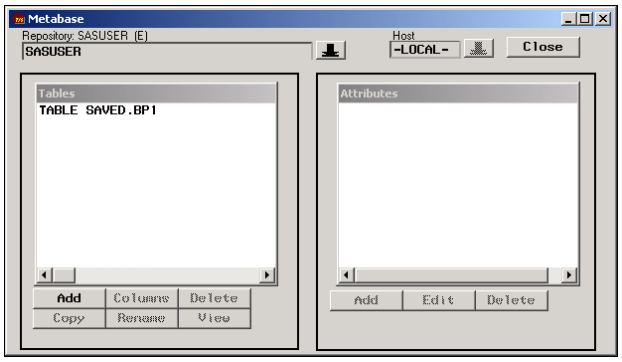


Select the BP1 data set.

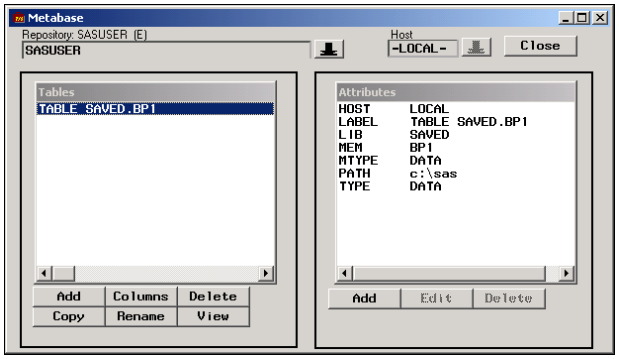




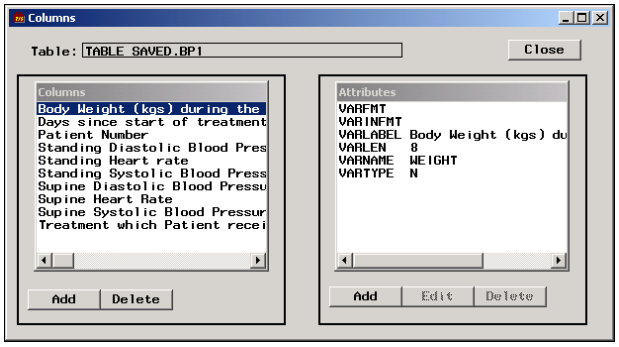
Click OK when finished.



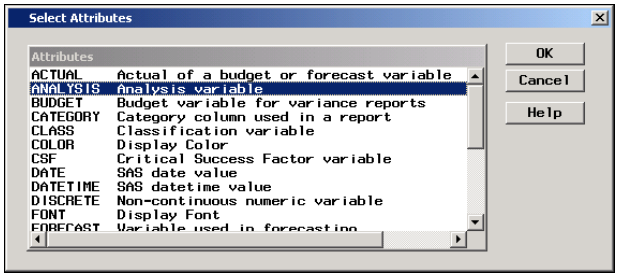
Highlight the Table listed to see the potential attributes currently assigned.



Click the Columns button to assign various attributes to particular columns in the data.

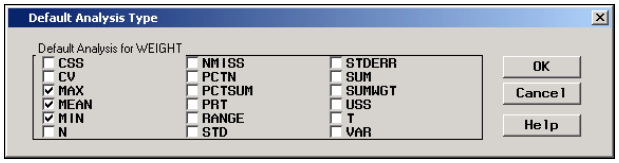


Once a column is highlighted, click Add to add the Analysis attribute to this variable.

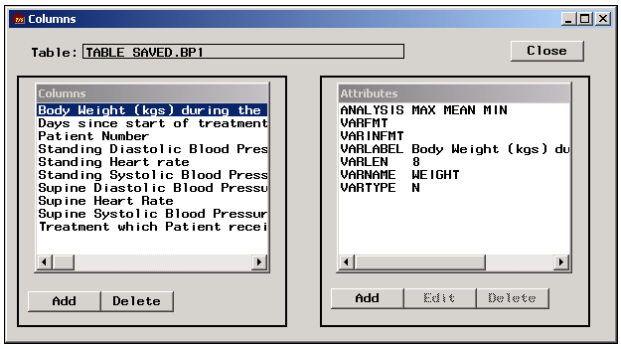


Click OK when finished.

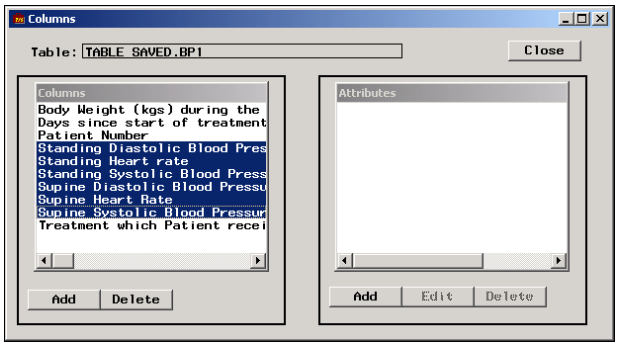
Select the type of statistics desired.



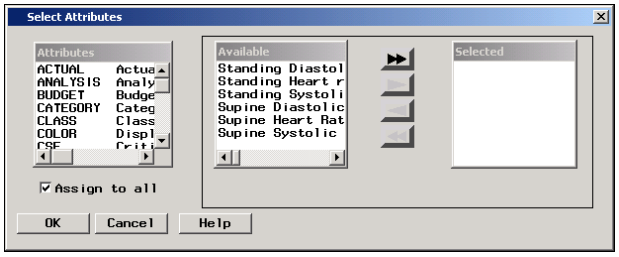
Click OK when finished.



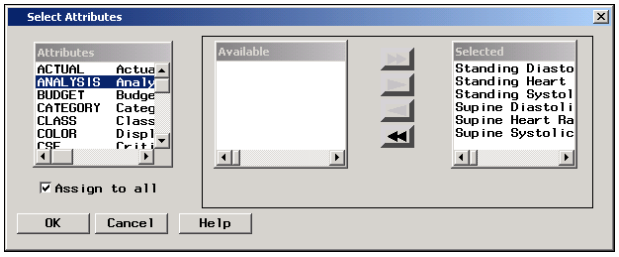
You can specify an attribute for several columns at a time by highlighting them.



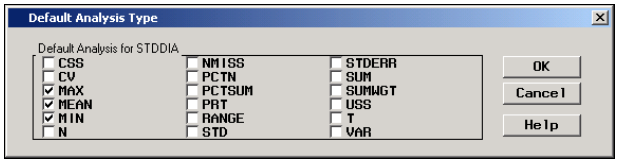
Click Add and the following screen will be displayed.



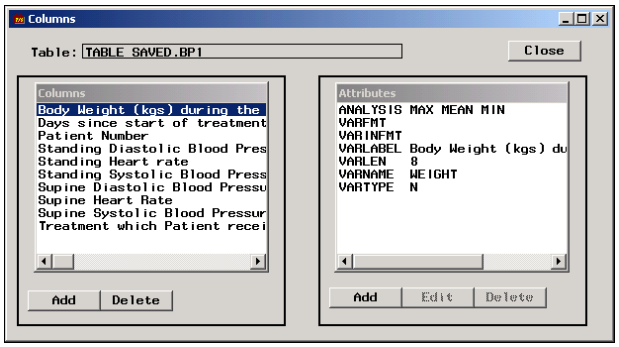
Select Analysis.



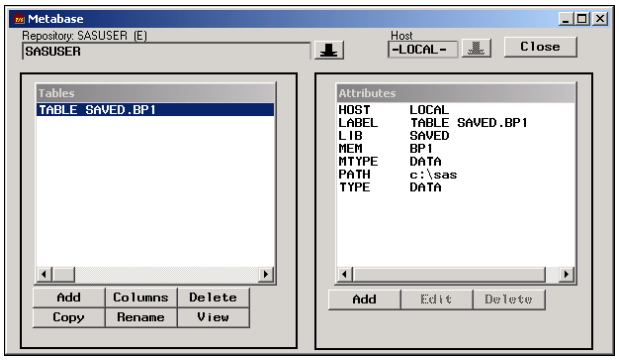
Click OK when finished.



Select the appropriate statistics and click OK

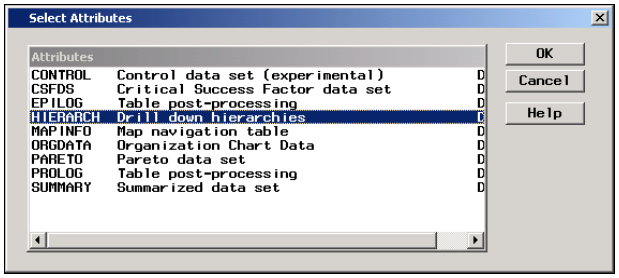


Click Close when you are done.



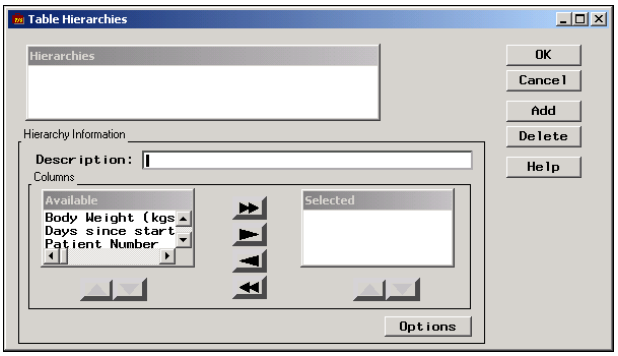
We will now add a Hierarchy to our data. This will allow some drilldown selection when the data is presented. Click the right hand Add button.

Select

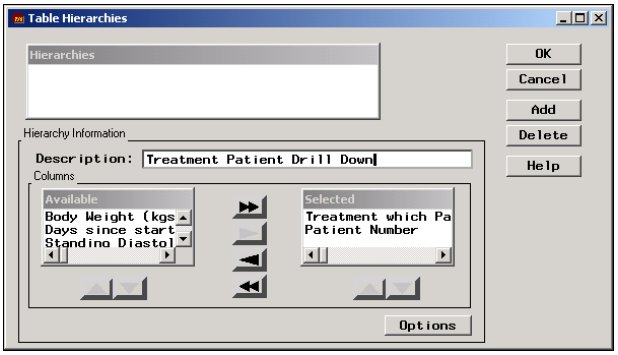


and click OK

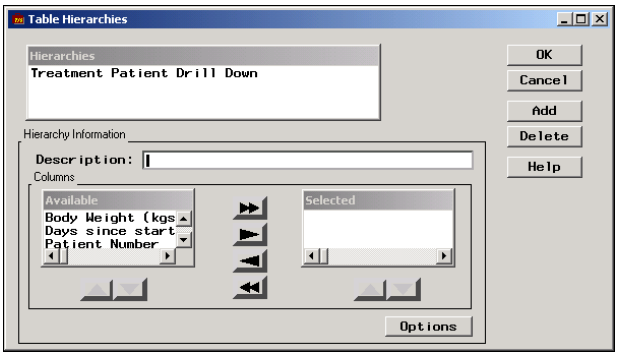
Now, we'll go through a process to define several variables to a hierarchy and then save the hierarchy with a name.



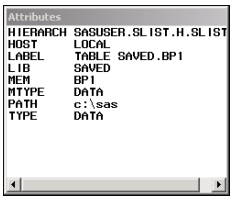
Type the name of the drilldown hierarchy and assign variables to it in the desired order.



Click Add when you have completed your selection for this hierarchy. You will see it appear at the top of the screen.



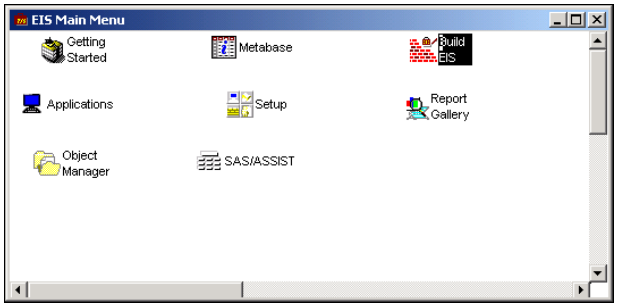
Click OK when finished and you'll see a hierarchy defined in the metadata.



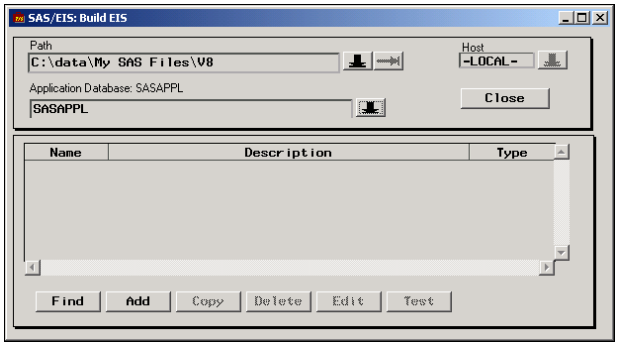
Click Close to close the metabase window.

Now we must build an MDDB. We'll do this through the prompting capabilities offered with SAS/EIS software. Otherwise, PROC MDDB could be used to perform similar tasks.

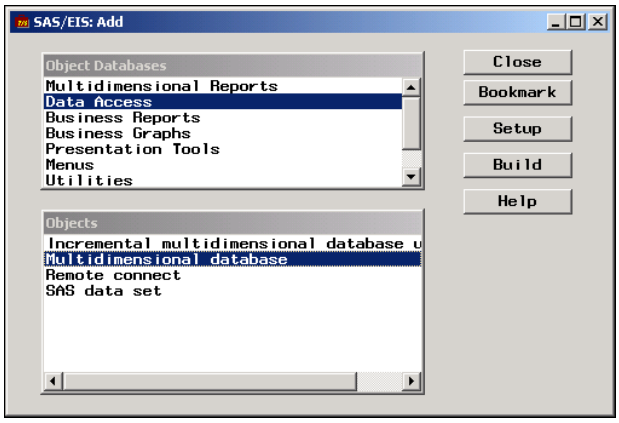
Select Build EIS.



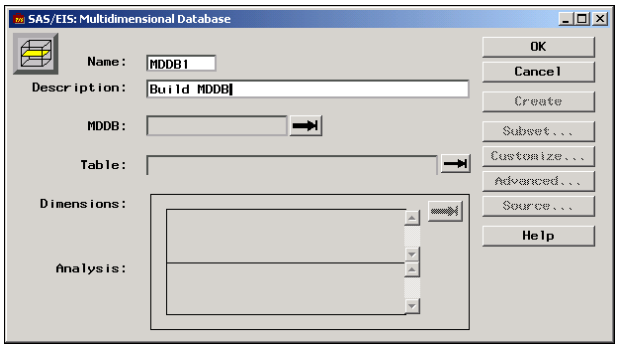
We will use the default EIS Application location. Notice it is currently empty.



Click Add to create an MDDB.



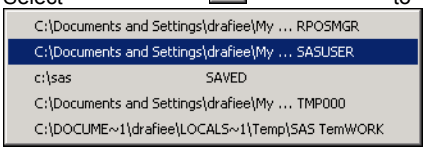
Select Build to take you to the following window.

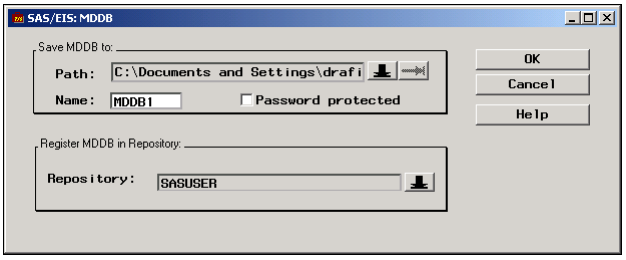


Select the first button

Choose a location and name for the MDDB>

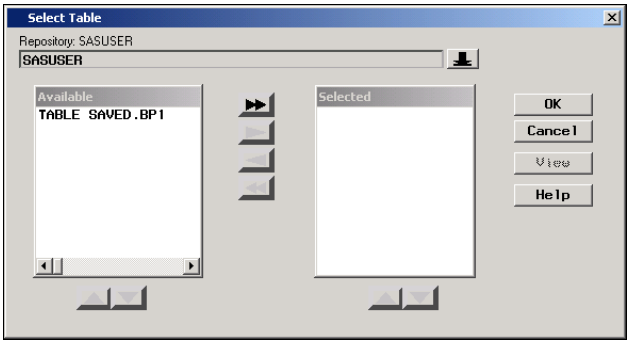
Select to choose



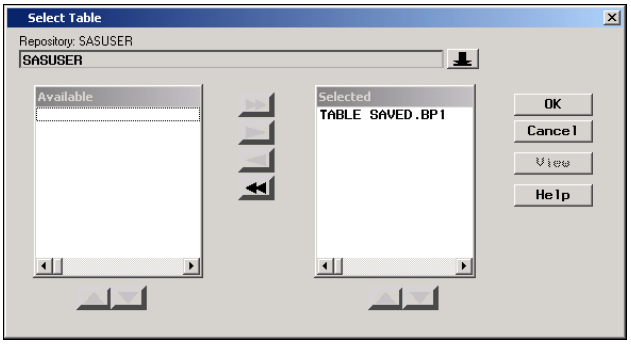


Select OK when finished.

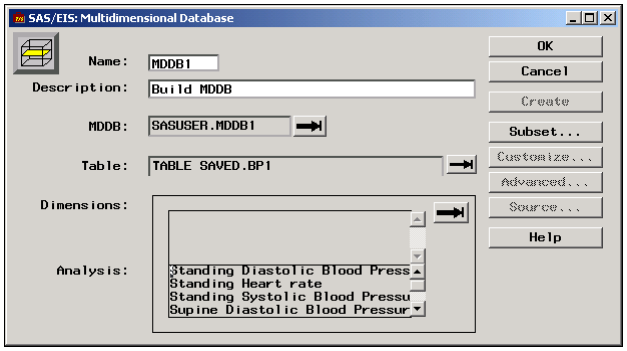
Select the second button to see the following screen.



It is important to choose a predefined data set at this point.



Select OK when finished.

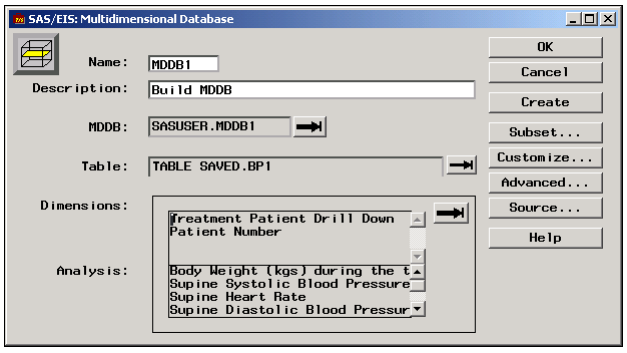


For the Dimension, select the button 

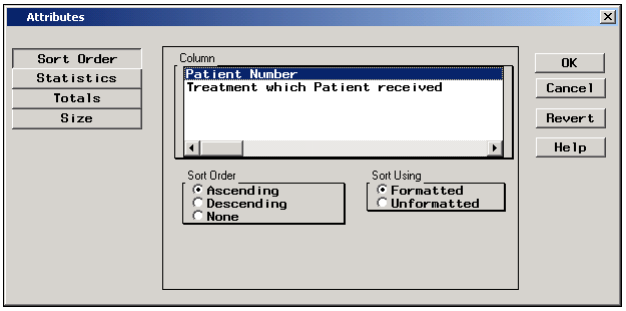
Choose the appropriate dimensions desired.

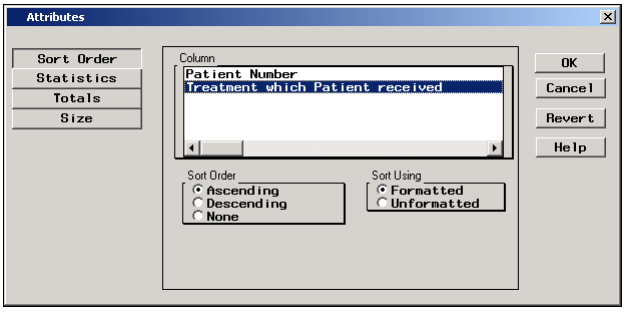
Choose the appropriate analysis desired.

Select  when finished.

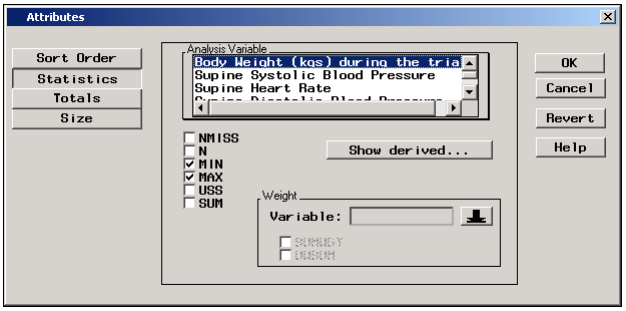


Choose  to select the order of column data. Here, we will select everything in ascending order.

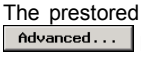


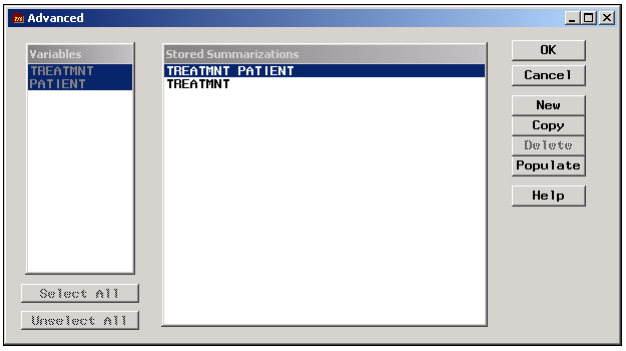


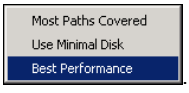
We will choose desired statistics.



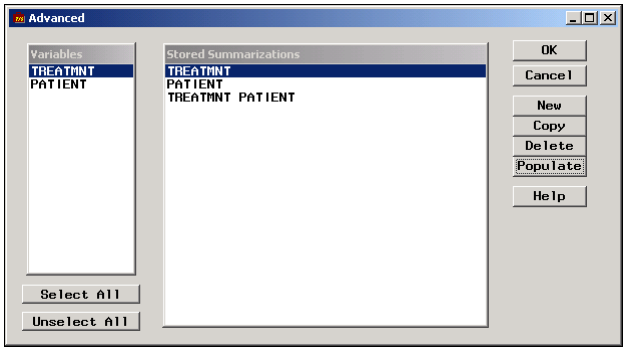
Click  when finished.

The prestored hierarchy of the MDDB can be selected with the  button.

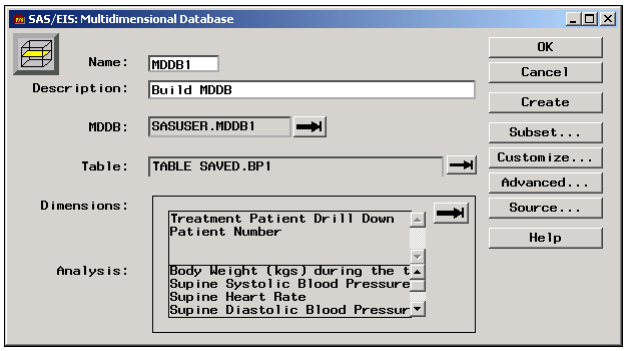


We will select  to choose 

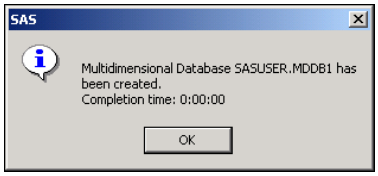
. This stores all possible hierarchies available. This works well for the fastest access speed during use. This also works well because the data we are using is very small. Results of MDDB size will vary with more hierarchies. It is a good idea to keep the MDDB structure to a minimal number of hierarchies.



Click OK when finished.

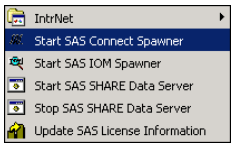


Click Create to actually create the MDDB. The following message will be displayed when processing is finished.

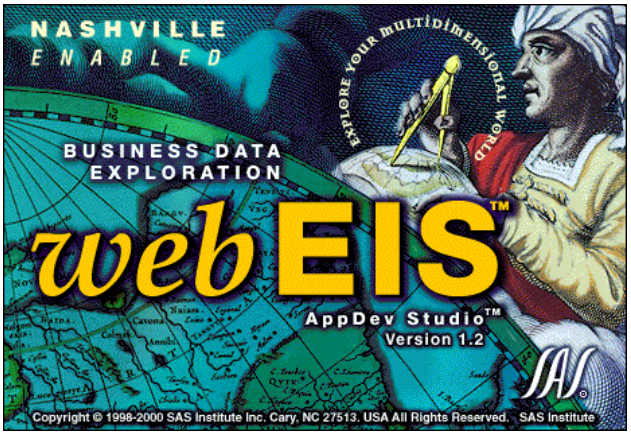


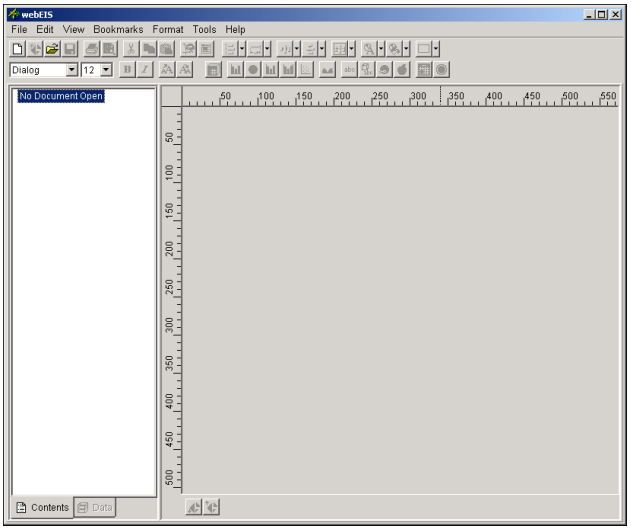
Click OK, OK and Close to end out of this.

Under the Appdev Studio menuing selection, load the Spawner. This is used for development purposes.

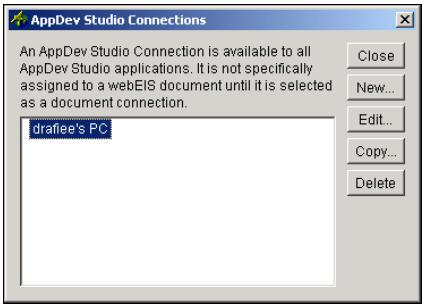


Now load webEIS from the Appdev Studio menu to see the following screen.

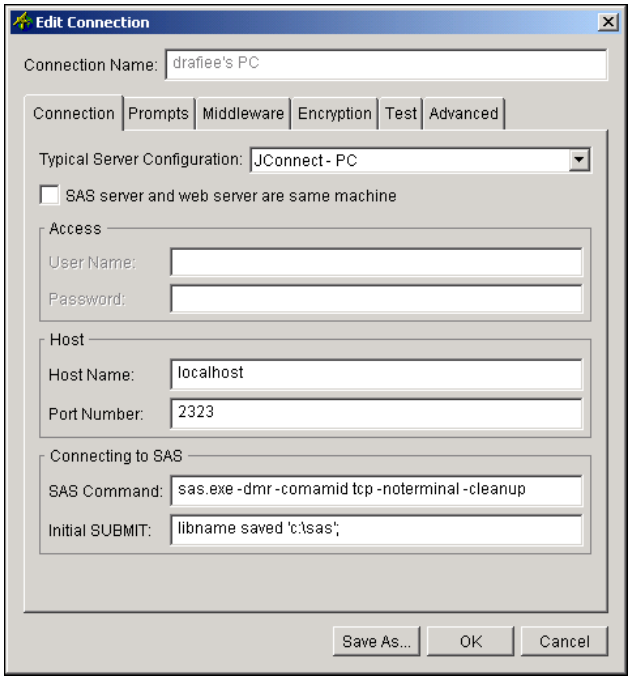




First, we need to check our connections to make sure we have access to all of the data needed. Select Tools, Appdev Studio Connections

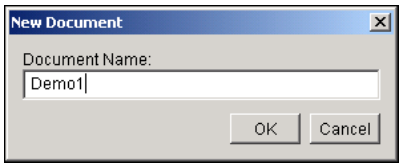


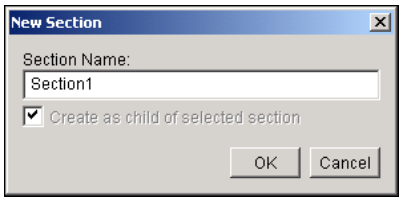
Select Edit... to see your connections.



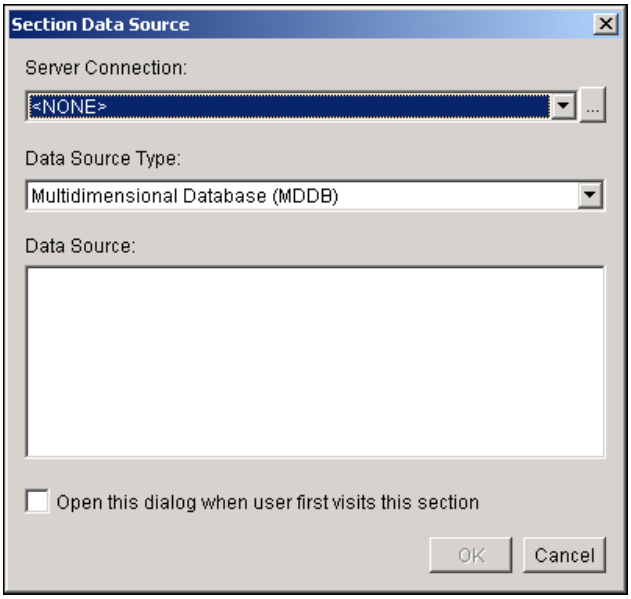
Modify it to add the appropriate libname statement in the Initial Submit section and then click  when done.

Now, create a new document and section. Select File, New

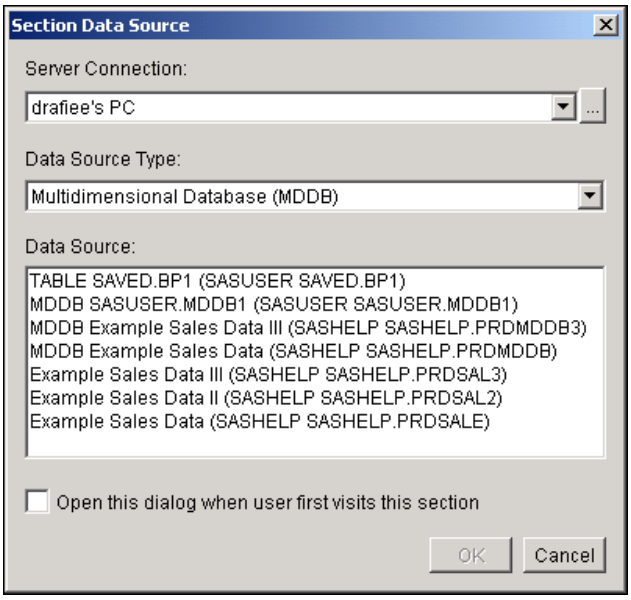




and click 

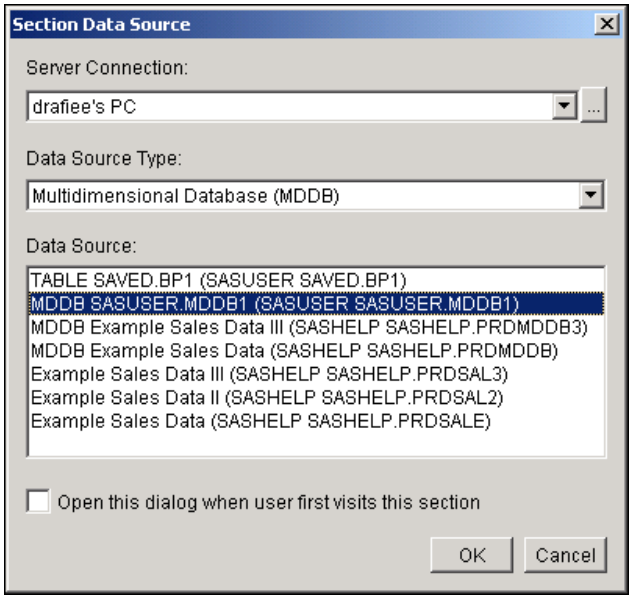


Select your PC

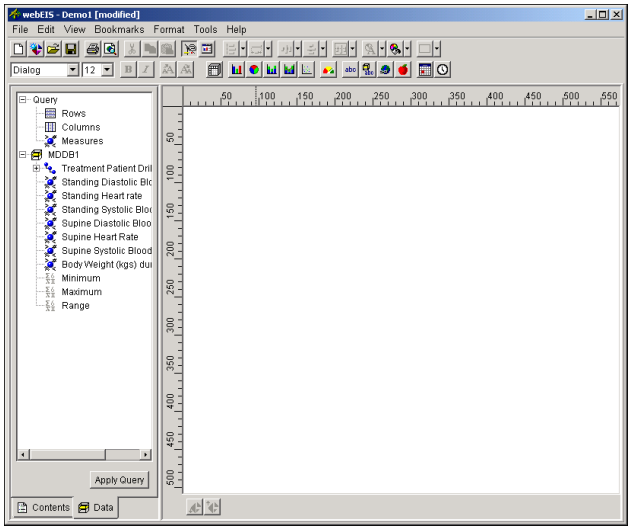


All of the tables available will be listed.

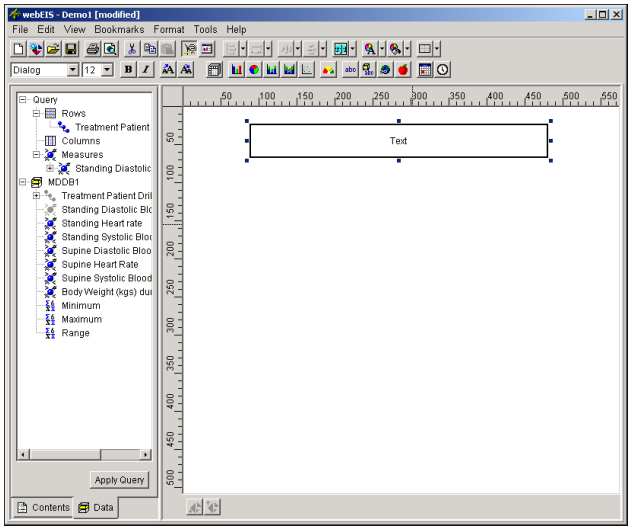
Select the MDDB file created earlier.

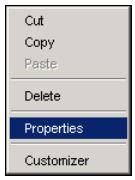


Click  and the following screen is displayed.

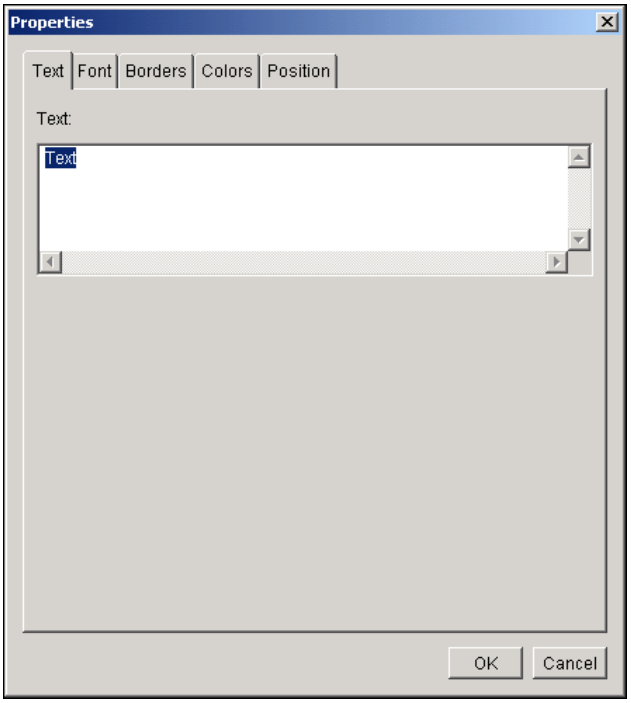


Place a title on the webEIS document by clicking on the abc button. Draw out the area for the title.

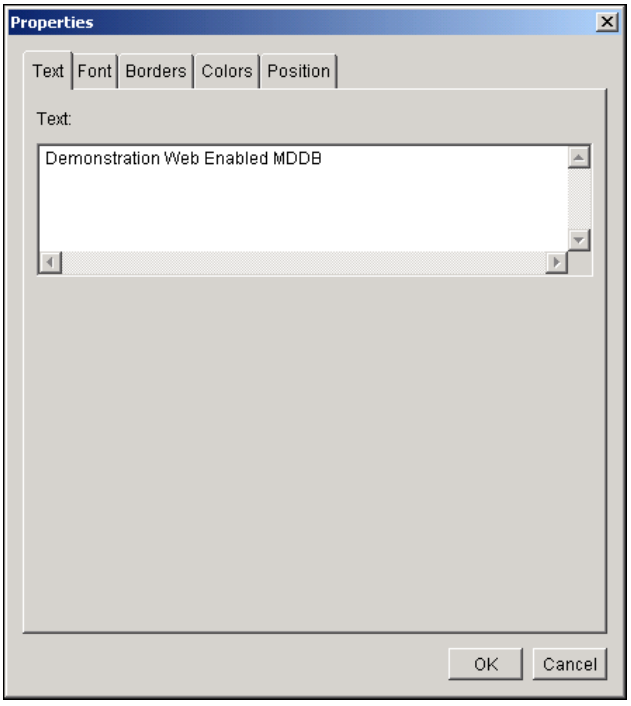




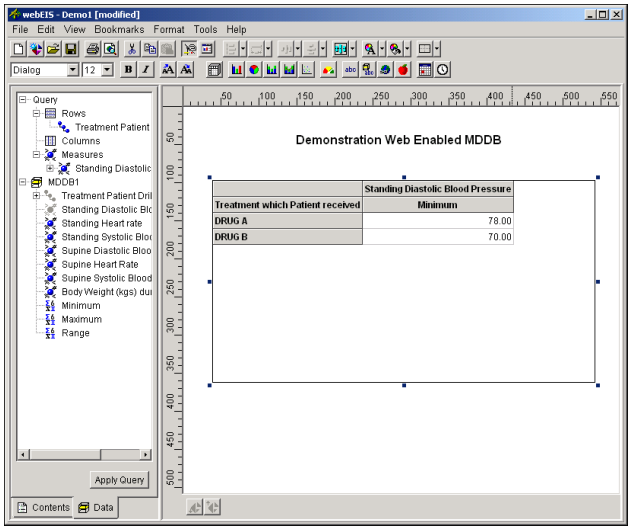
Right click on the Text box and select



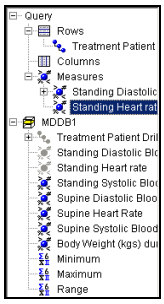
Change the title to what is appropriate and click 



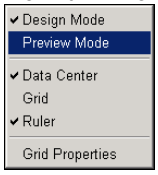
Select the MDDB button and draw out the location for the MDDB to be displayed.

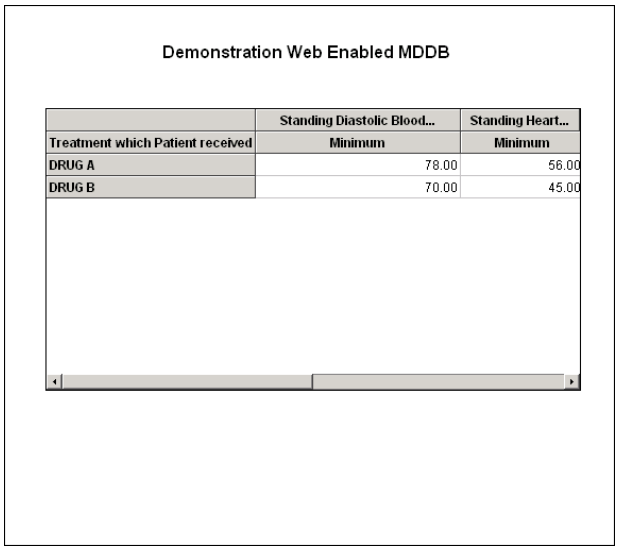


Drag and drop the desired statistics in the left hand display onto the word Measures.

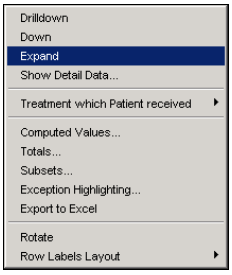


View it in Preview Mode.by Selecting the View pulldown and

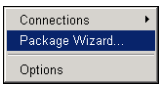




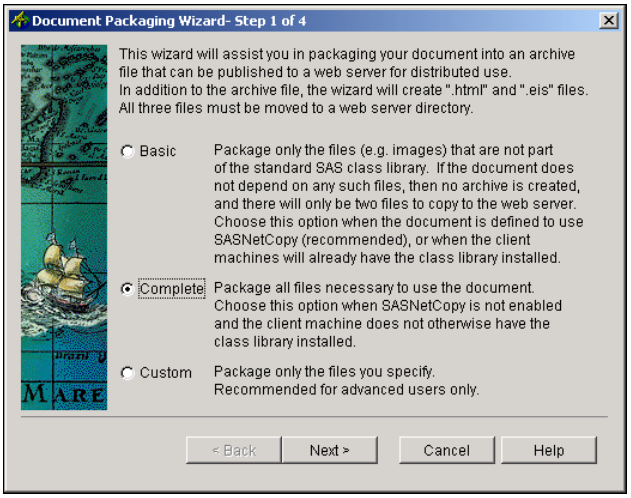
A right click displays the following menu.

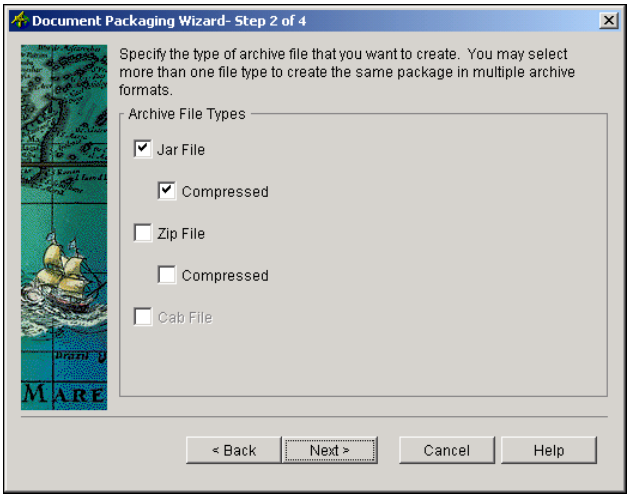


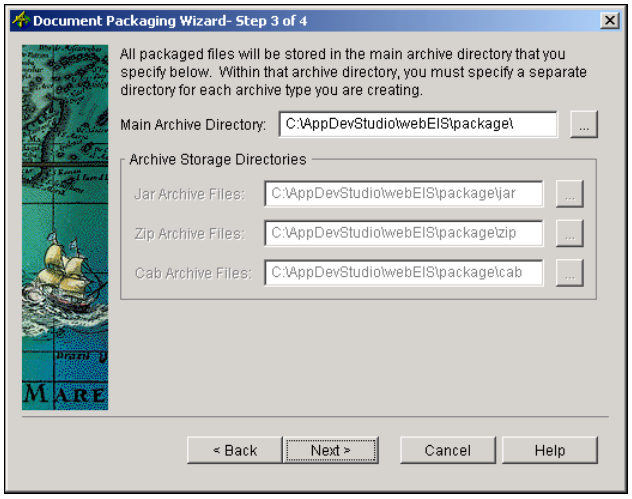
When you are ready to package it, select Tools

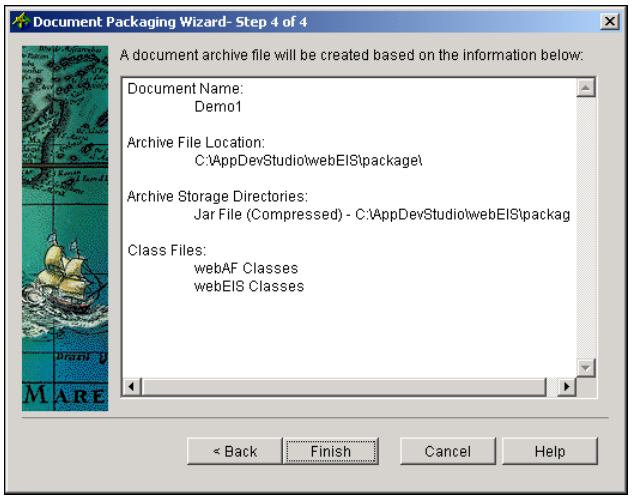


The following menu will appear. We strongly suggest you select Complete to use SASNetCopy. This bundles everything needed for long term use and speed of web based MDDB access. Click Next and following the menus, making sure the items are selected as we have them here.

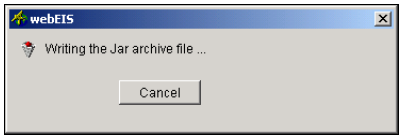


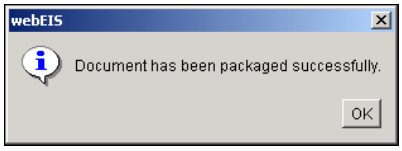




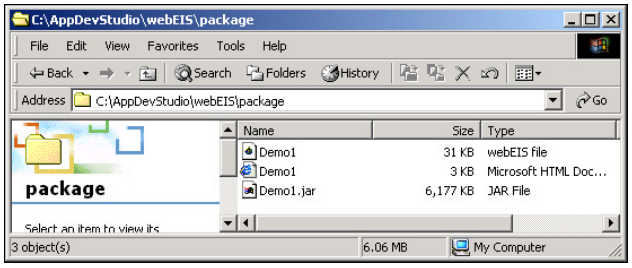


When you click Finish, prompts similar to the following will appear.





You will find all of the documents needed in the following location.



They must all be moved to the web server. Load the HTML file and the MDDB will appear.

Reading and Writing Data from Microsoft Excel/Word Using DDE

Destiny Corporation

Prepared by Dana Rafiee
Destiny Corporation • 100 Great Meadow Rd Suite 601• Wethersfield, CT 06109-2355
Phone: (860) 721-1684 • 1-800-7TRAINING • Fax: (860) 721-9784
Email: info@destinycorp.com Web Site: www.destinycorp.com

Dynamic Data Exchange to Excel

Dynamic Data Exchange is a means of transferring data between different Windows applications such as the SAS System and Microsoft Excel or Microsoft Word. The technique uses what is known as a Client / Server type relationship, where the Client application makes the requests and the Server application responds accordingly. When using DDE with the SAS System, SAS is always the Client.

The DDE technique may be used within a Data Step, a Macro or from inside a SAS/AF application.

Using the DDE technique involves specifying the following three components in what is called a DDE Triplet:

1. The path and name of the executable file for the server application. For example, with Microsoft Excel the name of the executable is EXCEL.

2. The full path and file name of the document or spreadsheet with which you wish to share data. This is referred to in SAS documentation as the Topic.

3. The location of data to be read or modified. With a spreadsheet application, this is a range of cells. In applications such as Microsoft Word, the location of data is defined by what are known as bookmarks.

An example of a DDE Triplet for an Excel Spreadsheet is as follows:

```
EXCEL | C:\COURSES\DATA\CLIN.XLS ! R4C17:R17C8
```

The DDE Triplet is then incorporated into a Filename statement of the following form:

```
FILENAME fileref DDE 'DDE-Triplet' | 'CLIPBOARD' <DDE-Options>;
```

where:

fileref is a valid fileref

DDE is the keyword that primes the SAS System for Dynamic Data Exchange.

'DDE-Triplet' is the three part specification of the DDE external file and takes the following form: 'Executable-file-name|Topic!Data-range'. Note that the triplet is application dependent and may differ between different releases of the same application.

CLIPBOARD' is an alternative to directly specifying the DDE triplet. To use this option, copy the required data from the Server application to the Clipboard and specify the 'CLIPBOARD' option in the Filename statement. SAS will then determine the Triplet for itself. You will be able to see the triplet definition by looking at the Log.

DDE-Options are as follows:

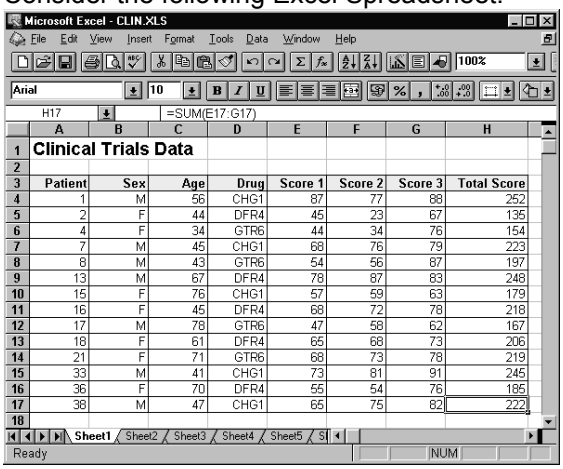
HOTLINK invokes the DDE HOTLINK facility, which causes the DDE link to be activated whenever data in the spreadsheet range is updated.

NOTAB makes the SAS System ignore tab characters between variables.

COMMAND allows remote commands to be sent to DDE server applications.

Example of using DDE to read data from Excel

Consider the following Excel Spreadsheet:



	A	B	C	D	E	F	G	H
1	Clinical Trials Data							
2								
3	Patient	Sex	Age	Drug	Score 1	Score 2	Score 3	Total Score
4	1	M	56	CHG1	87	77	88	252
5	2	F	44	DFR4	45	23	67	135
6	4	F	34	GTR6	44	34	76	154
7	7	M	45	CHG1	68	76	79	223
8	8	M	43	GTR6	54	56	87	197
9	13	M	67	DFR4	78	87	83	248
10	15	F	76	CHG1	57	59	63	179
11	16	F	45	DFR4	68	72	78	218
12	17	M	78	GTR6	47	58	62	167
13	18	F	61	DFR4	65	68	73	206
14	21	F	71	GTR6	68	73	78	219
15	33	M	41	CHG1	73	81	91	245
16	36	F	70	DFR4	55	54	76	185
17	38	M	47	CHG1	65	75	82	222
18								

With Excel running and the above spreadsheet open, a simple Data Step can be written to read the above data into a SAS Data Set.
The DDE Triplet components for the above example are as follows:

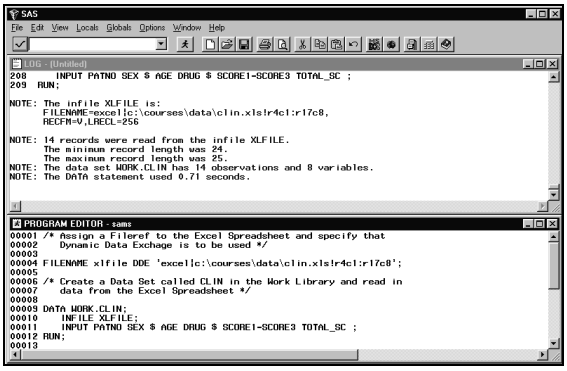
Name of Executable File: EXCEL

Full Path and Name of spreadsheet file: C:\COURSES\DATA\CLIN.XLS

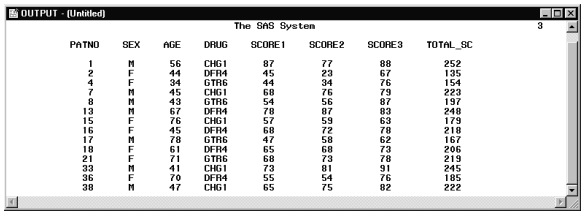
Location of data (Row and Column coordinates): R4C1:R17C8

Note how the two sets of coordinates that define the range of data are separated by a colon.

The SAS statements required to read the data into a SAS data set are as follows:



Running the above code generates the following data set:

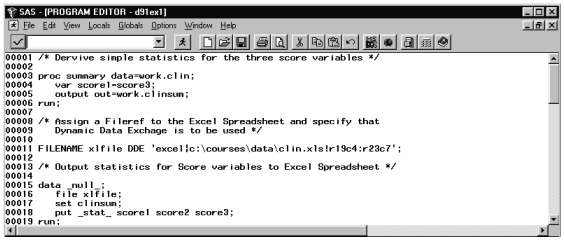


Example of using DDE to write data to Excel

To output data to an Excel spreadsheet, define a DDE Triplet which contains the co-ordinates of the cells to which data is to be written.

Then use a null data step containing 'Put' statements, to write out the values of required variables to the spreadsheet.

For example, the following code uses a Proc Summary to calculate simple statistics on the data that was originally read in from Excel. The statistics are output to a data set, which is used to update the Excel spreadsheet:



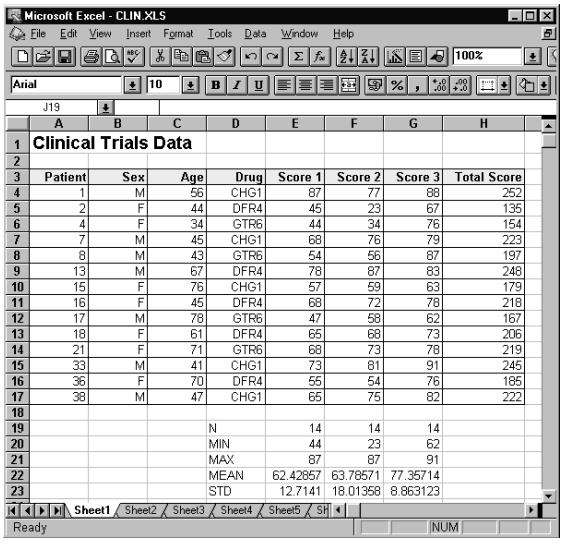
Note that the only change to the Filename statement compared with the previous example, is that the DDE Triplet points at a different range of cells in the spreadsheet.

A null data step is used to read the data set containing the statistics and 'Put' statements write out the values of selected variables to the Excel spreadsheet. This is no different to writing out data to a text file.

Note that a null data step is used when you wish to avoid creating another data set.

Running the above code updates the Excel spreadsheet accordingly:

The updated Excel spreadsheet complete with summary statistics:



Invoking an application from SAS

In the previous examples, the invocation of Excel and opening of a spreadsheet file has been done by clicking on the required Icon and using the pull down menus within Excel.

The application remains running in the background until the user actively closes it down.
Ideally, you only want an application running for the time that it takes to access the data. Otherwise, the idle application ties up valuable memory that can be better used. The solution to this problem is to open and close applications from your SAS program.

An application may be called from SAS by using the following statements:

```
OPTIONS NOXWAIT NOXSYNC;

X 'C:\EXCEL\EXCEL';

DATA _NULL_;
   T=SLEEP(5);
RUN;
```

NOXWAIT causes control to be returned to SAS as soon as the specified 'X' command has been executed. In the above example, as soon as Excel has been loaded, control will return to SAS.

NOXSYNC causes the application called from SAS to execute asynchronously so that it does not interfere with the SAS session.

Remember to reset the above options to XWAIT and XSYNC when the application is closed. The **X** statement contains the path and name of the executable file that calls the application. The **Sleep** statement inside the null data step simply gives the called application time to load. In the above example, 5 seconds has been allowed.

Issuing Commands from SAS

Having invoked an application like Excel using the above statements, you need to issue commands that carry out tasks such as opening the required spreadsheet file and selecting data. This is done through what is called the SYSTEM Topic:

Assign a Fileref that includes the DDE topic SYSTEM to enable Excel or Word commands to be issued from SAS.

```
FILENAME CMDS DDE 'Excel|SYSTEM';
```

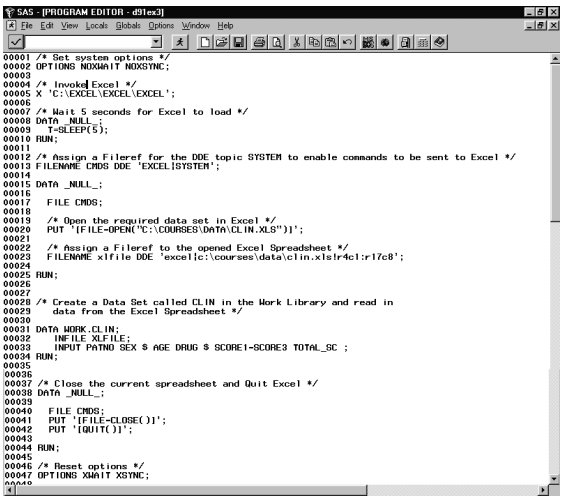
Commands may now be sent to Excel by using 'Put' statements inside a null data step. Note that the syntax of the commands issued is application dependent.

```
DATA _NULL_;

    FILE CMDS;
    PUT                                    '[FILE-
OPEN(|C:\COURSES\DATA\CLIN.XLS=)]';

RUN;
```

Example of invoking and issuing commands to Excel

The example program does the following:

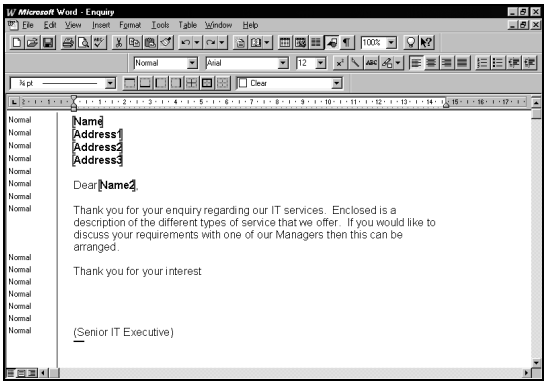
1. Invokes Excel.
2. Opens a Spreadsheet.
3. Reads spreadsheet data into a SAS data set.
4. Closes the Spreadsheet.
5. Closes Excel.



Dynamic Data Exchange to Word

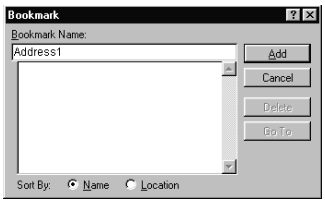
Using DDE to access data stored in a Microsoft Word document is the almost the same procedure as used for Excel. The only difference is that the location of data in Word is specified by referencing Bookmarks rather than a cell range.

A bookmark is simply a named marker placed within the Word document. The following Word Document uses bookmarks to denote the position of Name and Address fields:



To create a bookmark within a Word document, highlight the text which you want to use as a bookmark, then select **Edit** and **Bookmark** from the pull down menu:

Enter a name for the bookmark and click on **Add**



The components of the DDE Triplet for the above example are as follows:

Name of Executable File: WINWORD

Full Path and Name of Word document: C:\COURSES\DATA\ENQUIRY.DOC

Location of data (Name of Bookmark): Address1

Invoking Word from SAS

The procedure used to call Word from SAS is the same procedure used to call Excel or any other Windows application from SAS. Set the appropriate options, use an 'X' statement to call Word, and allow Word a few seconds to load.

```
OPTIONS NOXWAIT NOXSYNC;

X 'C:\WINWORD\WINWORD';
DATA _NULL_;
   T=SLEEP(5);
RUN;
```

Remember to reset the above Options to XWAIT and XSYNC when the application is closed.

Issuing Commands to Word from SAS

As with Excel, this is done by assigning a Fileref to the DDE topic SYSTEM:

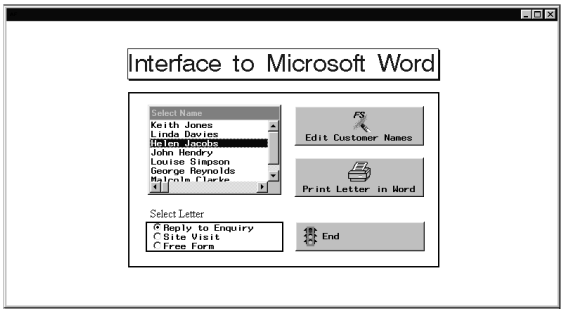
```
FILENAME CMDS DDE 'WINWORD|SYSTEM';
```

Once the above Fileref is assigned, commands can be sent to Word by using 'Put' statements inside a null data step.

```
DATA _NULL_;

    FILE CMDS;
    PUT
'[FILEOPEN(|C:\COURSES\DATA\ENQUIRY.DOC=)]';

RUN;
```

Example of using DDE to access a Word document

The following SAS/AF Application automatically inserts the name and address of a customer into the selected Word document and prints it:



The name and address information is stored in a SAS data set that may be accessed and updated by clicking on the 'Edit Customer Names' icon.

The name and address information of the selected customer is inserted into the Word document as follows:

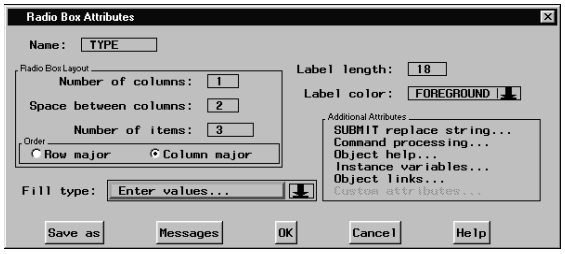
Note how the name and address details have been written to their respective bookmarks. The square brackets, which denote the boundary of a bookmark, do not appear in the printed document.

Note the following about the SCL Program that controls the Frame:

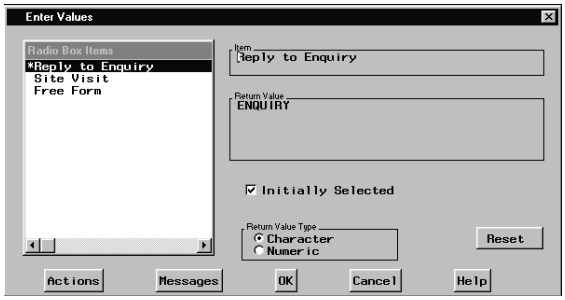
1. The physical paths for the Word executable file and Word documents, are assigned to Macro variables in the Autoexec file which runs before the application is called. Call Symget routines are used in the INIT section to assign the values of these Macro variables to SCL variables. This approach avoids having to hard code path names in the application.

2. The name of the Word document to open is derived by concatenating the document path, letter type, and the suffix '.DOC'.

3. The letter type is obtained by sending the _GET_TEXT_ method to the Radio Box, which returns the Character value assigned to the selected item. Each Radio Box item has a Character return value specified in the attributes window:



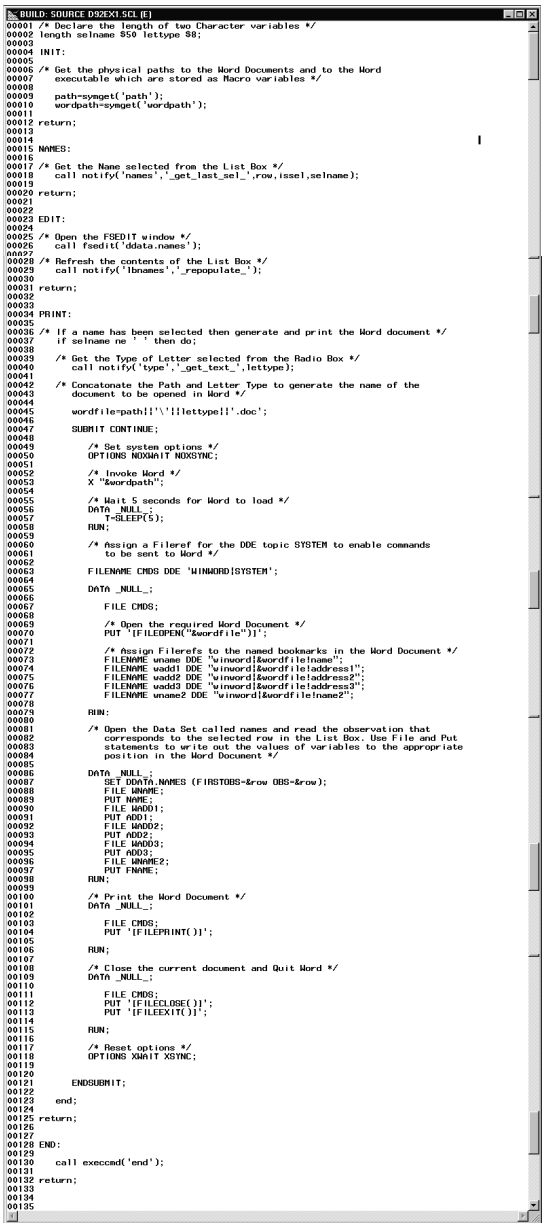
Click on  to access Return values for items



Clicking on a Radio Box Item displays its Return value.

4. A Submit Block contains all of the SAS code used to load Word, access the selected document and print it. Recall that an ampersand within a Submit Block references an SCL variable.
5. Five different Filrefs are assigned to point at the various name and address bookmarks in the Word document.

The SCL Program is as follows (see next column):



```
BUILD: SOURCE D92EX1.SCL (E)
00001 /* Declare the length of two Character variables */
00002 length selname $50 lettype $8;
00003
00004 INIT:
00005
00006 /* Get the physical paths to the Word Documents and to the Word
00007    executable which are stored as Macro variables */
00008
00009    path=symget('path');
00010    wordpath=symget('wordpath');
00011
00012 return;
00013
00014
00015 NAMES:
00016
00017 /* Get the Name selected from the List Box */
00018    call notify('names','_get_last_sel_',row,issel,selname);
00019
00020 return;
00021
00022
00023 EDIT:
00024
00025 /* Open the FSEDIT window */
00026    call fsedit('ddata.names');
00027
00028 /* Refresh the contents of the List Box */
00029    call notify('lbnames','_repopulate_');
00030
00031 return;
00032
00033
00034 PRINT:
00035
00036 /* If a name has been selected then generate and print the Word document */
00037    if selname ne '' then do;
00038
00039    /* Get the Type of Letter selected from the Radio Box */
00040       call notify('type','_get_text_',lettype);
00041
00042    /* Concatenate the Path and Letter Type to generate the name of the
00043       document to be opened in Word */
00044
00045       wordfile=path||'\'||lettype||'.doc';
00046
00047       SUBMIT CONTINUE;
00048
00049          /* Set system options */
00050          OPTIONS NOXWAIT NOXSYNC;
00051
00052          /* Invoke Word */
00053          X "&wordpath";
00054
00055          /* Wait 5 seconds for Word to load */
00056          DATA _NULL_;
00057             T=SLEEP(5);
00058          RUN;
00059
00060          /* Assign a Fileref for the DDE topic SYSTEM to enable commands
00061             to be sent to Word */
00062
00063          FILENAME CMDS DDE 'WINWORD|SYSTEM';
00064
00065          DATA _NULL_;
00066
00067             FILE CMDS;
00068
00069             /* Open the required Word Document */
00070             PUT '[FILEOPEN("&wordfile")]';
00071
00072             /* Assign Filerefs to the named bookmarks in the Word Document */
00073             FILENAME wname DDE "winword|&wordfilename";
00074             FILENAME wadd1 DDE "winword|&wordfile!address1";
00075             FILENAME wadd2 DDE "winword|&wordfile!address2";
00076             FILENAME wadd3 DDE "winword|&wordfile!address3";
00077             FILENAME wname2 DDE "winword|&wordfilename2";
00078
00079          RUN;
00080
00081          /* Open the Data Set called names and read the observation that
00082             corresponds to the selected row in the List Box. Use File and Put
00083             statements to write out the values of variables to the appropriate
00084             position in the Word Document */
00085
00086          DATA _NULL_;
00087             SET DDATA.NAMES (FIRSTOBS=&row OBS=&row);
00088             FILE WNAME;
00089             PUT NAME;
00090             FILE WADD1;
00091             PUT ADD1;
00092             FILE WADD2;
00093             PUT ADD2;
00094             FILE WADD3;
00095             PUT ADD3;
00096             FILE WNAME2;
00097             PUT FNAME;
00098          RUN;
00099
00100          /* Print the Word Document */
00101          DATA _NULL_;
00102
00103             FILE CMDS;
00104             PUT '[FILEPRINT()]';
00105
00106          RUN;
00107
00108          /* Close the current document and Quit Word */
00109          DATA _NULL_;
00110
00111             FILE CMDS;
00112             PUT '[FILECLOSE()]';
00113             PUT '[FILEEXIT()]';
00114
00115          RUN;
00116
00117          /* Reset options */
00118          OPTIONS XWAIT XSYNC;
00119
00120
00121       ENDSUBMIT;
00122
00123    end;
00124
00125 return;
00126
00127
00128 END:
00129
00130    call execcmd('end');
00131
00132 return;
00133
00134
00135
```

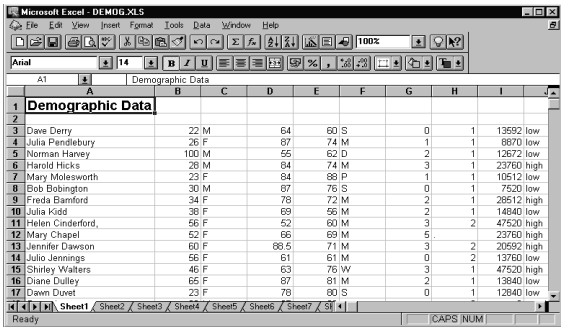
Workshop Session

Create an Excel Spreadsheet and enter the text 'Demographic Data' in cell A1. Save the spreadsheet and make a note of the path and file name used.

Write a SAS program to do the following:

1. Invoke Excel.
2. Open the spreadsheet file that you have created for this exercise.
3. Write out values of Demographic Data Set variables to the Excel spreadsheet.
4. Save the spreadsheet.
5. Close the spreadsheet file.
6. Close Excel.

The resulting spreadsheet should appear as follows:



	A	B	C	D	E	F	G	H	I
1	**Demographic Data**								
2									
3	Dave Derry	22	M	64	60	S	0	1	13592 low
4	Julia Pendlebury	26	F	87	74	M	1	1	8870 low
5	Norman Harvey	100	M	55	62	D	2	1	12672 low
6	Harold Hicks	28	M	84	74	M	3	1	23760 high
7	Mary Molesworth	23	F	84	88	P	1	1	10512 low
8	Bob Bobington	30	M	87	76	S	0	1	7520 low
9	Freda Bamford	34	F	78	72	M	2	1	28512 high
10	Julia Kidd	38	F	69	56	M	2	1	14840 low
11	Helen Cinderford,	56	F	52	60	M	3	2	47520 high
12	Mary Chapel	52	F	66	69	M	5		23760 high
13	Jennifer Dawson	60	F	88.5	71	M	3	2	20592 high
14	Julio Jennings	56	F	61	61	M	0	2	13760 low
15	Shirley Walters	46	F	63	76	W	3	1	47520 high
16	Diane Dulley	65	F	87	81	M	2	1	13840 low
17	Dawn Duvet	23	F	78	80	S	0	1	12840 low

Hints:

Use a separate Fileref to define the range of cells into which people's names are written and include the NOTAB option i.e.:

```
FILENAME XLF1 DDE
'EXCEL|C:\COURSES\DATA\DEMOG.XLS!r3c1:
r52c1' NOTAB;
```

The NOTAB option makes SAS ignore Tab Characters between variables. In this example, the space between the first name and surname is taken as a Tab Character, which results in the two parts of the name being put into different columns in Excel unless NOTAB is specified. Use a second Fileref to define the range of cells for the remaining variables but do not include the NOTAB option as non of the variables include space characters:

```
FILENAME XLF2 DDE
'EXCEL|C:\COURSES\DATA\DEMOG.XLS!r3c1:r52c1';
```

Within a null Data Step, use separate File and Put statements of the following form to output the data:

```
DATA _NULL_;
    FILE XLF1;
    PUT NAME;
    FILE XLF2;
    PUT AGE GENDER etc;
RUN;
```

Remember to reset options NOXWAIT and NOXSYNC to XWAIT and XSYNC respectively.

Interactive Proc Report Version 8

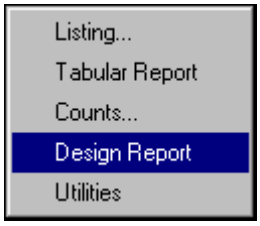
Prepared by

International SAS® Training and Consulting

Destiny Corporation • 100 Great Meadow Road, Suite 601 • Wethersfield, CT 06109-2379
Phone: (860) 721-1684 • 1-800-7TRAINING • Fax: (860) 721-9784
Web: WWW.DESTINYCORP.COM Email: info@destinycorp.com
2001-06-27

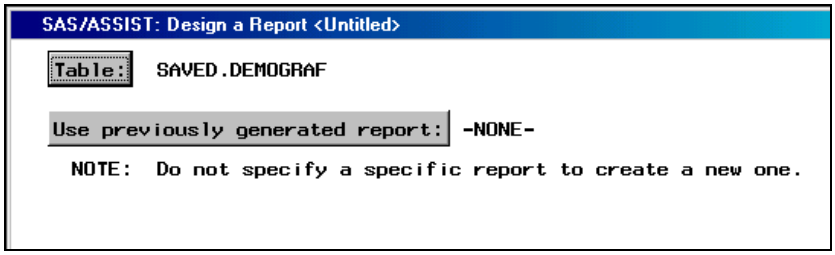
Interactive Report Generation

From the 

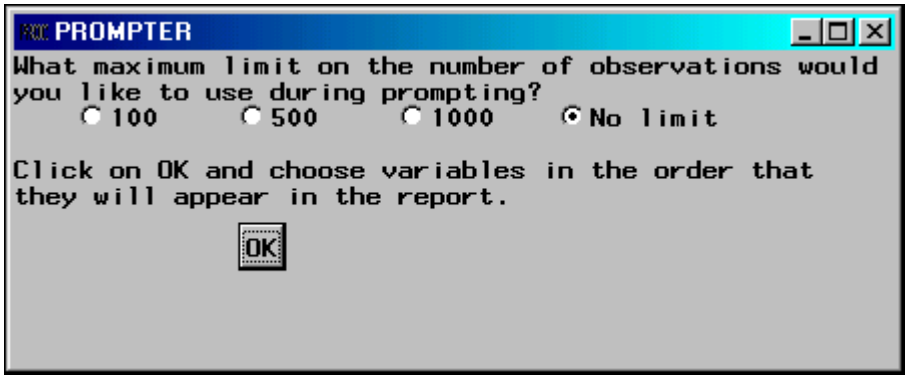
, click on the 

option. Select **'Design a report...'**

Select **SAVED.DEMOGRAF** AS THE 'Active data set:'



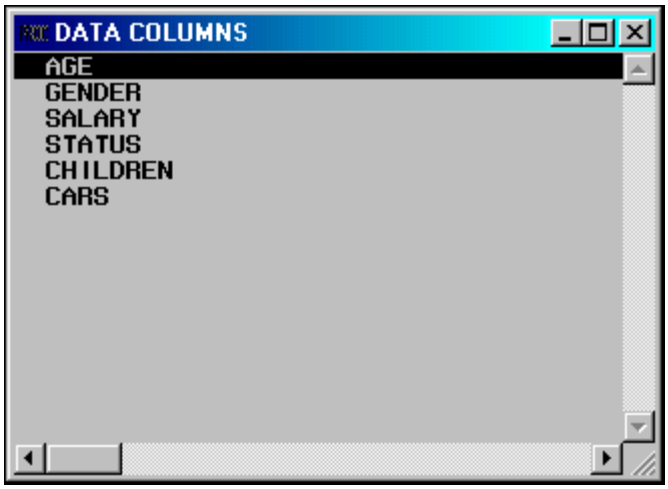
Run it and the prompter will appear. You can restrict the number of observations used to interactively create the report.



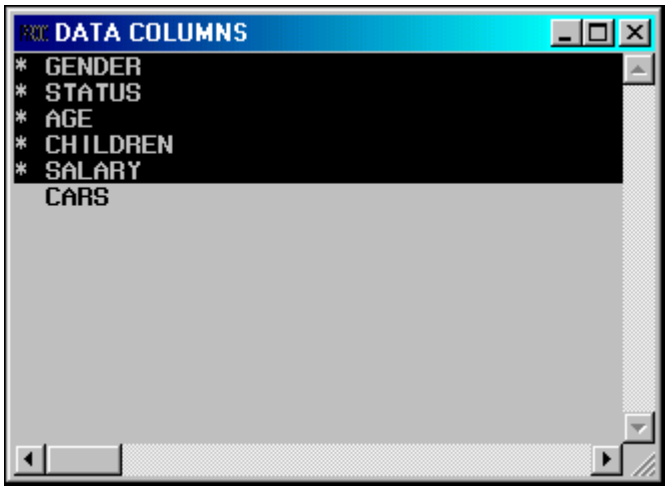
Click 

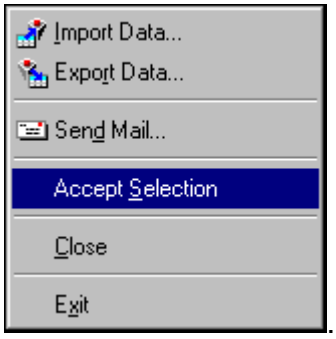
since this is a small table.

The next screen prompts you for the columns to be displayed on the report from left to right.



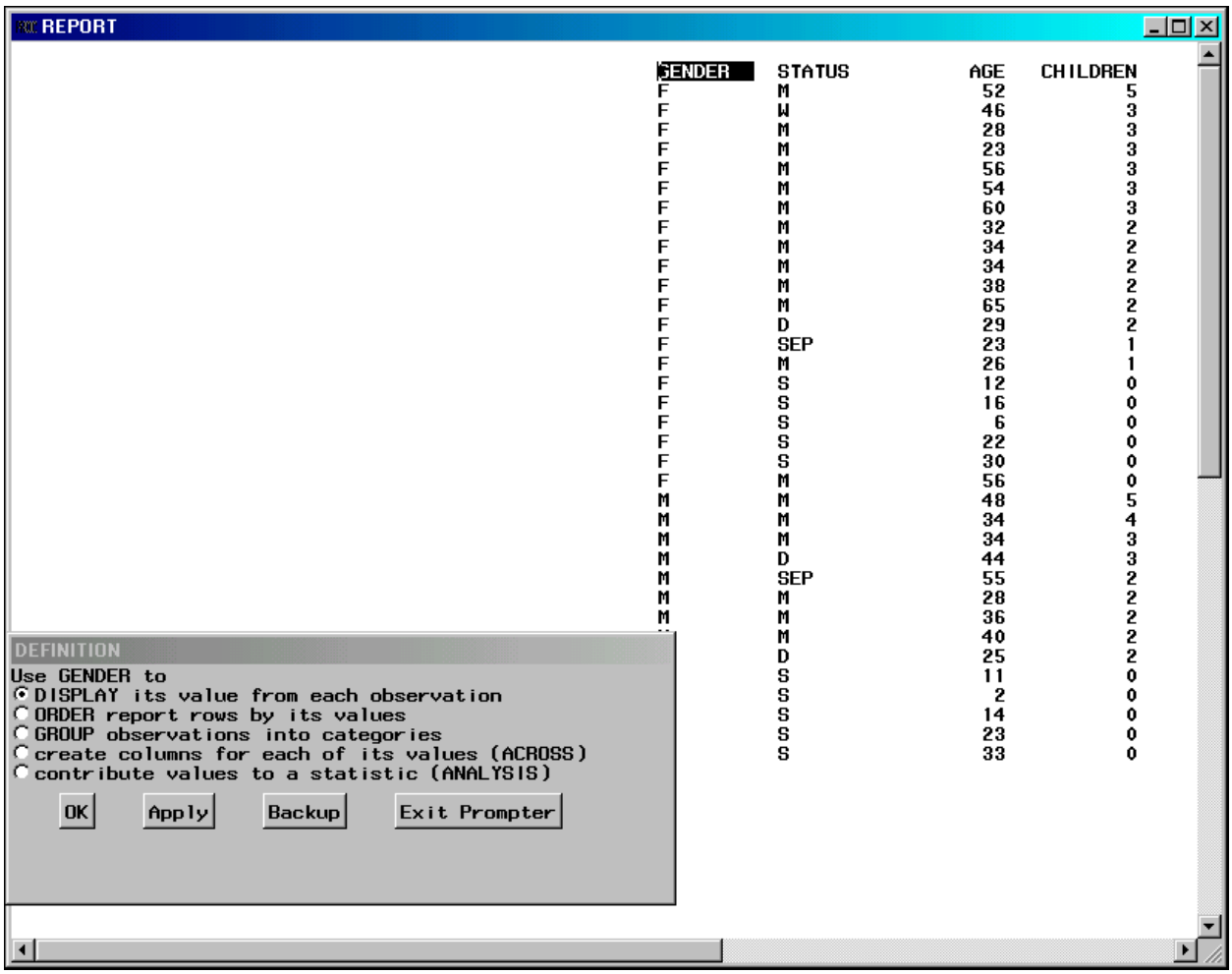
Select the order.



Click on the File pulldown and 

.

The following report is displayed.

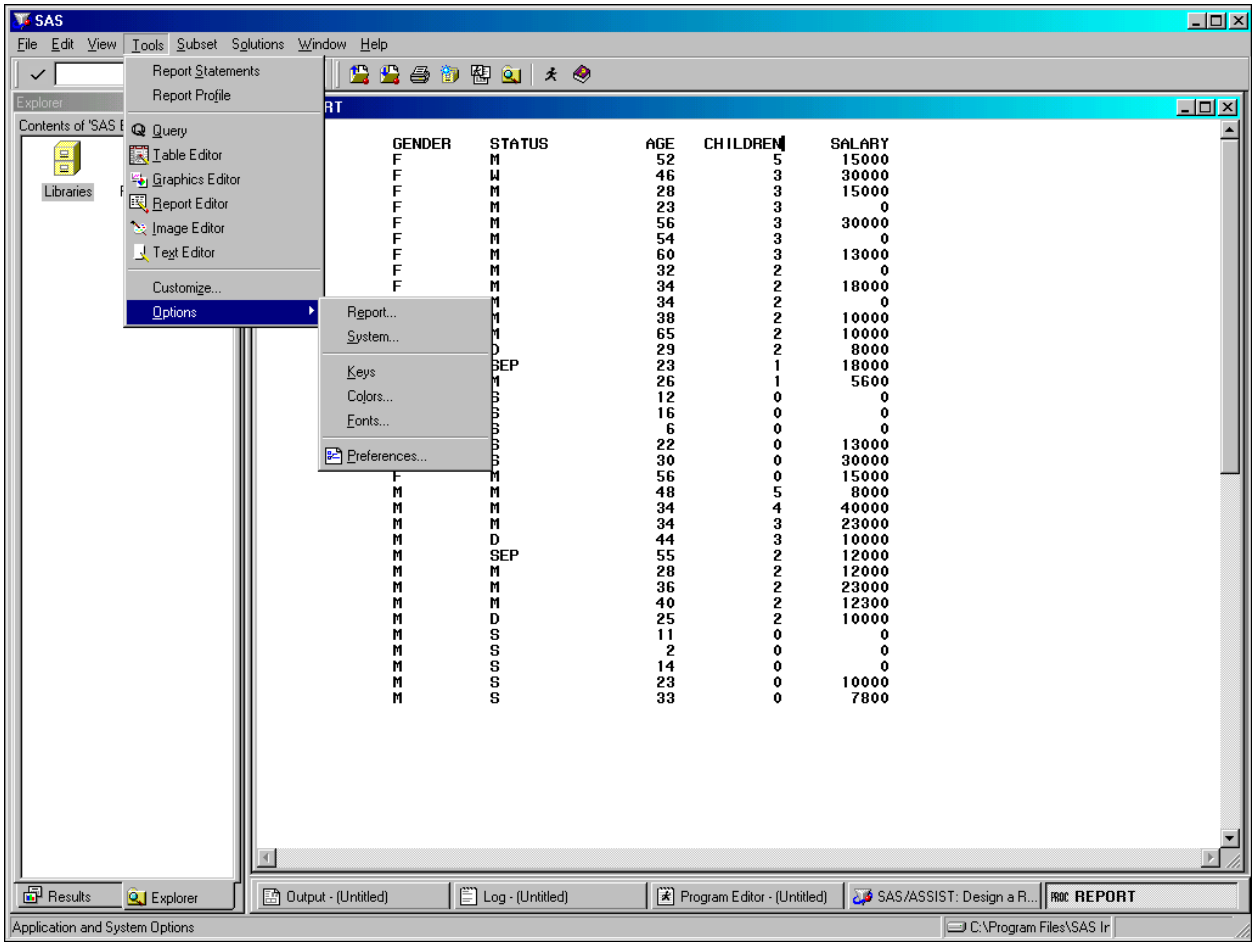


and you are prompted to define each column.

Click  on each of the prompt screens to accept the defaults. Notice character columns are automatically set to DISPLAY while numeric columns are automatically set to ANALYSIS.

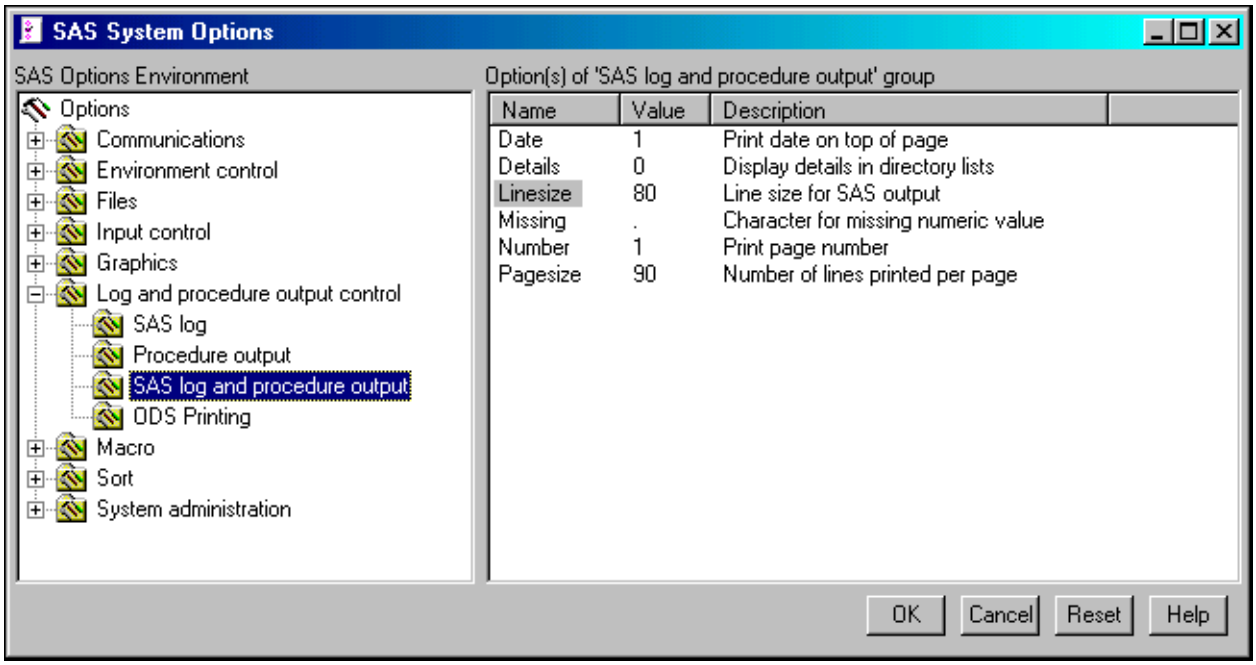
For better display, change the default line size (width) to about 80 for this demonstration.

Click on the Tools pulldown, Options and System.



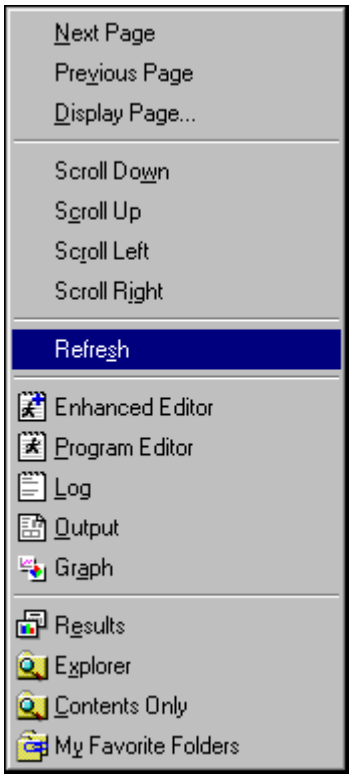
GENDER	STATUS	AGE	CHILDREN	SALARY
F	M	52	5	15000
F	W	46	3	30000
F	M	28	3	15000
F	M	23	3	0
F	M	56	3	30000
F	M	54	3	0
F	M	60	3	13000
F	M	32	2	0
F	M	34	2	18000
	M	34	2	0
	M	38	2	10000
	M	65	2	10000
	D	29	2	8000
	SEP	23	1	18000
	M	26	1	5600
	S	12	0	0
	S	16	0	0
	S	6	0	0
	S	22	0	13000
	S	30	0	30000
F	M	56	0	15000
M	M	48	5	8000
M	M	34	4	40000
M	M	34	3	23000
M	D	44	3	10000
M	SEP	55	2	12000
M	M	28	2	12000
M	M	36	2	23000
M	M	40	2	12300
M	D	25	2	10000
M	S	11	0	0
M	S	2	0	0
M	S	14	0	0
M	S	23	0	10000
M	S	33	0	7800

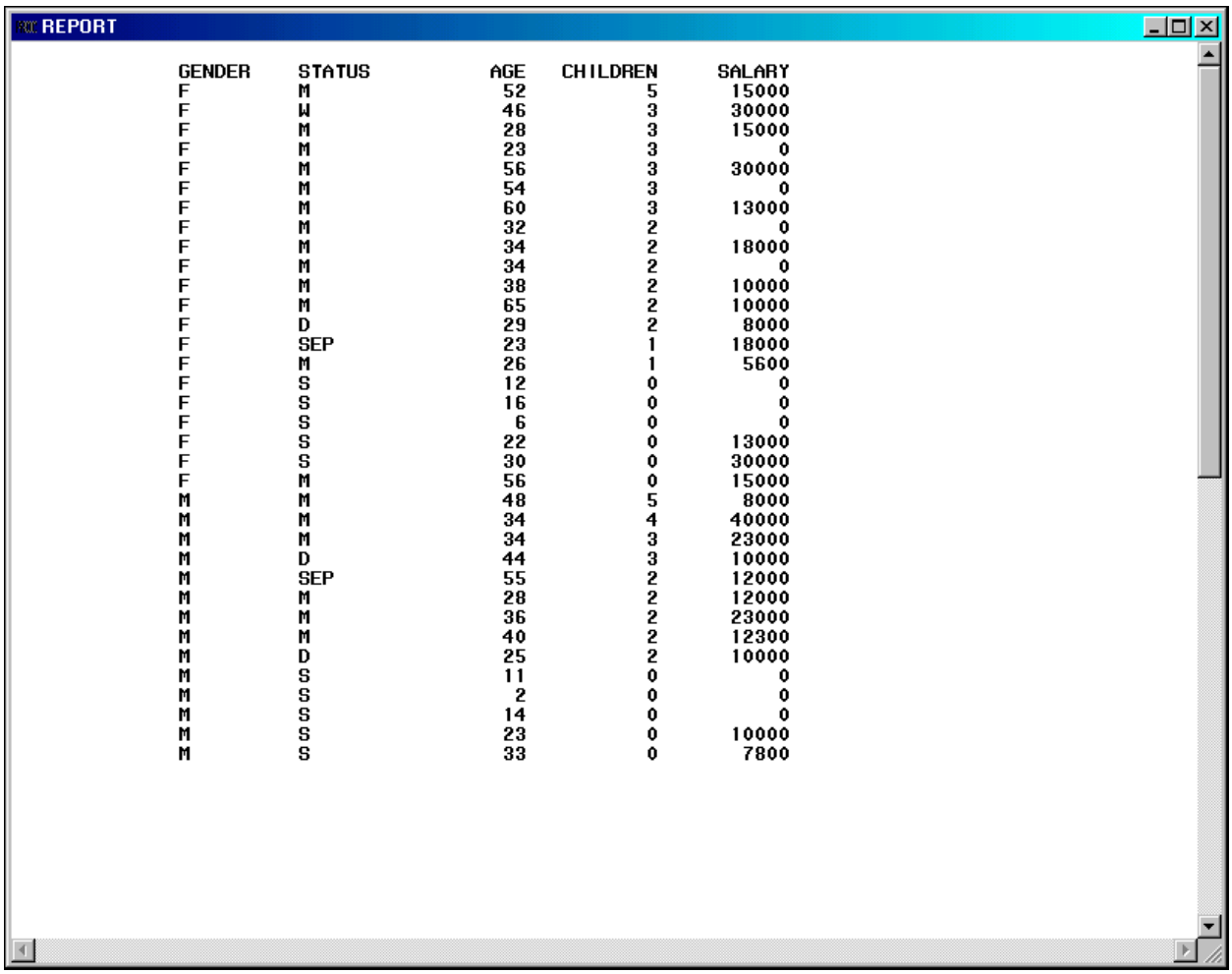
Select SAS Log and Procedure Output.



Double click on Linesize and change it to 80.

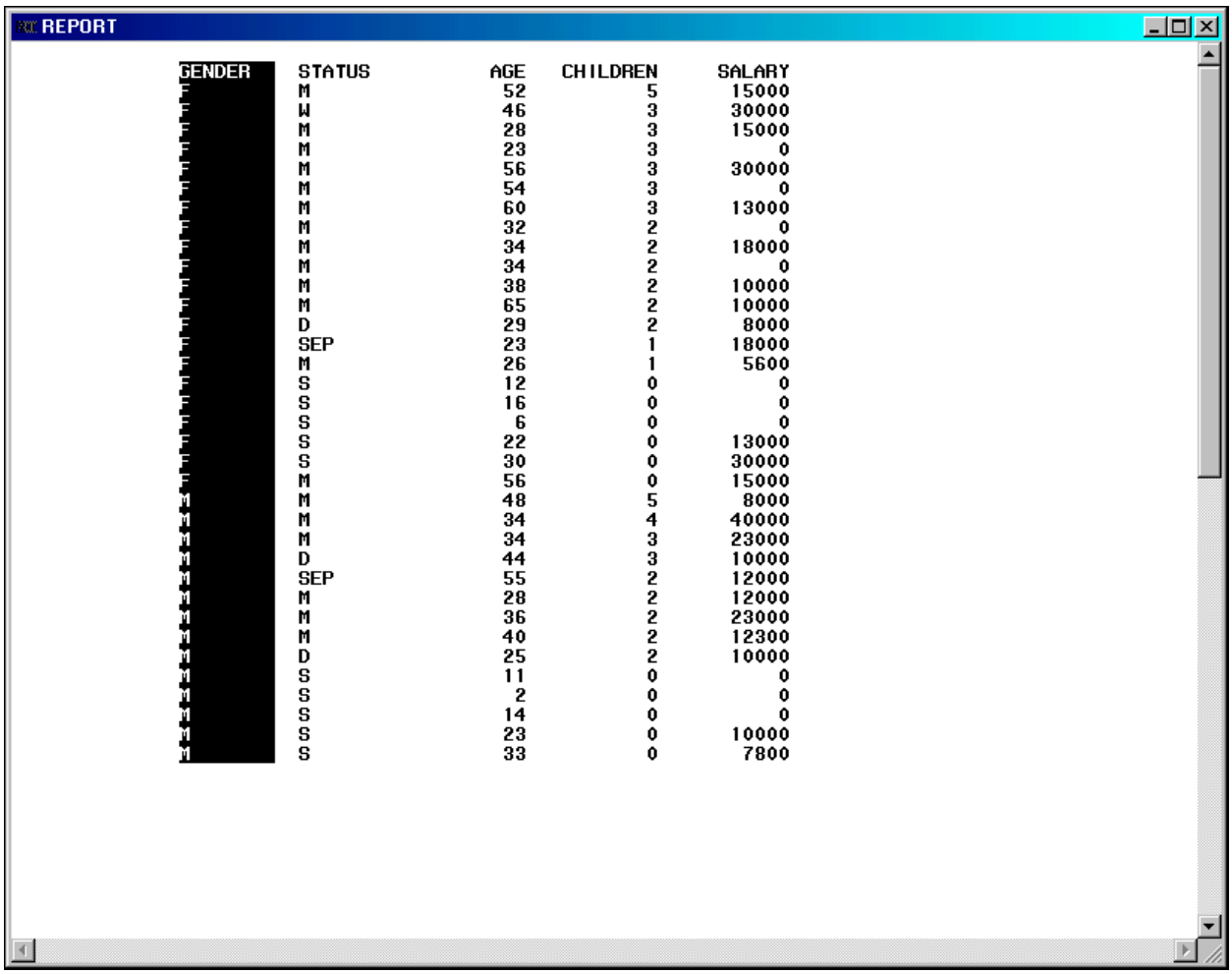
Select  when finished.

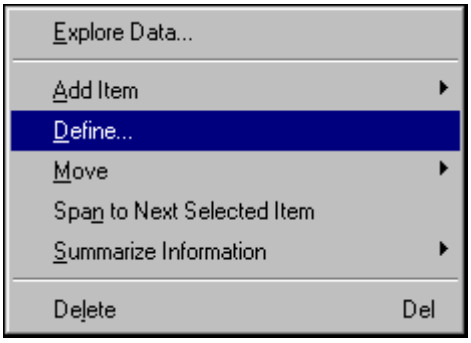
Refresh the report by selecting the View pull down and 



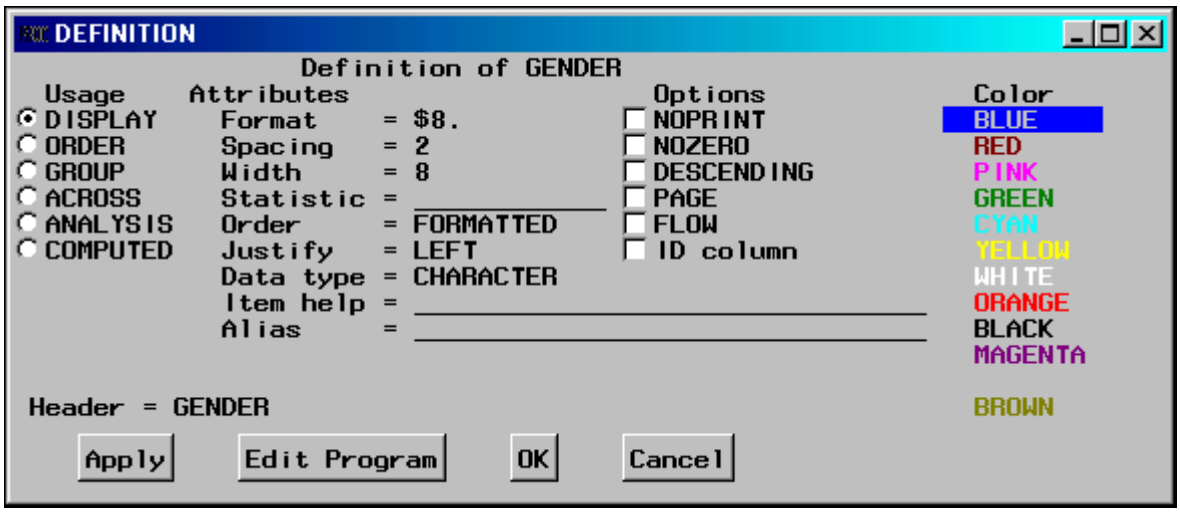
Let’s enhance the report. Let’s order the data and add subtotals and grand totals.

Highlight the Gender column.

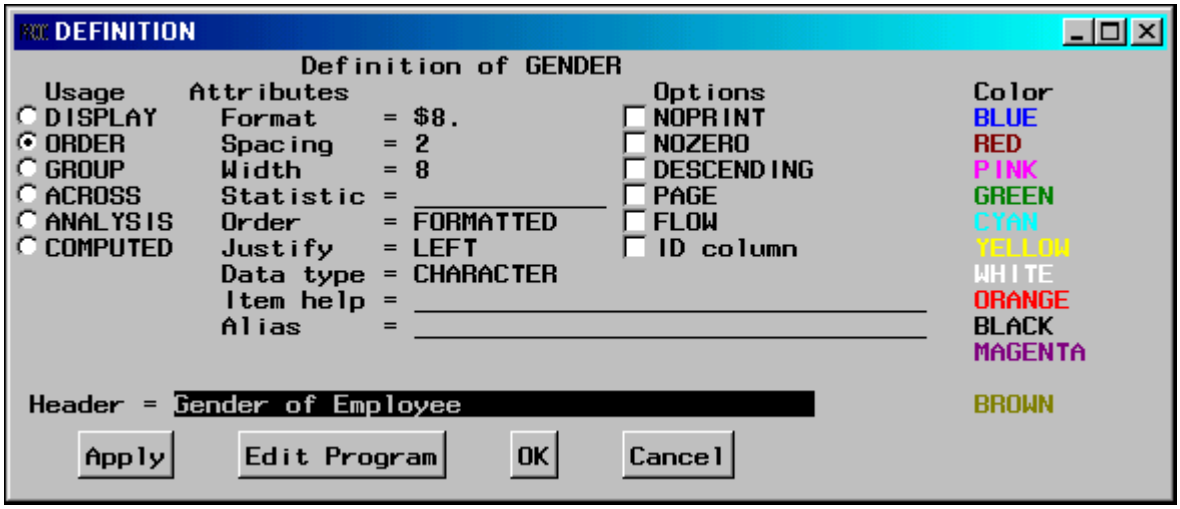


Click the Edit pull down and 

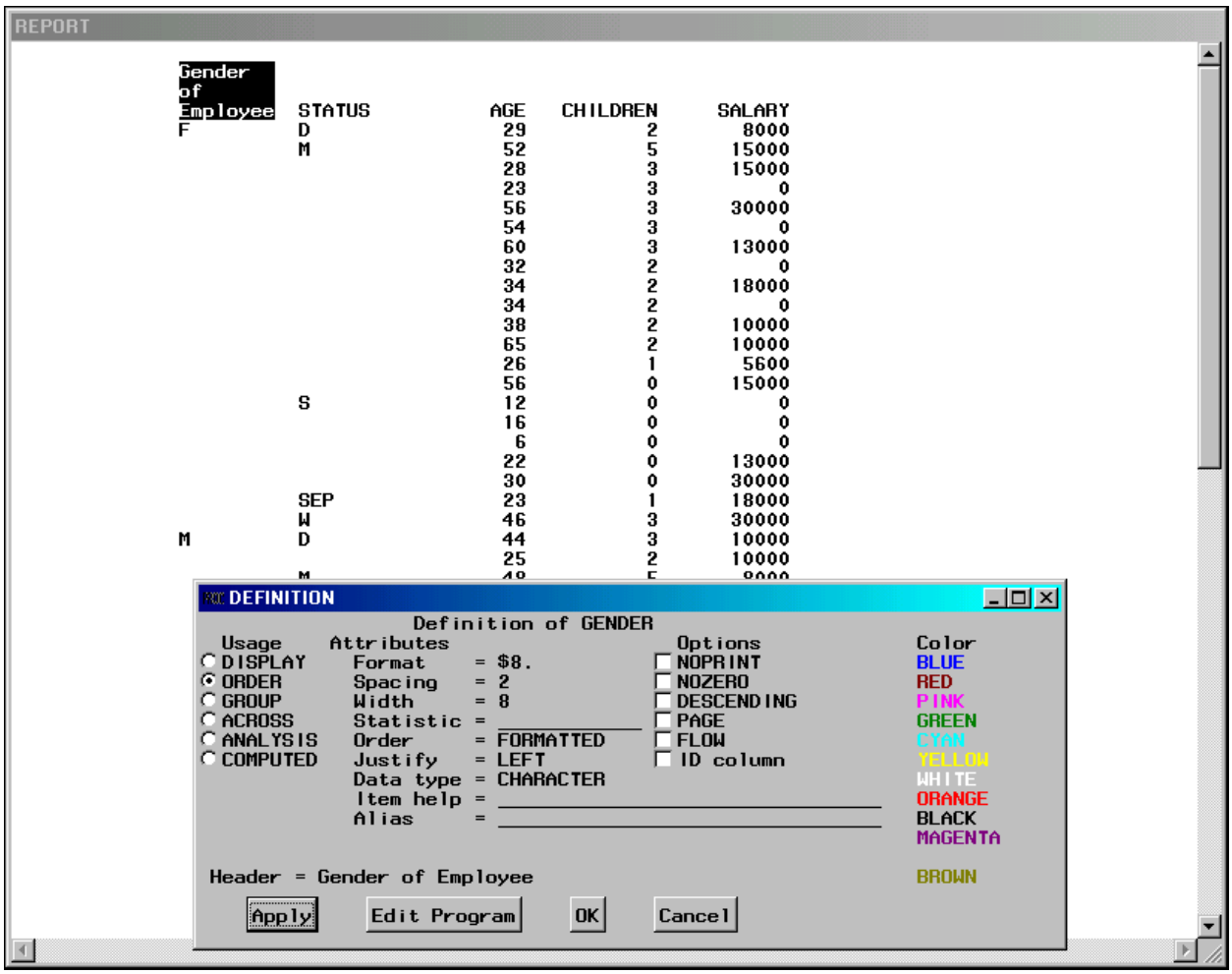
The following screen appears, allowing you to define the highlighted column.



Change it to modify the Usage to be Order and change the Header label.

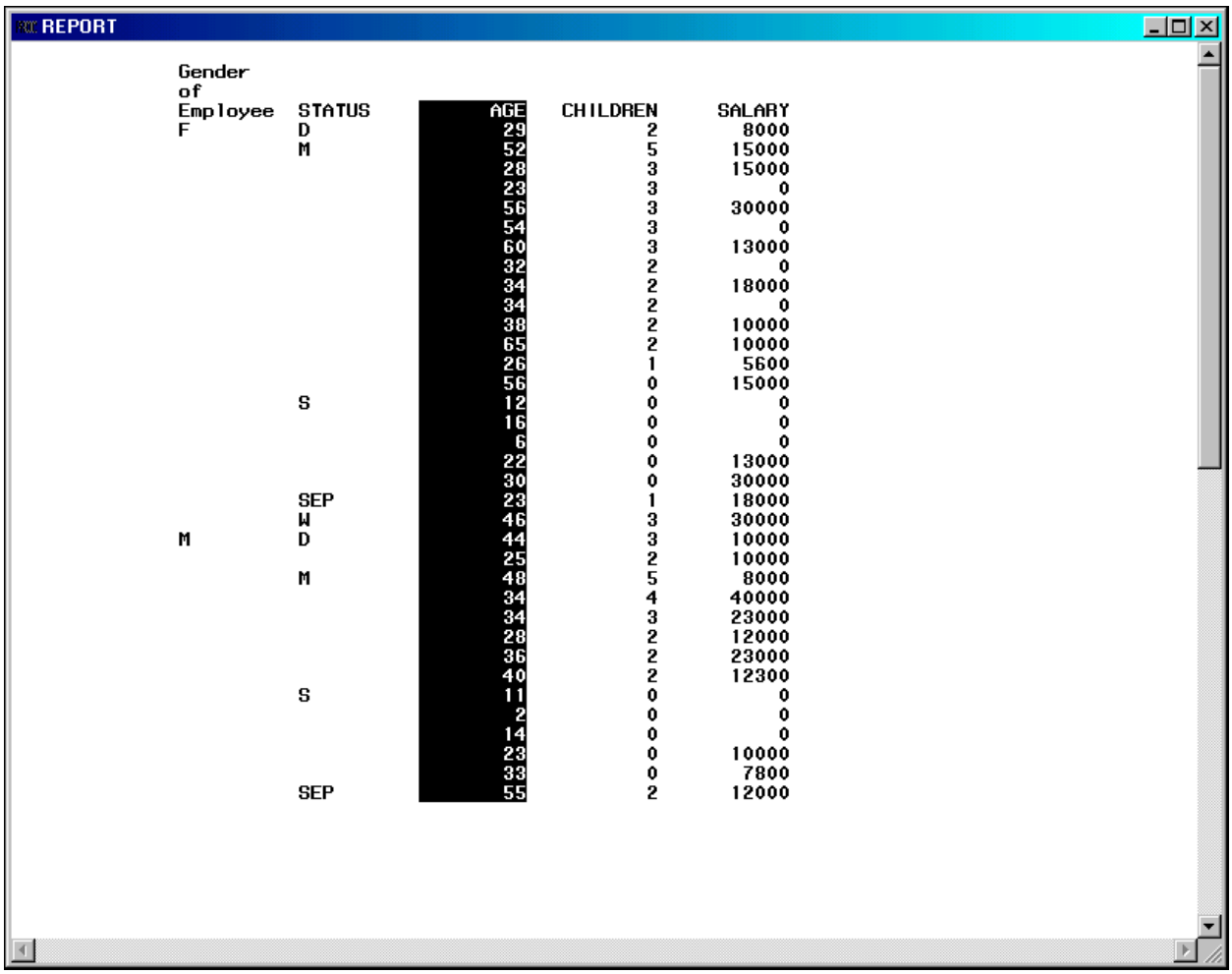


Click Apply to see the results immediately and OK when done.

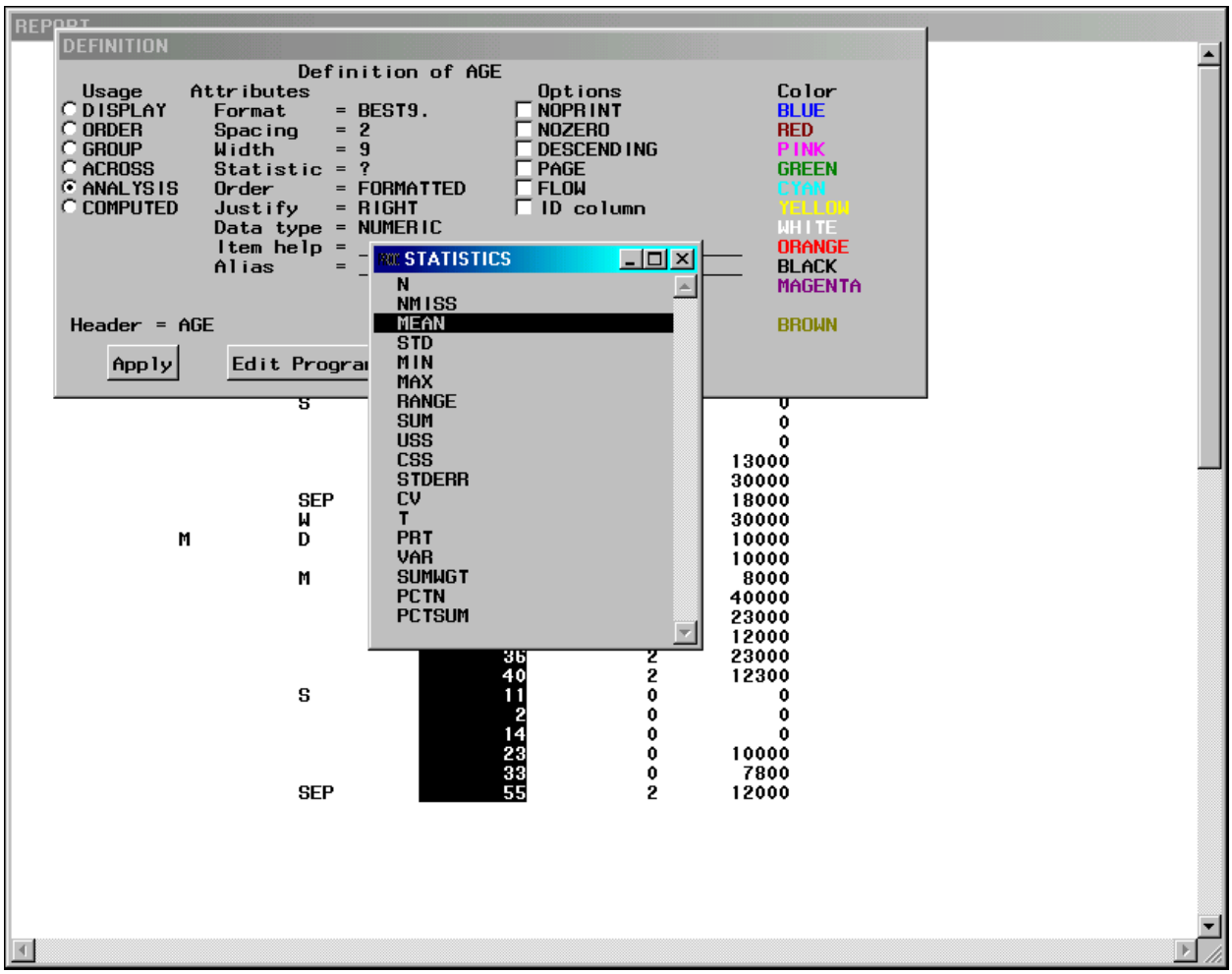


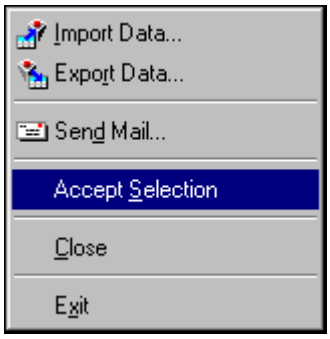
Notice how both GENDER and STATUS were both redefined as Order.

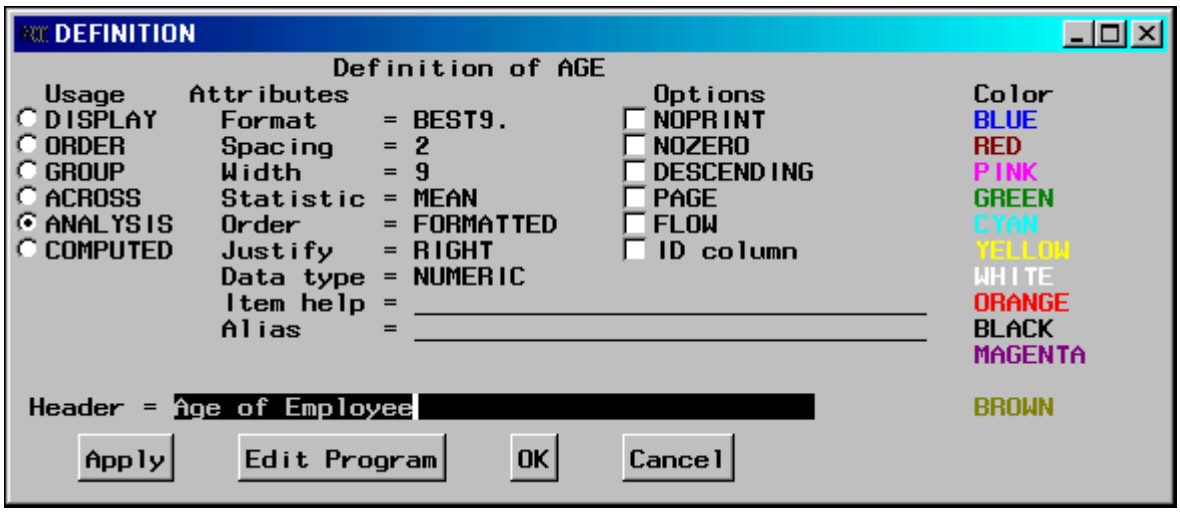
Change the Age column’s default statistic to Mean.



Type a ? in the statistic field to see the possible statistics.

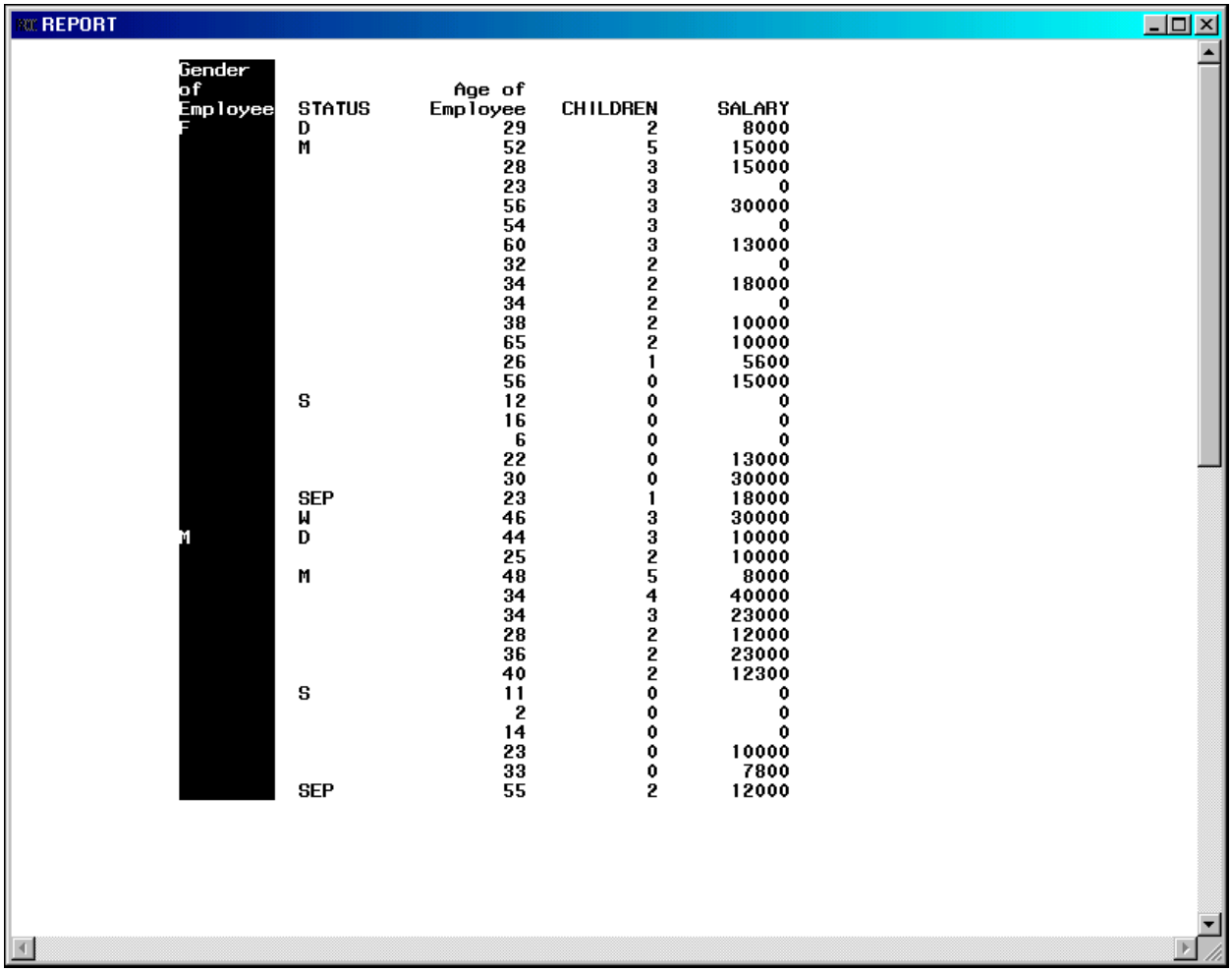


Select Mean, the File pull down,  and change the header label.



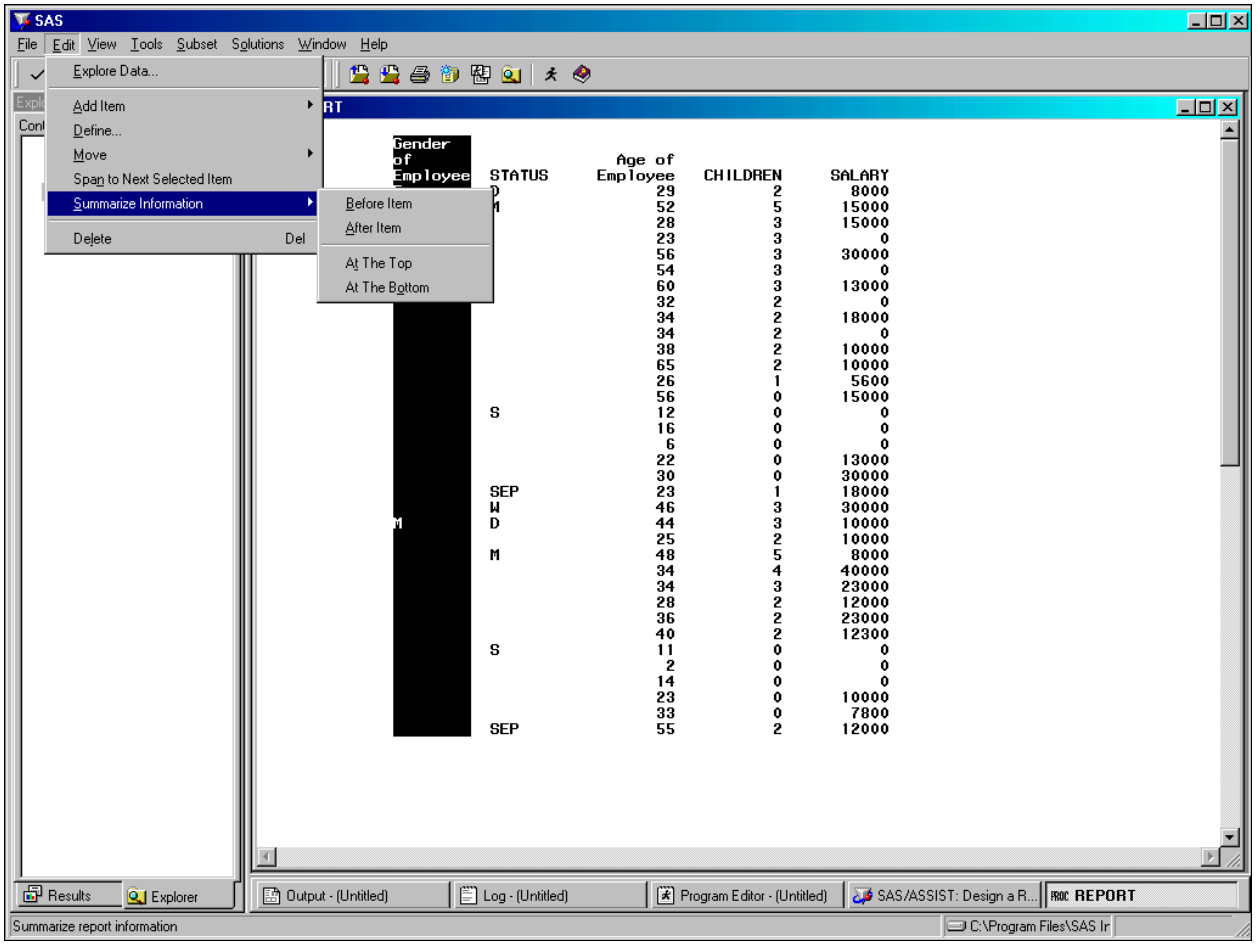
Click  when finished.

Now, let's add subtotals to the Gender Level. Highlight the Gender column.

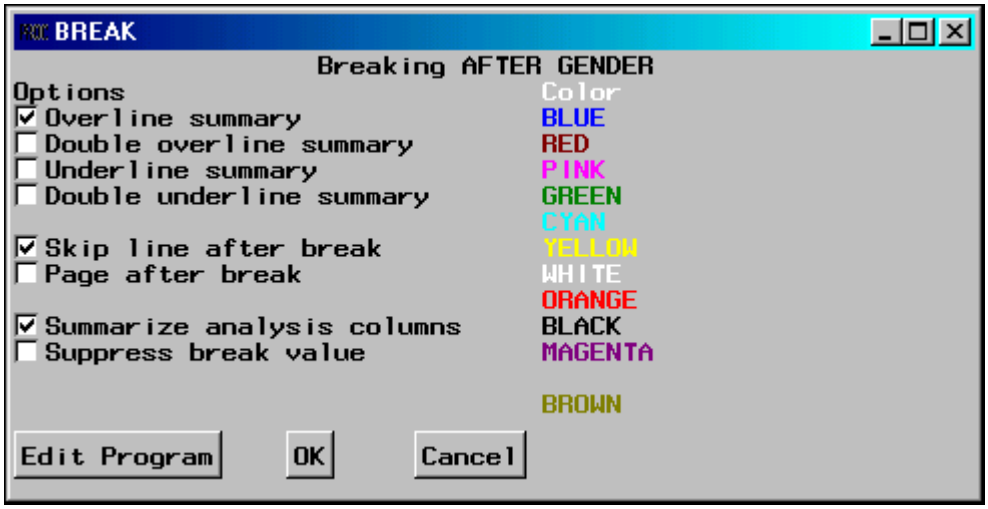


Gender of Employee	STATUS	Age of Employee	CHILDREN	SALARY
F	D	29	2	8000
	M	52	5	15000
		28	3	15000
		23	3	0
		56	3	30000
		54	3	0
		60	3	13000
		32	2	0
		34	2	18000
		34	2	0
		38	2	10000
		65	2	10000
		26	1	5600
		56	0	15000
	S	12	0	0
		16	0	0
		6	0	0
		22	0	13000
		30	0	30000
	SEP	23	1	18000
	W	46	3	30000
M	D	44	3	10000
		25	2	10000
	M	48	5	8000
		34	4	40000
		34	3	23000
		28	2	12000
		36	2	23000
		40	2	12300
	S	11	0	0
		2	0	0
		14	0	0
		23	0	10000
		33	0	7800
	SEP	55	2	12000

Select the Edit pull down, Summarize information, After Item.



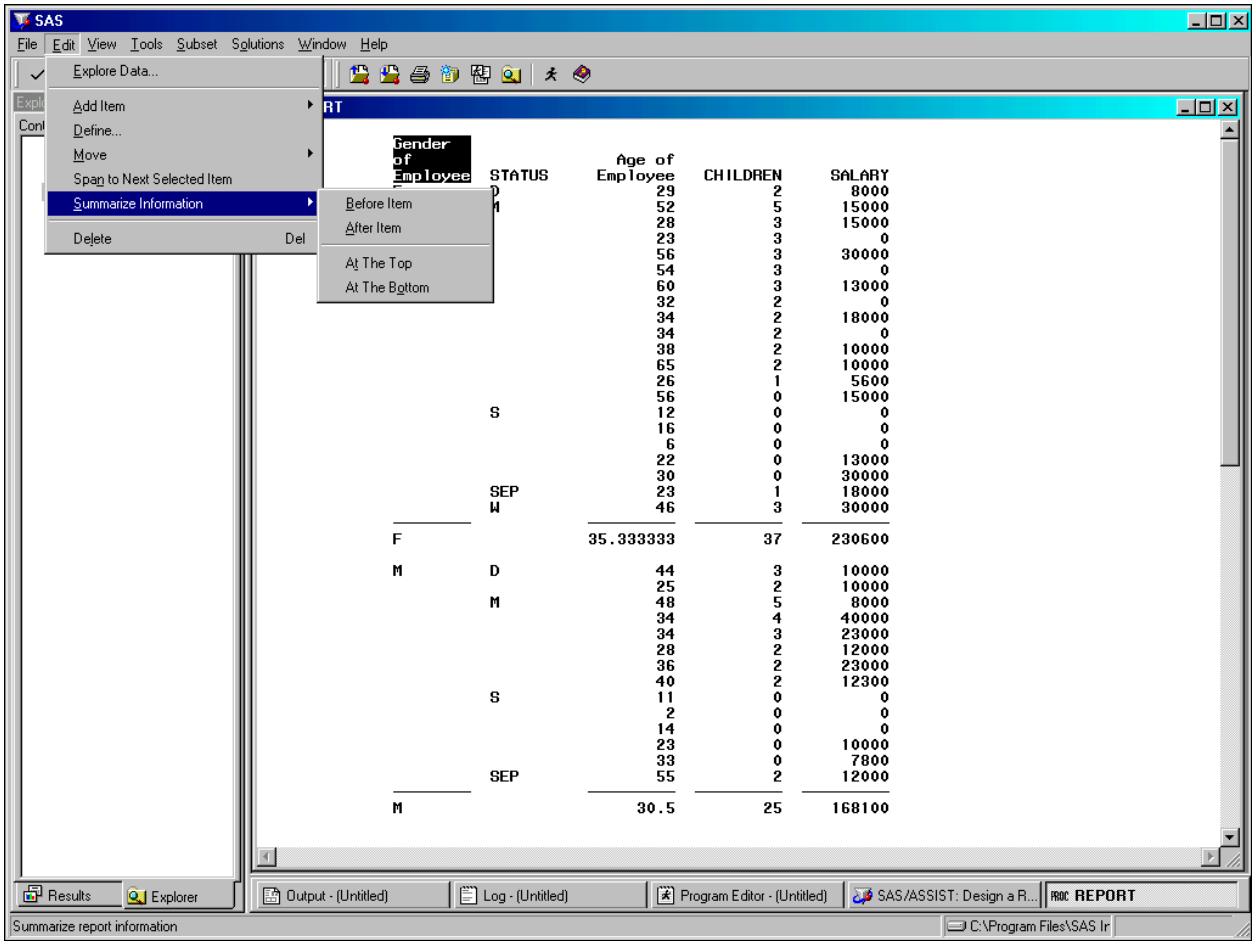
Change the following screen to specify how to break, which request summarization, an overline and to skip a line after the subtotal.



Click  when finished.

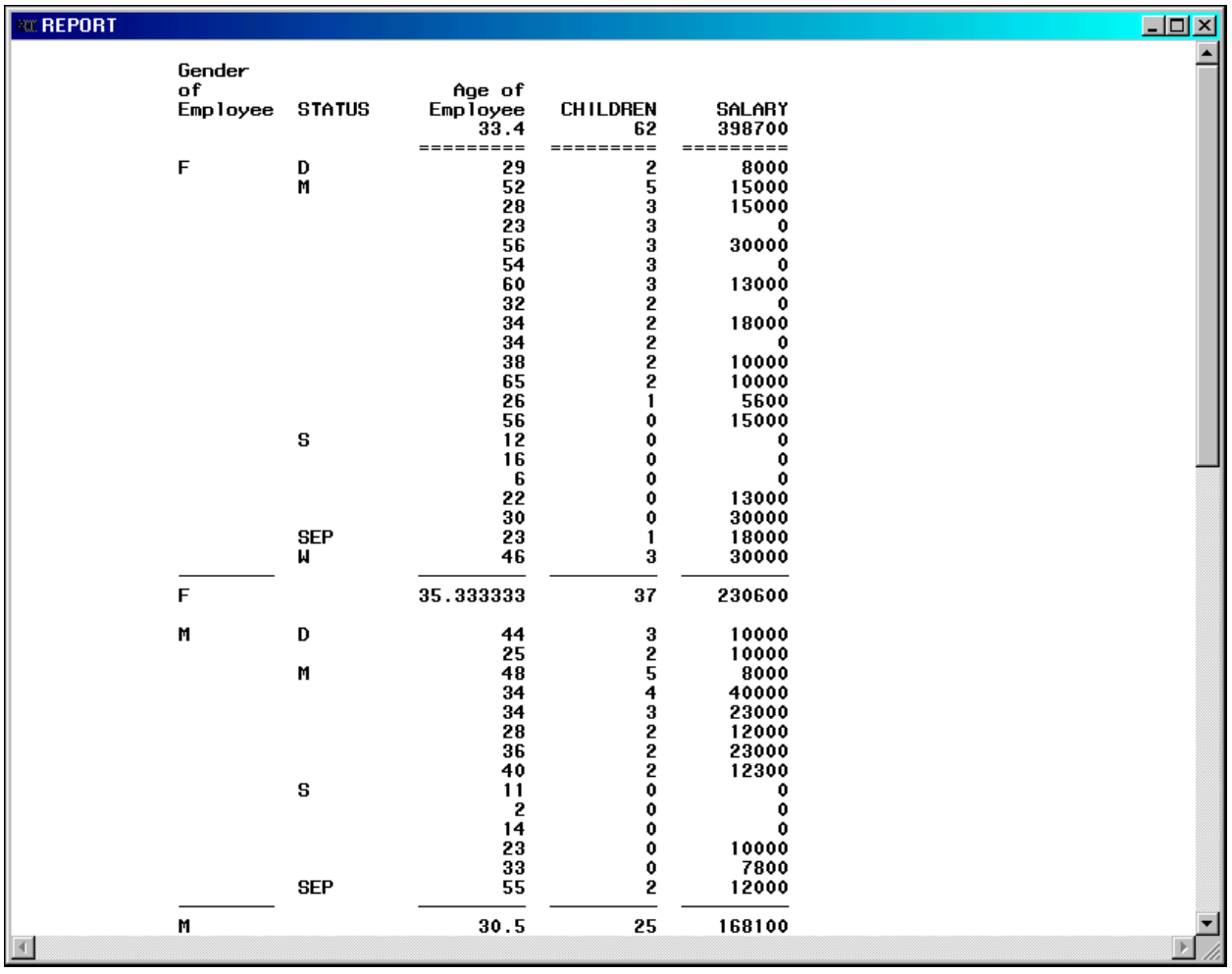
Let’s add a grand total.

Select the Edit pull down and Summarize information At The Top.



Modify the screen to look like the following.

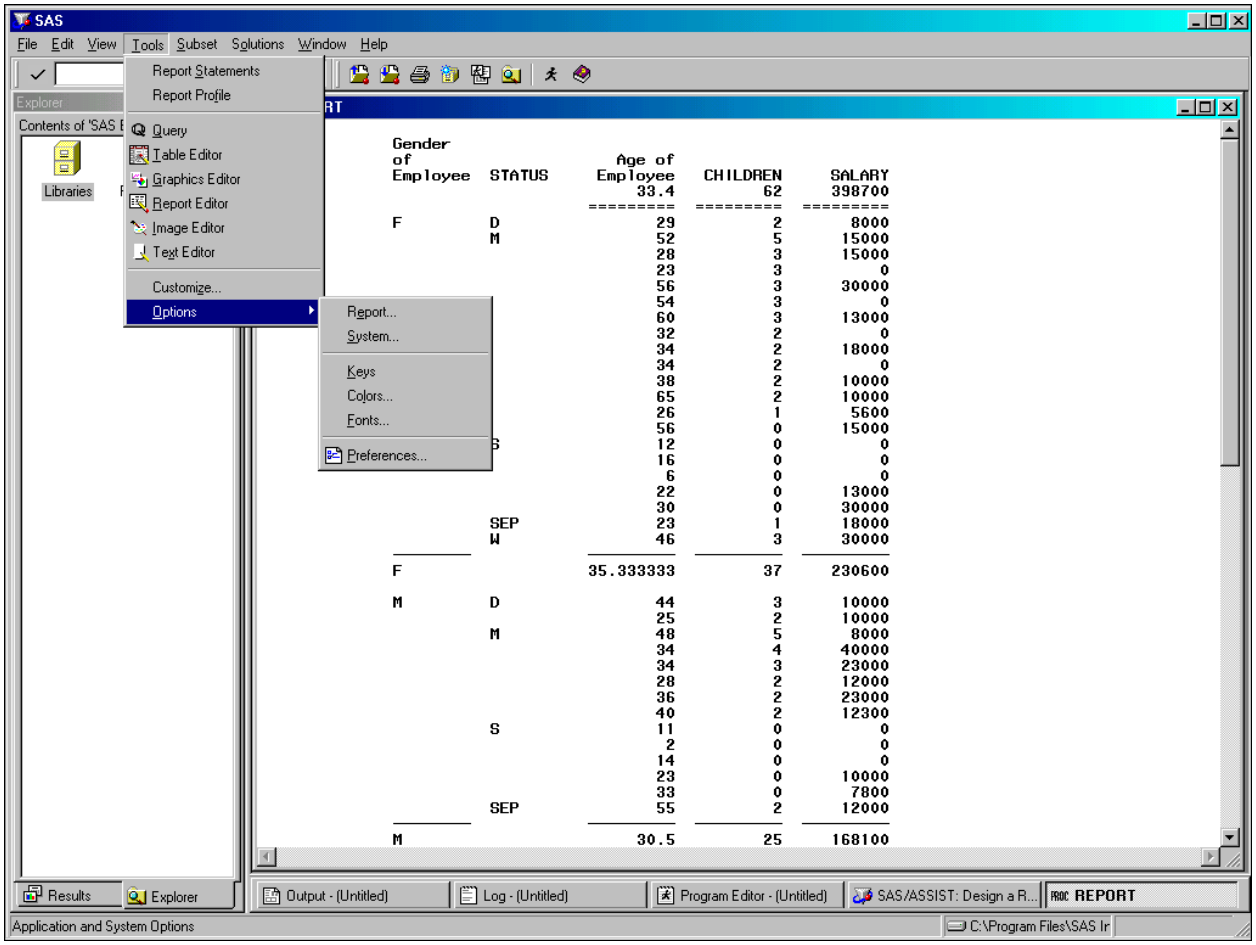
Click OK when finished.



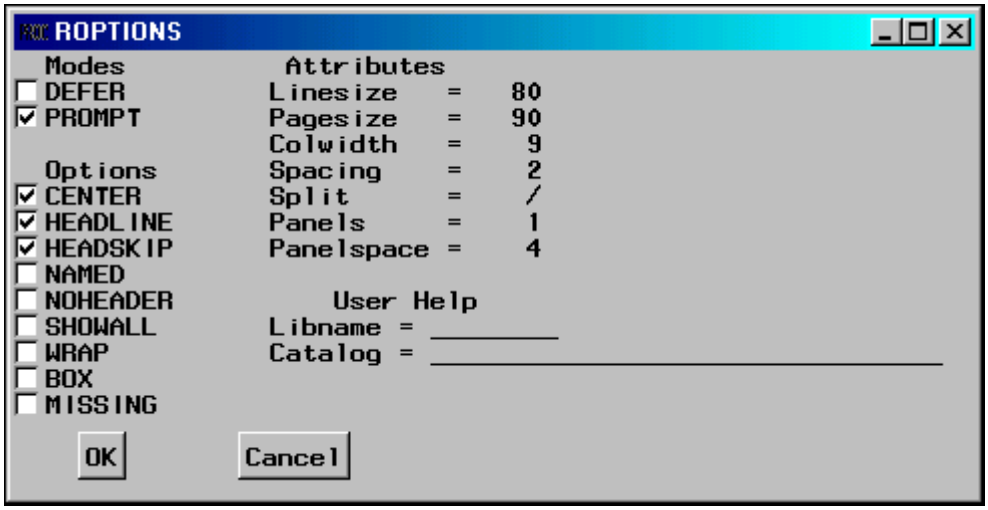
Gender of Employee	STATUS	Age of Employee	CHILDREN	SALARY
		33.4	62	398700
F	D	29	2	8000
	M	52	5	15000
		28	3	15000
		23	3	0
		56	3	30000
		54	3	0
		60	3	13000
		32	2	0
		34	2	18000
		34	2	0
		38	2	10000
		65	2	10000
		26	1	5600
		56	0	15000
	S	12	0	0
		16	0	0
		6	0	0
		22	0	13000
		30	0	30000
	SEP	23	1	18000
	W	46	3	30000
F		35.333333	37	230600
M	D	44	3	10000
		25	2	10000
	M	48	5	8000
		34	4	40000
		34	3	23000
		28	2	12000
		36	2	23000
		40	2	12300
	S	11	0	0
		2	0	0
		14	0	0
		23	0	10000
		33	0	7800
	SEP	55	2	12000
M		30.5	25	168100

Let’s add some final touches, including titles and headers.

Select the Tools pull down, Options, Report.



Select Headline and Headskip.

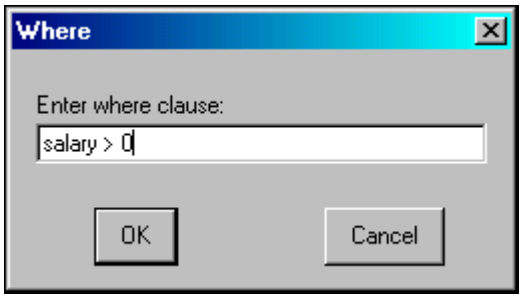


Click OK when finished.

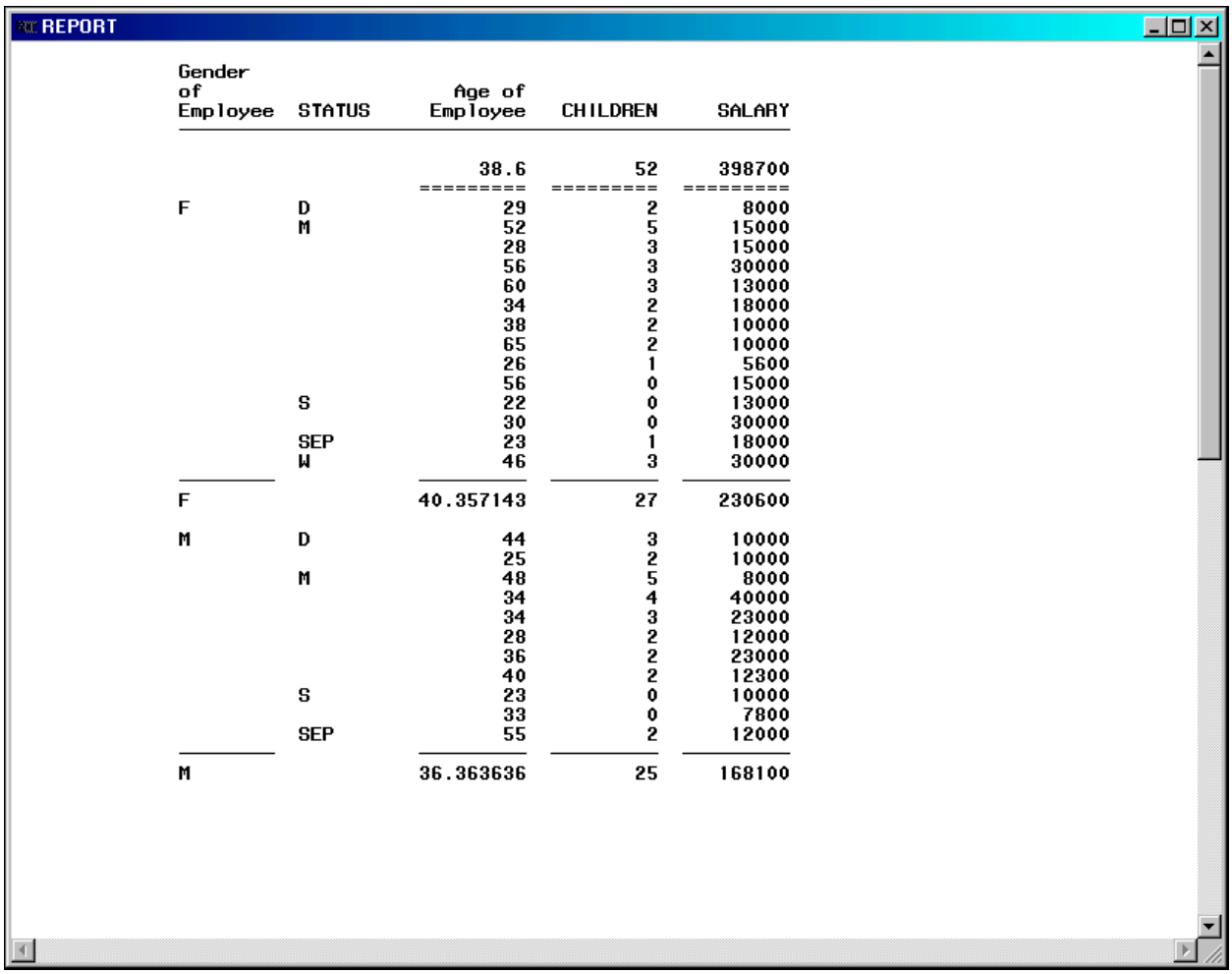
REPORT

Gender of Employee	STATUS	Age of Employee	CHILDREN	SALARY
		33.4	62	398700
F	D	29	2	8000
	M	52	5	15000
		28	3	15000
		23	3	0
		56	3	30000
		54	3	0
		60	3	13000
		32	2	0
		34	2	18000
		34	2	0
		38	2	10000
		65	2	10000
		26	1	5600
		56	0	15000
	S	12	0	0
		16	0	0
		6	0	0
		22	0	13000
		30	0	30000
	SEP	23	1	18000
	W	46	3	30000
F		35.333333	37	230600
M	D	44	3	10000
		25	2	10000
	M	48	5	8000
		34	4	40000
		34	3	23000
		28	2	12000
		36	2	23000
		40	2	12300
	S	11	0	0
		2	0	0
		14	0	0
		23	0	10000
		33	0	7800
	SEP	55	2	12000

Let's subset the report. Select the Subset pull down and



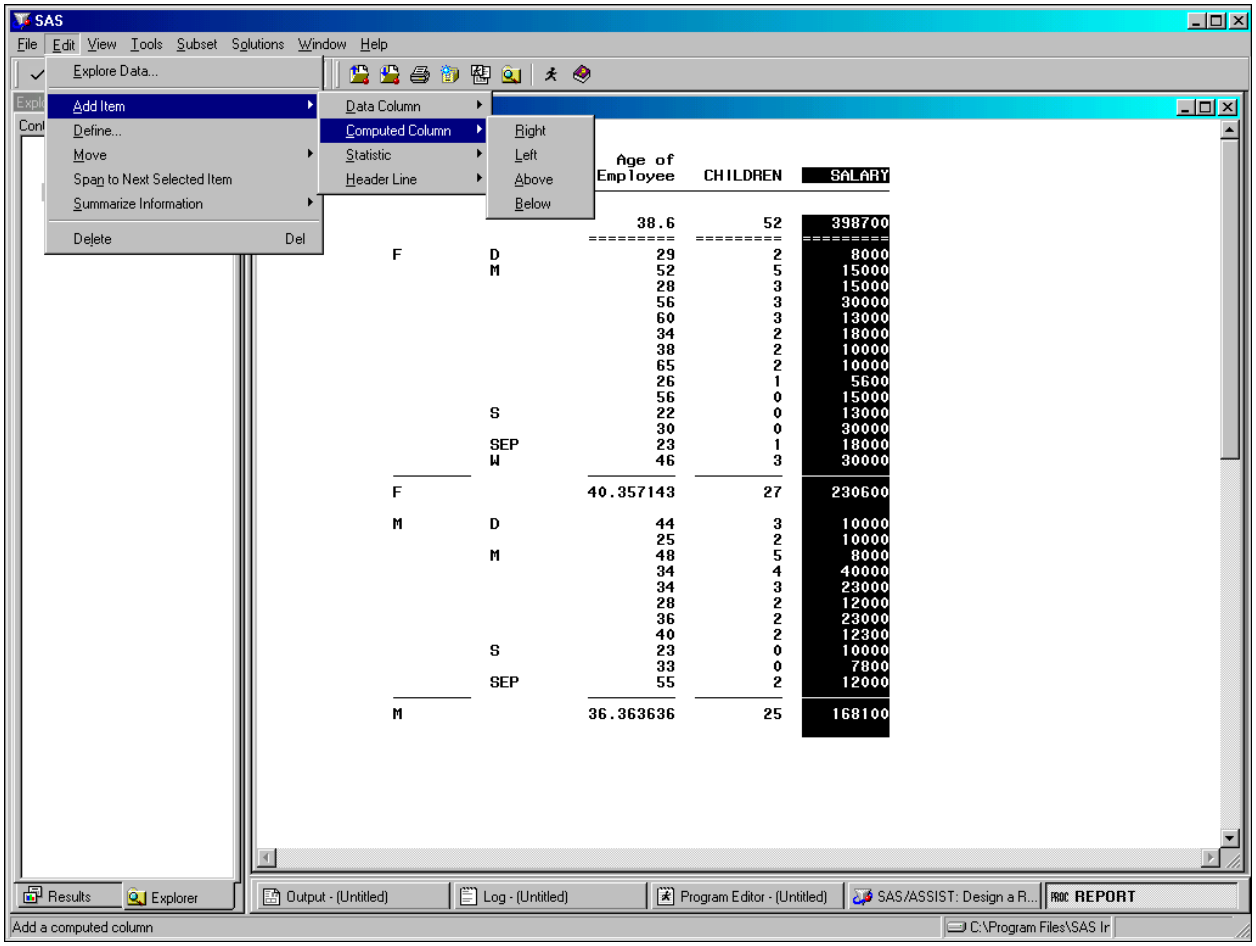
Click 



Gender of Employee	STATUS	Age of Employee	CHILDREN	SALARY
		38.6	52	398700
F	D	29	2	8000
	M	52	5	15000
		28	3	15000
		56	3	30000
		60	3	13000
		34	2	18000
		38	2	10000
		65	2	10000
		26	1	5600
		56	0	15000
	S	22	0	13000
		30	0	30000
	SEP	23	1	18000
	W	46	3	30000
F		40.357143	27	230600
M	D	44	3	10000
		25	2	10000
	M	48	5	8000
		34	4	40000
		34	3	23000
		28	2	12000
		36	2	23000
		40	2	12300
	S	23	0	10000
		33	0	7800
	SEP	55	2	12000
M		36.363636	25	168100

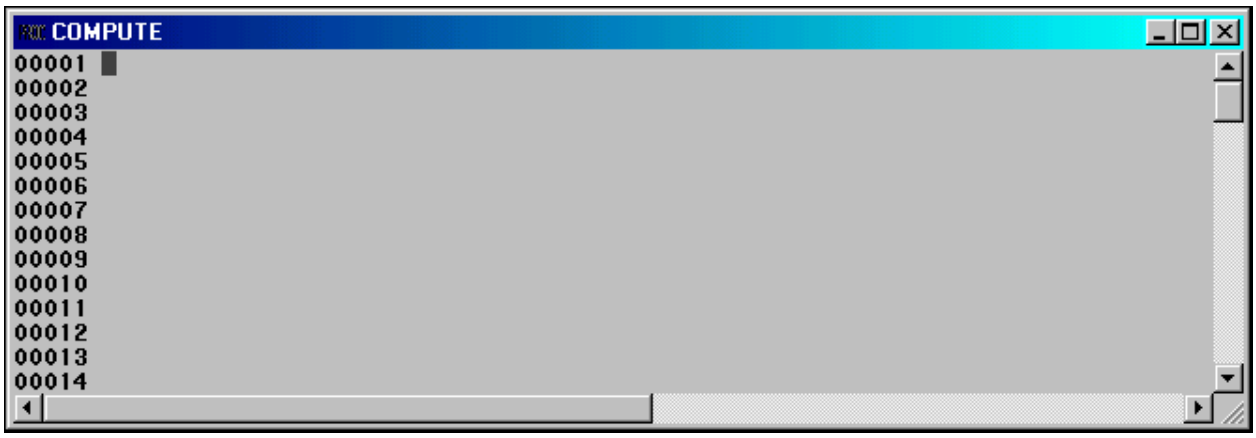
Let's add a computed column to the right (very important) of the existing data. We will add a new column called RAISE. It will use SALARY, which will be printed to the left. SALARY must be available so RAISE can use it in a calculation.

Highlight Salary, select the Edit pull down, Add Item, Computed, Right.

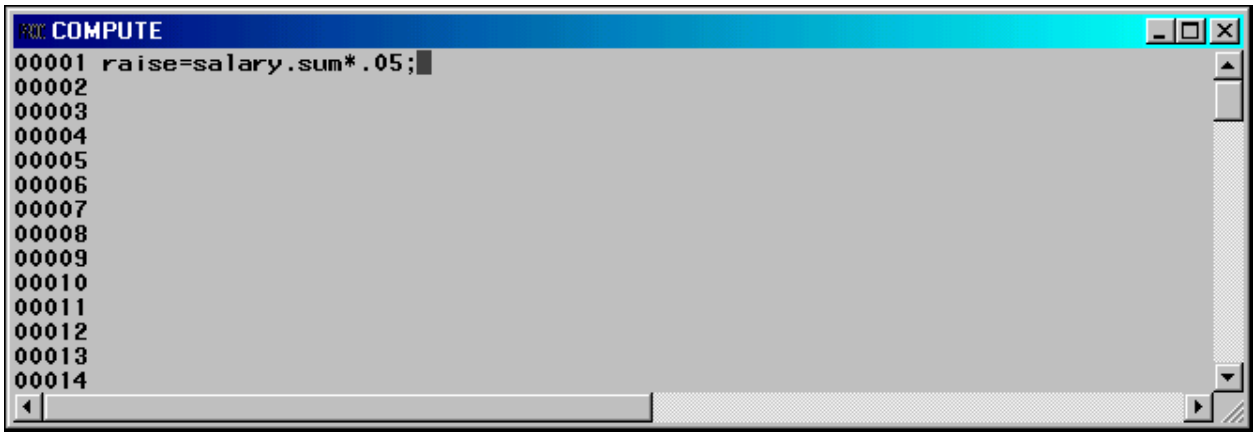


Type in the name of RAISE.

Click Edit Program to see



Type in the calculation.

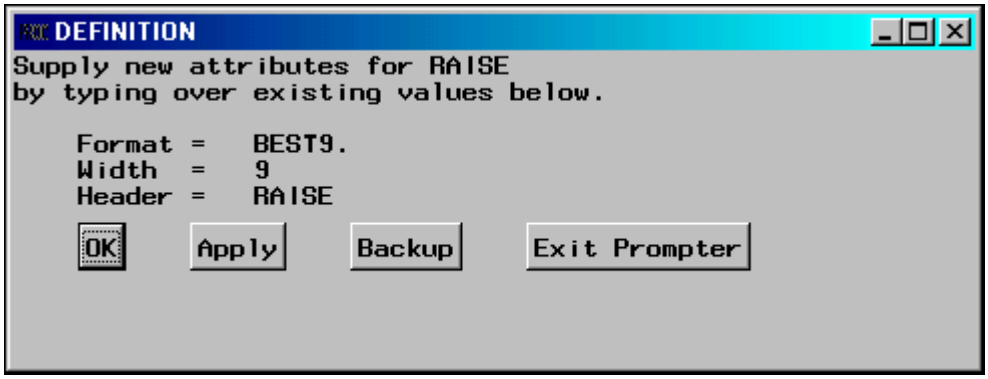


Remember, SALARY is really known as SALARY.SUM, because the statistic has been applied to the column on the report.

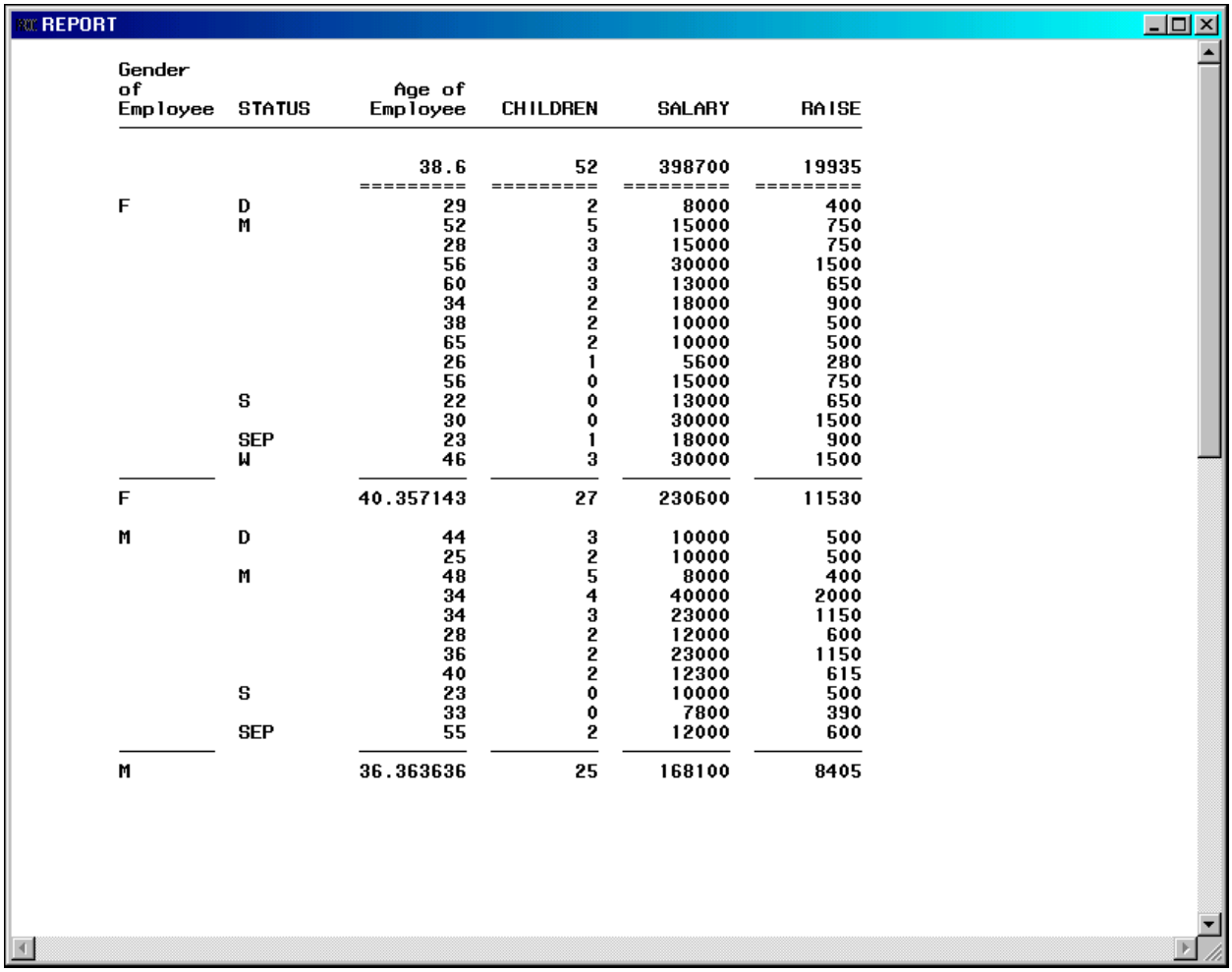
Close the window and click 

.

Define the column as necessary.



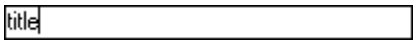
Click  and the following is displayed.

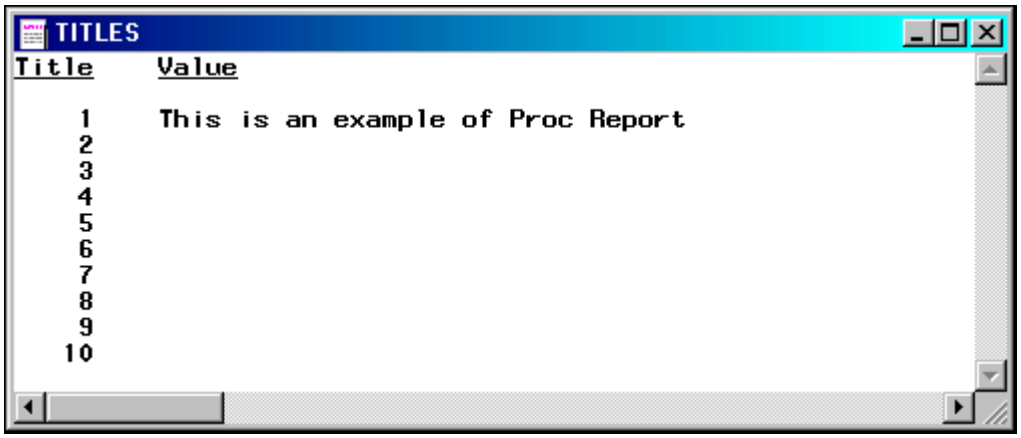


REPORT

Gender of Employee	STATUS	Age of Employee	CHILDREN	SALARY	RAISE
		38.6	52	398700	19935
F	D	29	2	8000	400
	M	52	5	15000	750
		28	3	15000	750
		56	3	30000	1500
		60	3	13000	650
		34	2	18000	900
		38	2	10000	500
		65	2	10000	500
		26	1	5600	280
		56	0	15000	750
	S	22	0	13000	650
		30	0	30000	1500
	SEP	23	1	18000	900
	W	46	3	30000	1500
F		40.357143	27	230600	11530
M	D	44	3	10000	500
		25	2	10000	500
	M	48	5	8000	400
		34	4	40000	2000
		34	3	23000	1150
		28	2	12000	600
		36	2	23000	1150
		40	2	12300	615
	S	23	0	10000	500
		33	0	7800	390
	SEP	55	2	12000	600
M		36.363636	25	168100	8405

Add a title by opening up the title window.

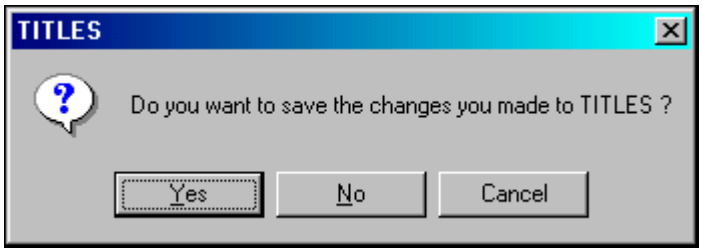
Type  in the command box.



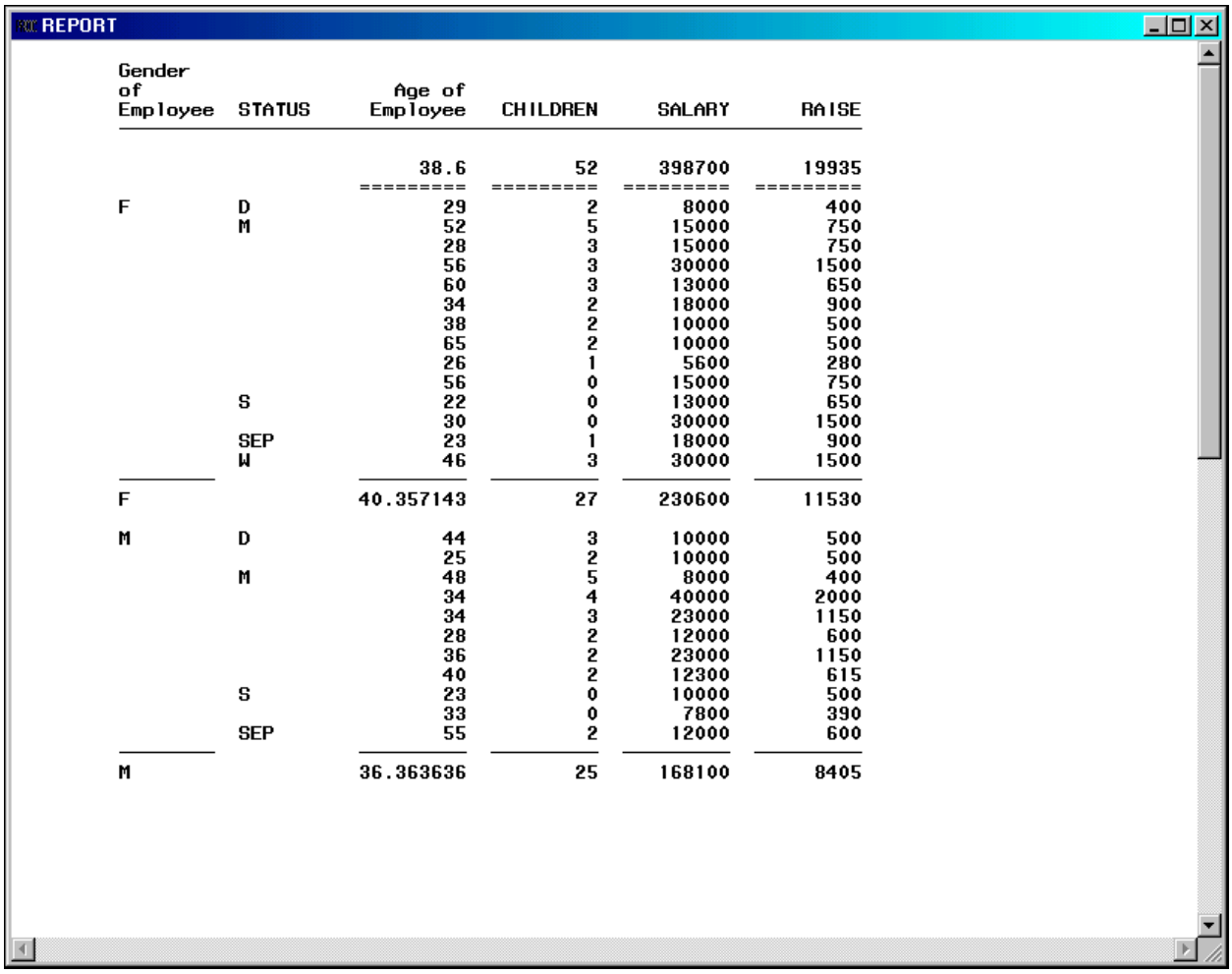
TITLES

Title	Value
1	This is an example of Proc Report
2	
3	
4	
5	
6	
7	
8	
9	
10	

Modify as necessary and close.

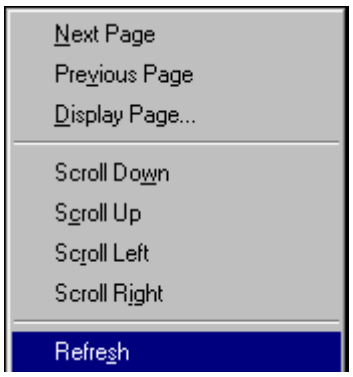


Click 



Gender of Employee	STATUS	Age of Employee	CHILDREN	SALARY	RAISE
		38.6	52	398700	19935
F	D	29	2	8000	400
	M	52	5	15000	750
		28	3	15000	750
		56	3	30000	1500
		60	3	13000	650
		34	2	18000	900
		38	2	10000	500
		65	2	10000	500
		26	1	5600	280
		56	0	15000	750
	S	22	0	13000	650
		30	0	30000	1500
	SEP	23	1	18000	900
	W	46	3	30000	1500
F		40.357143	27	230600	11530
M	D	44	3	10000	500
		25	2	10000	500
	M	48	5	8000	400
		34	4	40000	2000
		34	3	23000	1150
		28	2	12000	600
		36	2	23000	1150
		40	2	12300	615
	S	23	0	10000	500
		33	0	7800	390
	SEP	55	2	12000	600
M		36.363636	25	168100	8405

Titles don’t appear until the display is refreshed. Select the View pull down and



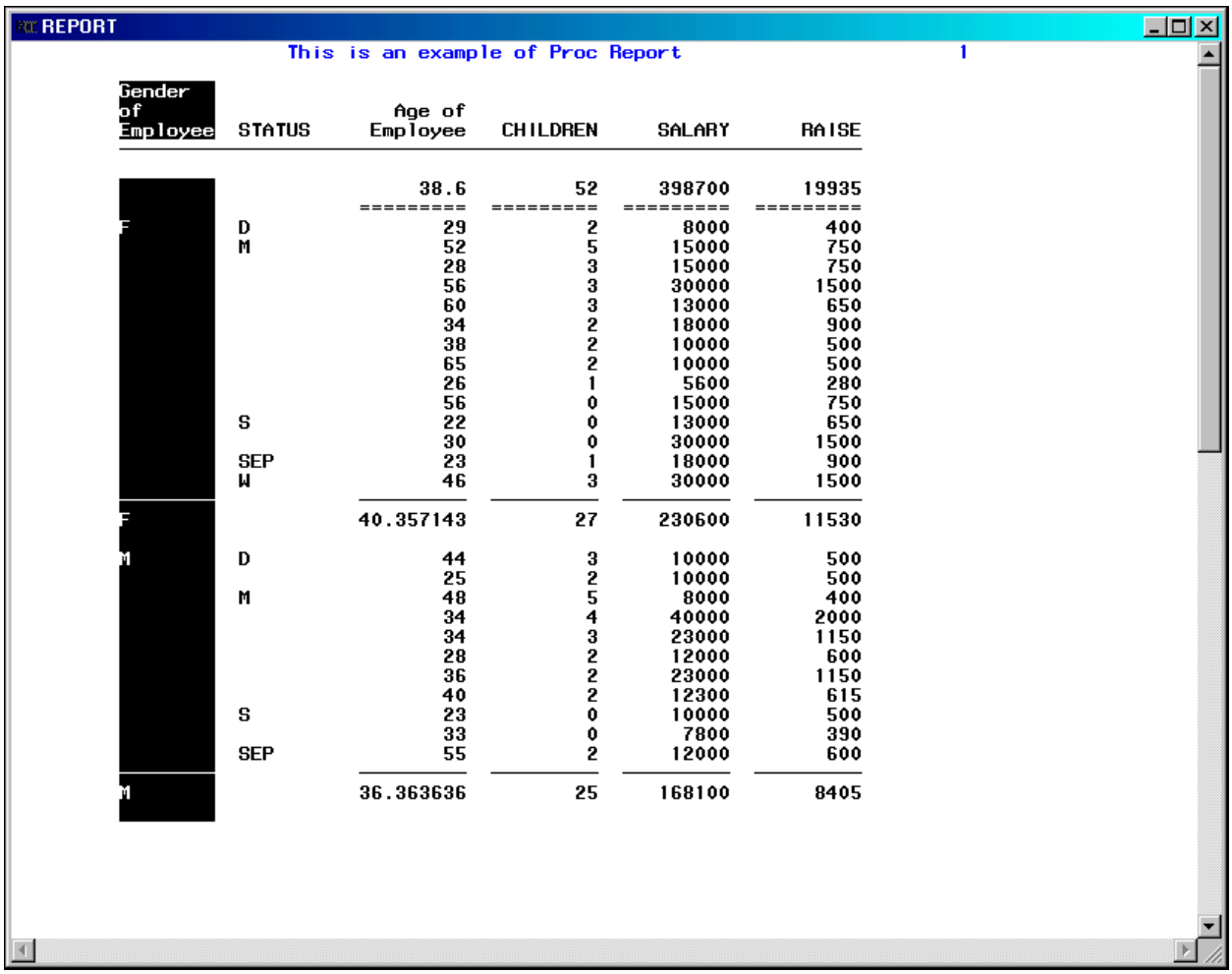
REPORT

This is an example of Proc Report 1

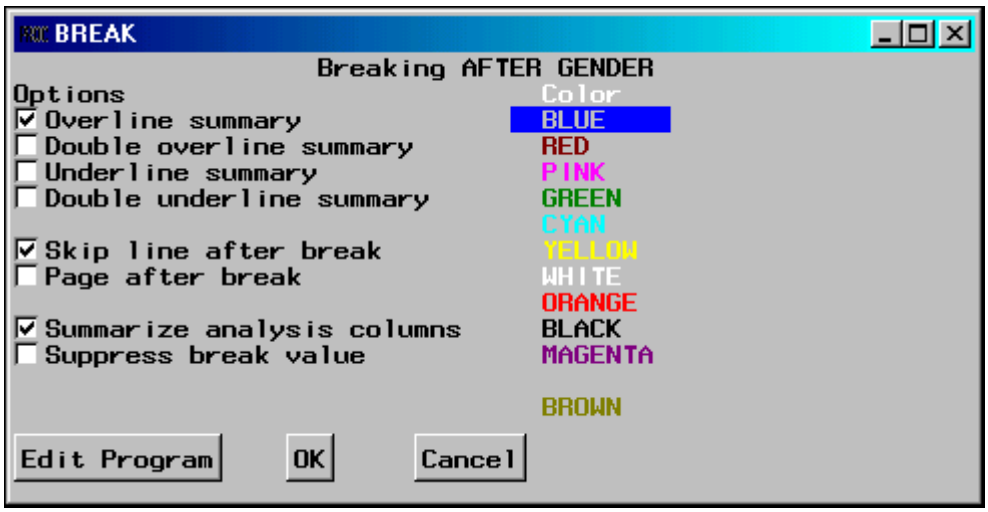
Gender of Employee	STATUS	Age of Employee	CHILDREN	SALARY	RAISE
		38.6	52	398700	19935
F	D	29	2	8000	400
	M	52	5	15000	750
		28	3	15000	750
		56	3	30000	1500
		60	3	13000	650
		34	2	18000	900
		38	2	10000	500
		65	2	10000	500
		26	1	5600	280
		56	0	15000	750
	S	22	0	13000	650
		30	0	30000	1500
	SEP	23	1	18000	900
	W	46	3	30000	1500
F		40.357143	27	230600	11530
M	D	44	3	10000	500
		25	2	10000	500
	M	48	5	8000	400
		34	4	40000	2000
		34	3	23000	1150
		28	2	12000	600
		36	2	23000	1150
		40	2	12300	615
	S	23	0	10000	500
		33	0	7800	390
	SEP	55	2	12000	600
M		36.363636	25	168100	8405

Finally, let's consider how grouping is used. First, we must restructure the report. Remove subtotaling an change all Order definitions to Group.

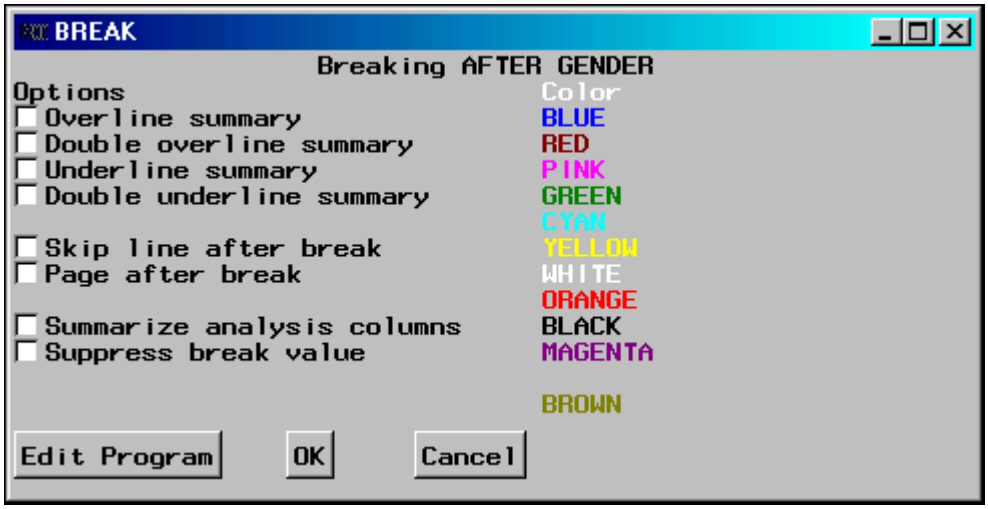
Highlight GENDER.



Select the Edit pull down and Summarize Information,

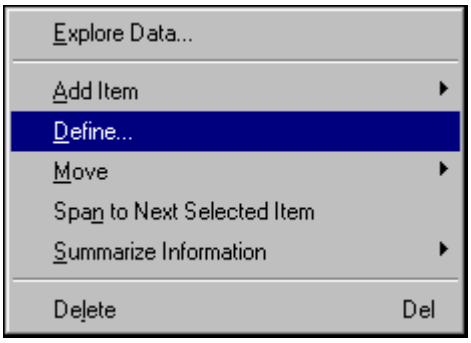


Unselect summary information.

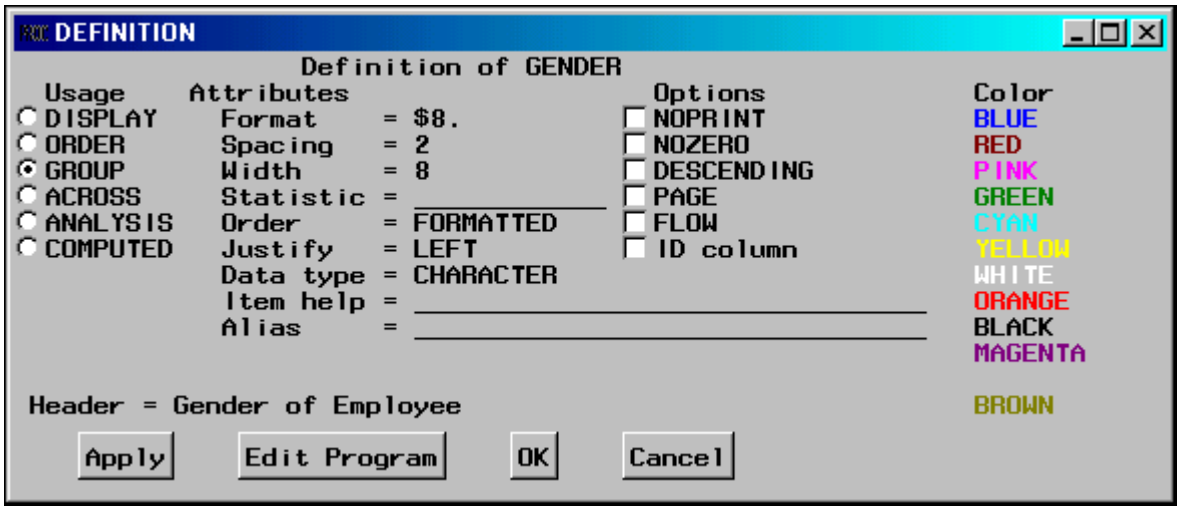


Click 

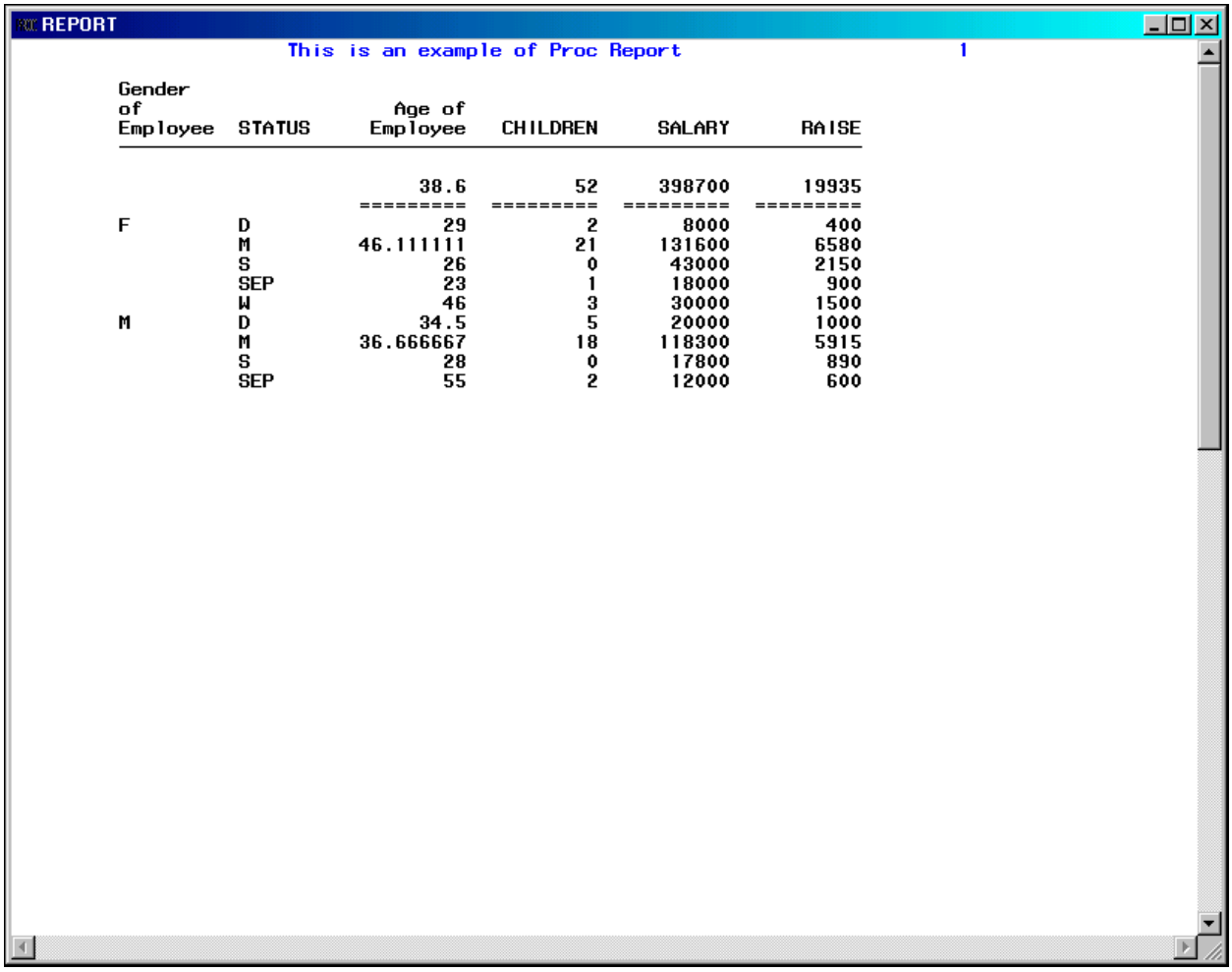
While GENDER is still highlighted, select the Edit pull down and



Change the Definition to Group.



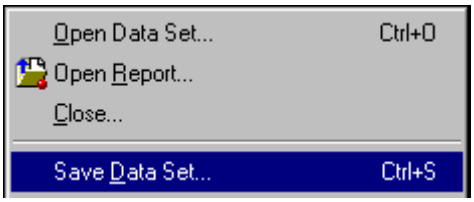
Click OK and see the result.

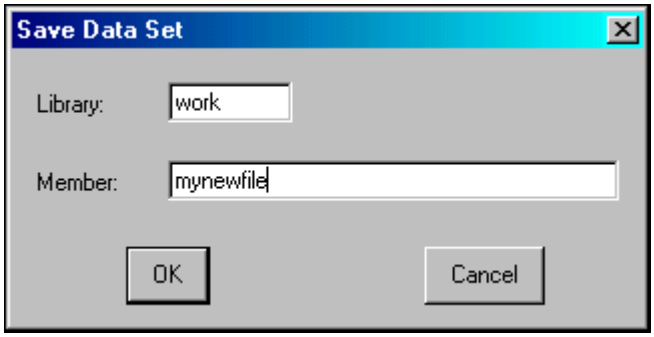


Notice how the report can also do the summarization for us.

Two utilities can be useful here. We can take this report structure and output it to a SAS data set or we can simply save the code for future use.

To output the result to a data set, select the File pull down and select

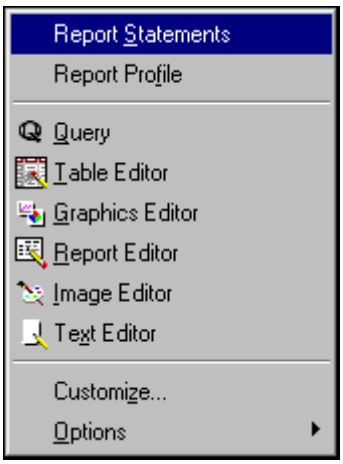


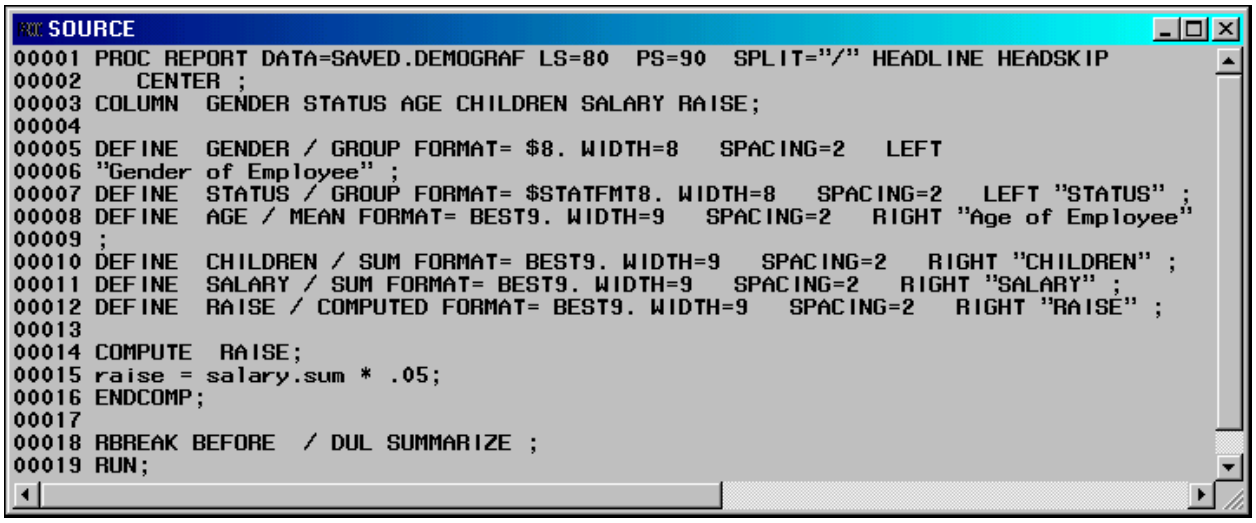


VIEWTABLE: work.mynewfile

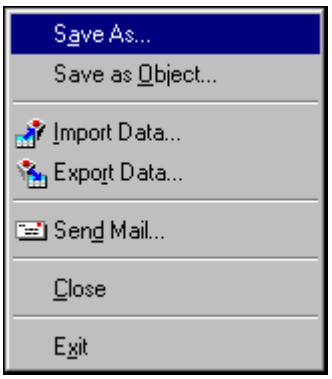
	Gender of Employee	STATUS	Age of Employee	CHILDREN	SALARY	RAISE	_BREAK_
1			38.6	52	398700	19935	_RBREAK_
2	F	D	29	2	8000	400	
3	F	M	46.111111111	21	131600	6580	
4	F	S	26	0	43000	2150	
5	F	SEP	23	1	18000	900	
6	F	W	46	3	30000	1500	
7	M	D	34.5	5	20000	1000	
8	M	M	36.666666667	18	118300	5915	
9	M	S	28	0	17800	890	
10	M	SEP	55	2	12000	600	

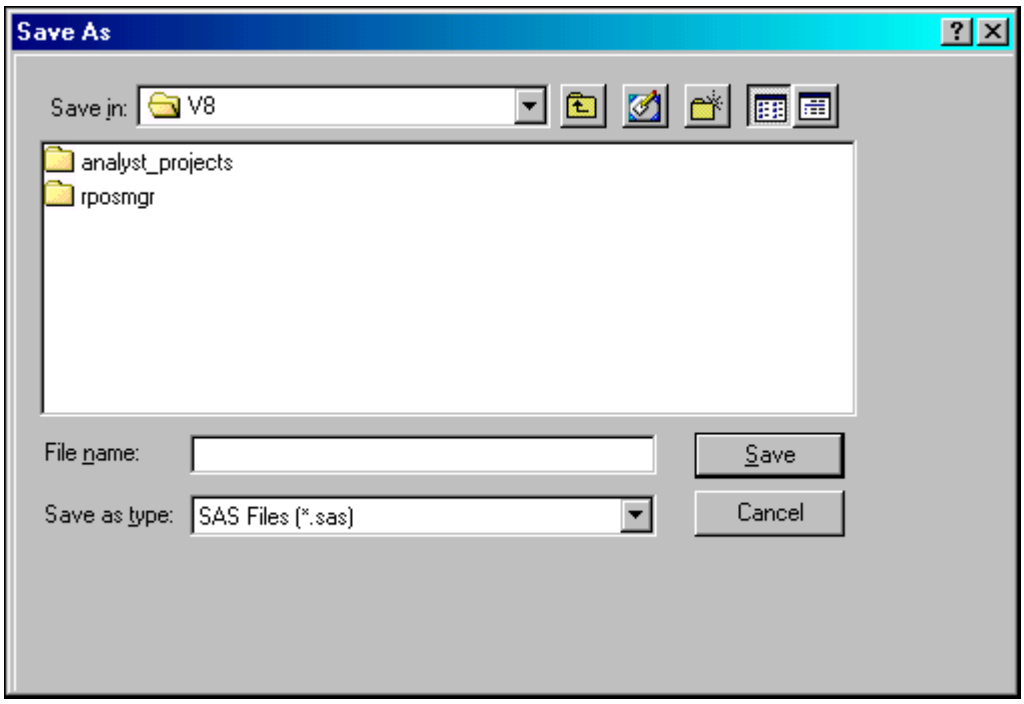
To save the code created to a file, simple select on the Tools pull down and





Notice no Title statements are part of the code. Select the File pull down





and give it a name.

Graphing in SAS Software

Prepared by

International SAS® Training and Consulting

Destiny Corporation – 100 Great Meadow Rd Suite 601 - Wethersfield, CT 06109-2379
Phone: (860) 721-1684 - 1-800-7TRAINING
Fax: (860) 721-9784
Email: info@destinycorp.com
Web: www.destinycorp.com

Proc Gplot

Proc GPLOT is the standard Sas procedure that allows us to create graphs in plot form.

The GPLOT procedure allows you to plot one variable against another, each pair derived from the same observation in the input data set.

We can produce relatively simple plots using a few statements and then enhance the result.

With GPLOT we can do the following:

- Draw reference lines on the plot
- Overlay plots
- Use any symbol to represent the points
- Reverse the order on the vertical scale
- Plot character variables (<= length 16)
- Select colors, symbols, interpolation methods, line styles
- Produce 'bubble' charts
- Plot a second vertical axis
- Produce logarithmic plots

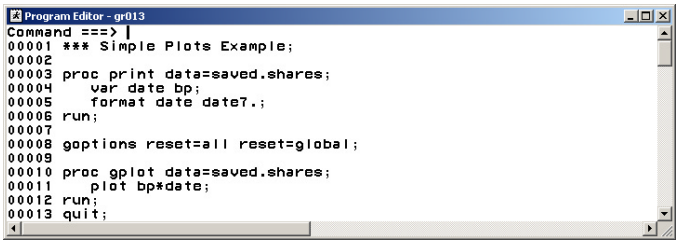
Simple Plots

Proc GPLOT uses the Share Price Data Set and produces a scatter plot for two variables.

The Plot statement first specifies the Y-axis.

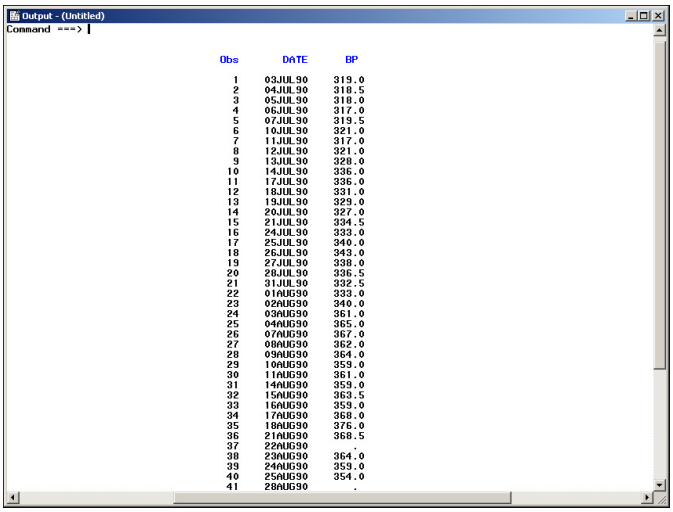
A Gplot step may contain any number of Plot statements.

A Quit statement is required at the end of the program code because the procedure resides in memory.

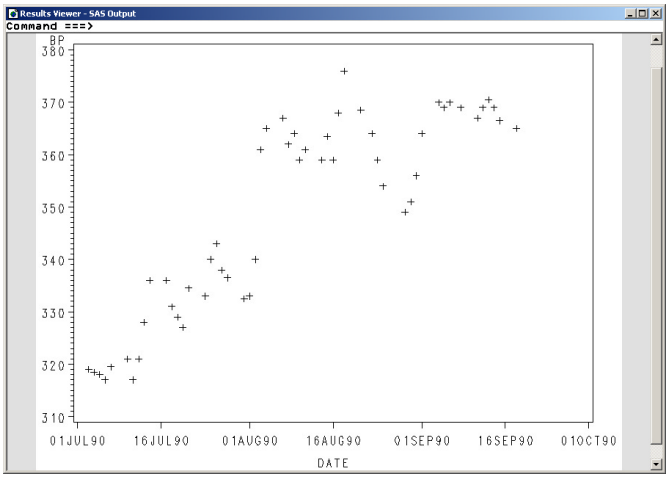


Note: Confirm that the Create Listing option in the Results tab in the Preferences window is checked. This is critical for SAS to be able to store the output created (in list format) and create a graph from this output.

SAS creates the following list output and graph output:



Obs	DATE	BP
1	03JUL90	319.0
2	04JUL90	318.5
3	05JUL90	318.0
4	06JUL90	317.0
5	07JUL90	319.5
6	10JUL90	321.0
7	11JUL90	317.0
8	12JUL90	321.0
9	13JUL90	328.0
10	14JUL90	336.0
11	17JUL90	336.0
12	18JUL90	331.0
13	19JUL90	329.0
14	20JUL90	327.0
15	21JUL90	334.5
16	24JUL90	333.0
17	25JUL90	340.0
18	26JUL90	343.0
19	27JUL90	338.0
20	28JUL90	336.5
21	31JUL90	332.5
22	01AUG90	333.0
23	02AUG90	340.0
24	03AUG90	361.0
25	04AUG90	365.0
26	07AUG90	367.0
27	08AUG90	362.0
28	09AUG90	364.0
29	10AUG90	359.0
30	11AUG90	361.0
31	14AUG90	359.0
32	15AUG90	363.5
33	16AUG90	359.0
34	17AUG90	368.0
35	18AUG90	376.0
36	21AUG90	368.5
37	22AUG90	.
38	23AUG90	364.0
39	24AUG90	359.0
40	25AUG90	354.0
41	28AUG90	.



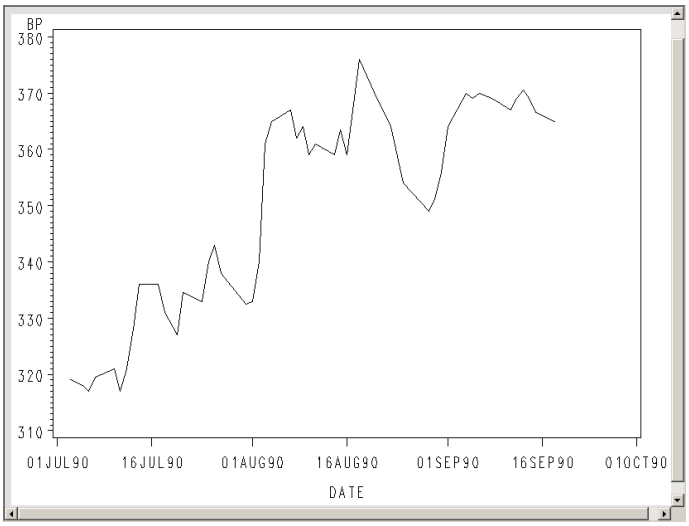
Connecting the Dots

By default, the values on the graph are left unconnected. This is probably the most accurate depiction of the data, because once lines are drawn between data points, we begin making assumptions about the data.

The Symbol statement is used to specify the line drawn between the data points. Join and Spline are two methods used by the symbol statement to connect the dots.

An example of the Symbol statement with the Join option is displayed below:

```
Program Editor - gr014
Command ===>
00001 *** Joining The Dots Example;
00002
00003 goptions reset=all reset=global;
00004
00005 symbol1 i=join;
00006 proc gplot data=saved.shares;
00007    plot bp*date;
00008 run;
00009 quit;
```



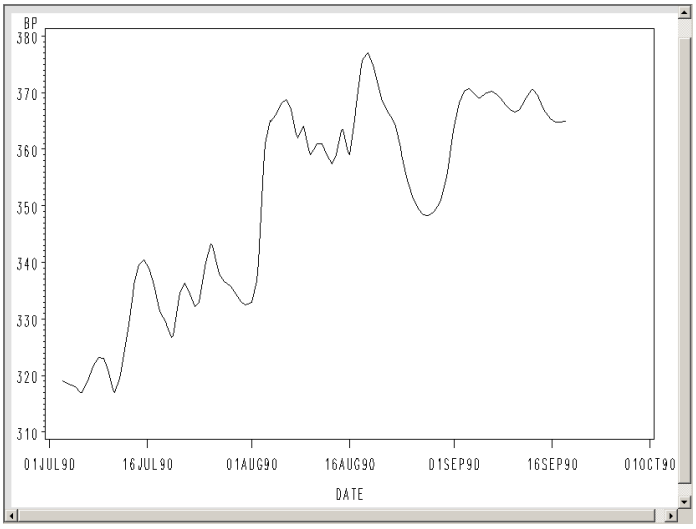
Once specified, the symbol statement remains in effect until it is canceled by another specification or by specifying the following:

Goptions Reset=Global;

This is only one way of joining the points on a graph.

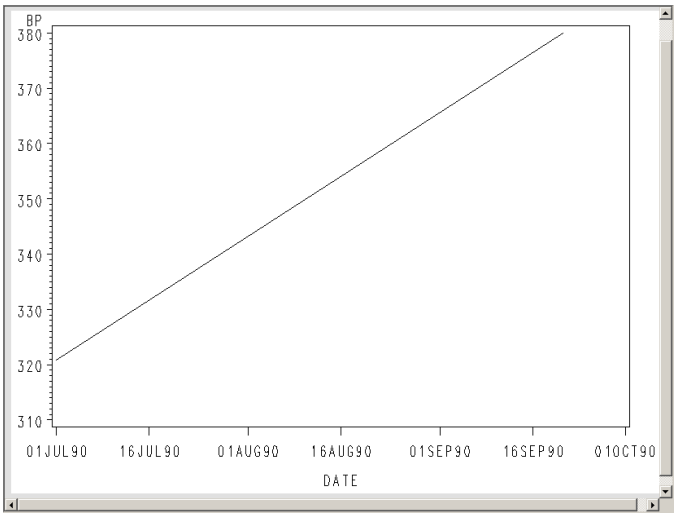
The symbol statement controls the type of line drawn, its color, width, line style and the symbol used to mark the data points. Now let's resubmit the code using the Spline option, which applies a smoothing effect to the points being connected.

```
Program Editor - gr014a
Command ===>
00001 *** Joining The Dots Example;
00002
00003 goptions reset=all reset=global;
00004
00005 symbol1 i=spline;
00006 proc gplot data=saved.shares;
00007    plot bp*date;
00008 run;
00009 quit;
```



Incorporating the rcli95 option in the Symbol Statement:

```
Program Editor - gr014b
Command ===>
00001 *** Joining The Dots Example;
00002
00003 goptions reset=all reset=global;
00004
00005 symbol1 i=rcli95;
00006 proc gplot data=saved.shares;
00007    plot bp*date;
00008 run;
00009 quit;
```

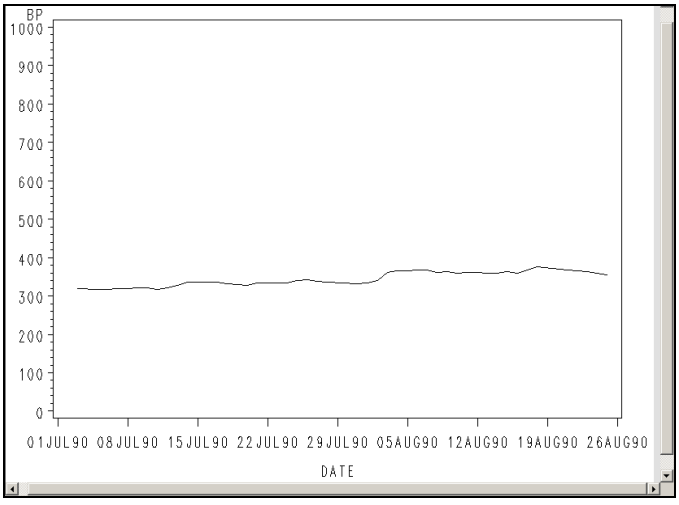


Labeling the Axes

In the previous example, the range of vertical axis of the graph (representing bp) is mapped by default. It ranges from 310 to 380 in scale.

We can change the default settings for both axes by specifying the ranges for the axes, as displayed in the code below:

```
Program Editor - gr015
Command ===>
00001 *** Labeling The Axes Example;
00002
00003 goptions reset=all reset=global;
00004
00005 symbol1 i=join;
00006 proc gplot data=saved.shares;
00007    plot bp*date /
00008    vaxis=0 to 1000 by 100
00009    haxis='01jul90'd to '01sep90'd by 7;
00010 run;
00011 quit;
```

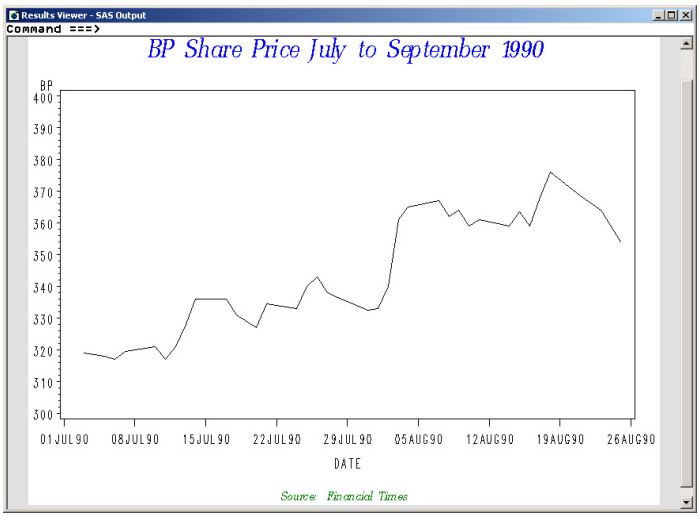


We can change the presentation of the graph, in this manner.

The symbol statement has been repeated here for clarity. It is not necessary, since it was submitted previously.

Adding Titles, Footnotes and Legends

```
Program Editor - gr016
Command ===>
00001 *** Adding Titles, Footnotes Example;
00002
00003 goptions reset=all reset=global;
00004
00005 symbol1 i=join;
00006 title1 h=2 c=blue f=zapfi
00007    'BP Share Price July to September 1990';
00008 footnote1 h=1 c=green f=zapfi 'Source:  Financial Times';
00009
00010 proc gplot data=saved.shares;
00011    plot bp*date /
00012    vaxis=300 to 400 by 10
00013    haxis='01jul90'd to '01sep90'd by 7;
00014 run;
00015 quit;
```



There is no legend on the graph even though the legend was requested.

We will see how to display legends using a different form of the plot statement, in a subsequent section.

Legends can be constructed with footnote statements on simple plots.

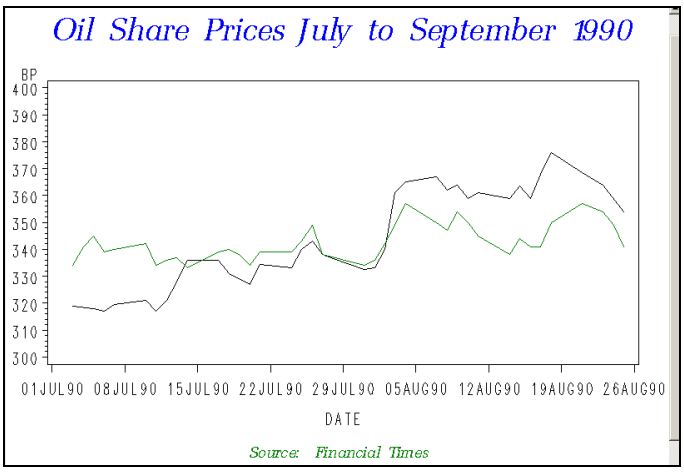
Once specified, the Title, Footnote and Symbol statements remain in effect until canceled.

Overlaying

The overlay option is used to place multiple graphs on a page.

This is illustrated in the code below:

```
Program Editor - gr017
Command ===>
00001 *** Overlaying Example;
00002
00003 goptions reset=all reset=global;
00004
00005 symbol1 i=join;
00006 title1 h=2 c=blue f=zapfi
00007    'Oil Share Prices July to September 1990';
00008 footnote1 h=1 c=green f=zapfi 'Source:  Financial Times';
00009 footnote3 h=1 c=blue  f=zapfi 'NYSE';
00010 proc gplot data=saved.shares;
00011    plot bp*date burmah*date ultramar*date /
00012    overlay
00013    vaxis=300 to 400 by 10
00014    haxis='01jul90'd to '01sep90'd by 7;
00015 run;
00016 quit;
```



Notice the colors of the plot lines.

One symbol statement has been used in this graph, all with the option i=join.

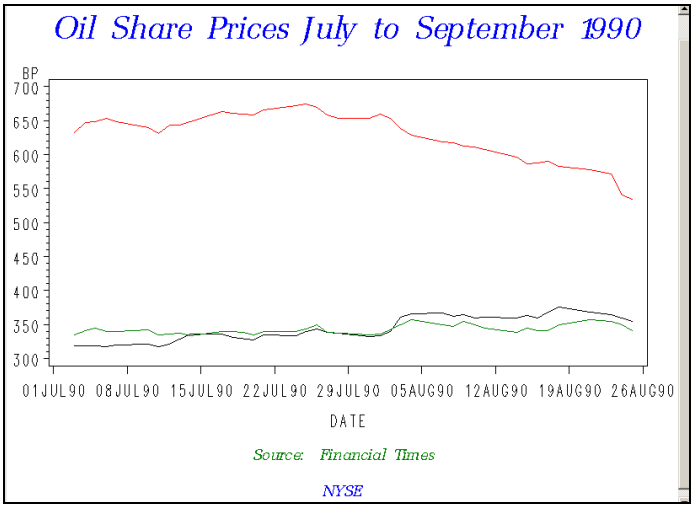
The second plot requested was not produced. The SAS log informs us that the values to be plotted lie outside the axis range specified: either the vaxis or the haxis specification is not sufficient to cover all the data.

Since the values of date are the same for all share prices, we can assume that the vaxis needs to be extended.

Let's re-submit the code after making modifications to the vaxis, as displayed:

```
Program Editor - gr017a
Command ===>
00001 *** Overlaying Example;
00002
00003 goptions reset=all reset=global;
00004
00005 symbol1 i=join;
00006 title1 h=2 c=blue f=zapfi
00007    'Oil Share Prices July to September 1990';
00008 footnote1 h=1 c=green f=zapfi 'Source:  Financial Times';
00009 footnote3 h=1 c=blue  f=zapfi 'NYSE';
00010 proc gplot data=saved.shares;
00011    plot bp*date burmah*date ultramar*date /
00012    overlay
00013    vaxis=300 to 700 by 50
00014    haxis='01jul90'd to '01sep90'd by 7;
00015 run;
00016 quit;
```

The graph now displays all the plots as requested in the code.



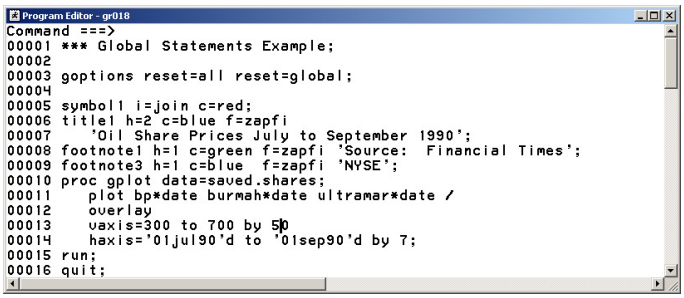
Global Statements

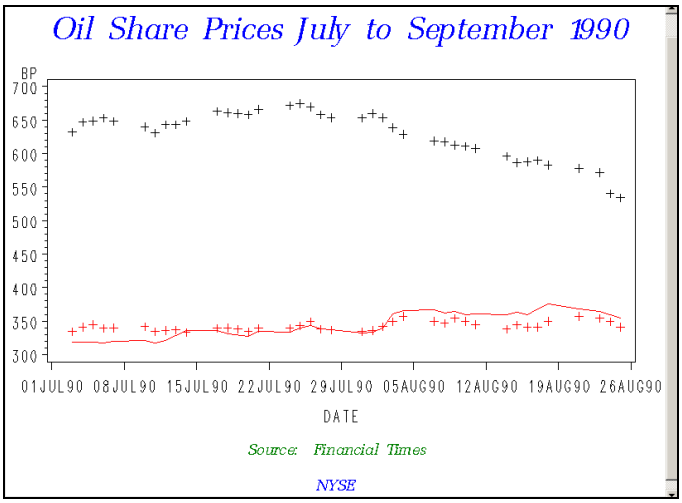
Symbol statements cycle round the list of colors if not color is specified

Symbol statements are Global.

The program below first resets the global statements to the default and then sets the Symbol, Title and Footnotes again.

This is clearly inefficient, and has been shown solely for illustration purposes.





Symbol statements are additive, in addition to being global in nature.

The result of Symbol1 displayed in the previous code is:

	Symbol1 i=join;
+	Symbol1 c=red;
=	Symbol1 i=join c=red;

The graph shows one symbol statement being used, a joined red line.

The other plots do not have symbol statements and a join is not displayed. They have begun to cycle round the list of colors for the device.

Cycling Colors

If the Symbol statement is not given a color, then it cycles round the list of colors for the device, generating a symbol statement each time.

Given a device with the following colors:

goptions colors=(black,red,green,blue,orange,brown);

Consider the effect of the following Symbol statements:

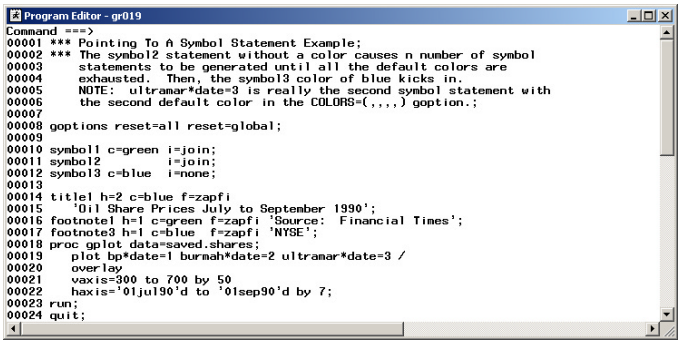
Symbol1 c=green i=join;
Symbol2 i=join;
Symbol3 c=blue i=join;

Results in the following symbol statements generated:

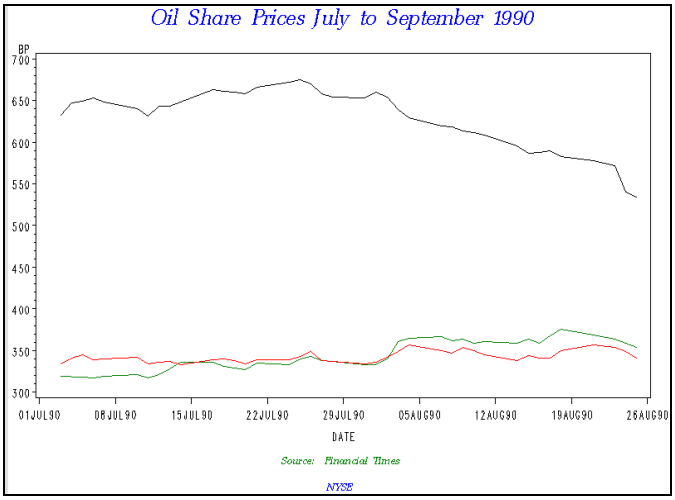
```
symbol c=green  i=join;* 1st statement generated;
symbol c=black  i=join;* 2nd generated;
symbol c=red    i=join;* 3rd generated, etc...;
symbol c=green  i=join;
symbol c=blue   i=join;
symbol c=orange i=join;
symbol c=brown  i=join;
symbol c=blue   i=join;
```

Pointing to a Symbol Statement

The Plot statement can point to a Symbol Statement. It points to the nth generated symbol statement, as displayed below:



The values inform SAS how to assign the Symbol Statements.



Line Control

Proc Gplot controls lines using the Symbol Statement. The symbol statement can accept many options Symbols.

SYMBOL STATEMENT LINE = OPTIONS

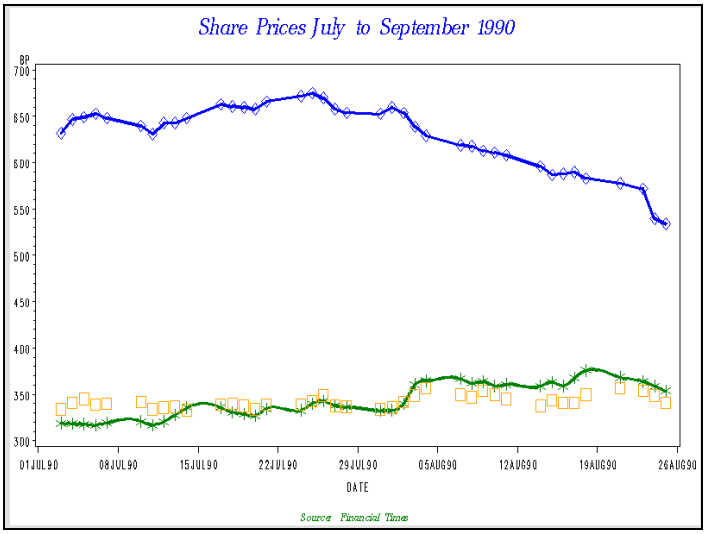
SYMBOL STATEMENT VALUE= OPTIONS

SYMBOLS CODE(SYMBOLS)

Variations

Examples of Symbol Statements:

```
Program Editor - gr020
Command ===>
00001 *** Variations of Symbol Statements Example;
00002
00003 goptions reset=all reset=global;
00004
00005 symbol1 i=spline c=green  v=star    h=2 w=3;
00006 symbol2 i=join   c=blue   v=diamond h=2 w=3;
00007 symbol3 i=none   c=orange v=square  h=2 w=2;
00008
00009 title1    h=2 c=blue  f=zapfi 'Share Prices July to September 1990';
00010 footnote1 h=1 c=green f=zapfi 'Source:  Financial Times';
00011 proc gplot data=saved.shares;
00012    plot bp*date burmah*date ultramar*date /
00013    overlay
00014    vaxis=300 to 700 by 50
00015    haxis='01jul90'd to '01sep90'd by 7;
00016 run;
00017 quit;
```

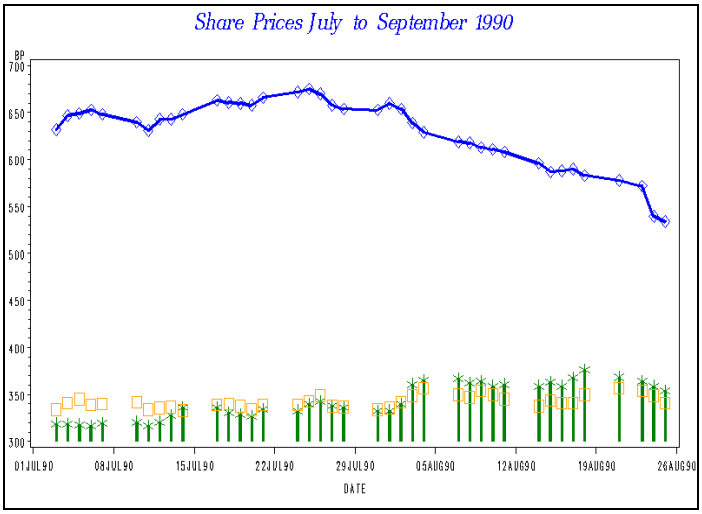


Now let's modify the code by changing the interpolation method for symbol1 to needle.

Resubmit the code:

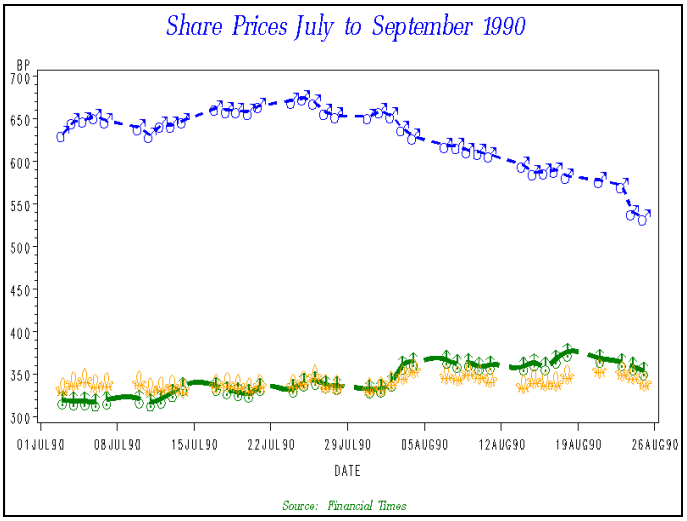
```
Program Editor - gr020a
Command ===>
00001 *** Variations of Symbol Statements Example;
00002
00003 goptions reset=all reset=global;
00004
00005 symbol1 i=needle c=green  v=star    h=2 w=3;
00006 symbol2 i=join   c=blue   v=diamond h=2 w=3;
00007 symbol3 i=none   c=orange v=square  h=2 w=2;
00008
00009 title1    h=2 c=blue  f=zapfi 'Share Prices July to September 1990';
00010 footnote1 h=1 c=green f=zapfi 'Source:  Financial Times';
00011 proc gplot data=saved.shares;
00012    plot bp*date burmah*date ultramar*date /
00013    overlay
00014    vaxis=280 to 400 by 20
00015    haxis='01jul90'd to '01sep90'd by 7;
00016 run;
00017 quit;
```

The output is displayed below.

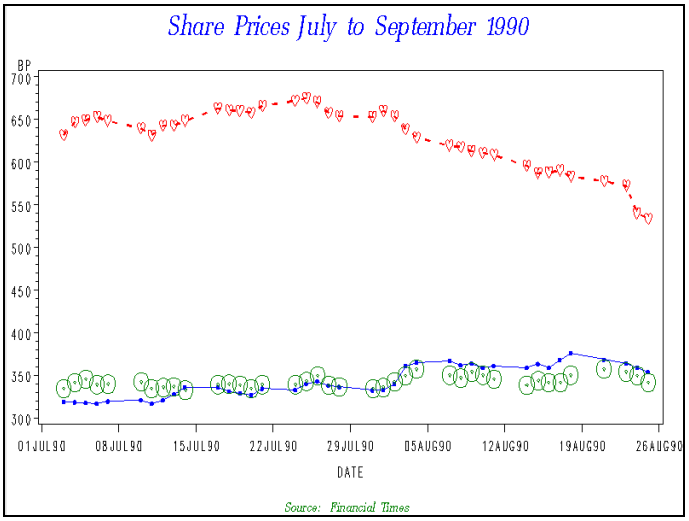


Symbol Statement Examples

```
Program Editor - gr023
Command ===>
00001 *** Variations of Symbol Statements Example 2;
00002
00003 goptions reset=all reset=global;
00004
00005 symbol1 i=spline c=green  v='.' h=2 w=6 l=5;
00006 symbol2 i=join   c=blue   v='>' h=2 w=3 l=2;
00007 symbol3 i=none   c=orange v="'" h=2 w=2 l=10;
00008
00009 title1    h=2 c=blue  f=zapfi 'Share Prices July to September 1990';
00010 footnote1 h=1 c=green f=zapfi 'Source:  Financial Times';
00011 proc gplot data=saved.shares;
00012    plot bp*date burmah*date ultramar*date /
00013    overlay
00014    vaxis=300 to 700 by 50
00015    haxis='01jul90'd to '01sep90'd by 7;
00016 run;
00017 quit;
```



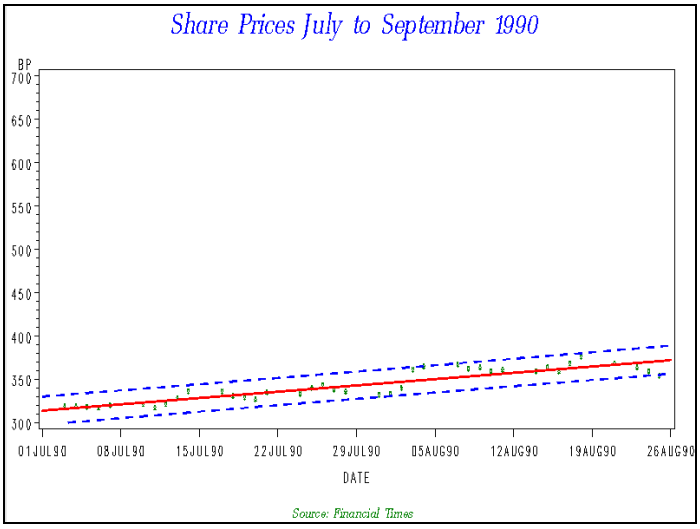

```
Program Editor - gr024
Command ===>
00001 *** Variations of Symbol Statements Example 3;
00002
00003 goptions reset=all reset=global;
00004
00005 symbol1 i=join l=1  c=blue  v=dot h=0.5 w=1;
00006 symbol2 i=L    l=34 c=red   v='*' h=1.2 w=2;
00007 symbol3 i=none l=45 c=green v="-" h=2.3 w=3;
00008
00009 title1    h=2 c=blue  f=zapfi 'Share Prices July to September 1990';
00010 footnote1 h=1 c=green f=zapfi 'Source:  Financial Times';
00011 proc gplot data=saved.shares;
00012    plot bp*date burmah*date ultramar*date /
00013    overlay
00014    vaxis=300 to 700 by 50
00015    haxis='01jul90'd to '01sep90'd by 7;
00016 run;
00017 quit;
```



Next, we use the option designed to present information graphing confidence limits.

Program Editor - gr026

```
Command ===>
00001 *** Symbol Statement Examples 5;
00002
00003 goptions reset=global reset=all;
00004 symbol i=rqcli95 ci=red co=blue cv=green v='@' w=2 h=0.5;
00005
00006 title1 h=2 c=blue f=zapfi 'Share Prices July to September 1990';
00007 footnote1 h=1 c=green f=zapfi 'Source: Financial Times';
00008 proc gplot data=saved.shares;
00009     plot bp*date /
00010     vaxis=300 to 700 by 50
00011     haxis='01jul90'd to '01sep90'd by 7;
00012 run;
00013 quit;
```



X*Y=Z and Axis Control

A normal plot statement of the form:

Plot A*C B*C / Overlay;

The above code displays two plots on one graph, but no legend showing which line corresponded to A, and which to B.

X*Y=Z

Using the X*Y=Z form of the Plot statement generates automatic Legends which display what each line represents.

Plot X*Y=Z;

To do this, the data must be in a certain structure.

Review the Flu data set:

Program Editor - gr027

```
Command ===>
00001 *** Axis Control FLU Data Set Printout;
00002
00003 title;
00004 footnote;
00005 proc print data=saved.flu;
00006 run;
```

Output in list form:

Output - (Untitled)

```
Command ===>
                                                    14:57 Wednesday, April 4, 2001

Obs    A    B    C    D    E    WEEK_NO    DATE

  1   10    8   12   15   17          1   10838
  2   21   15   12   11   26          2   10845
  3   25   16   13   18   22          3   10852
  4   32   16   13   11   41          4   10859
  5   36   24   23   45   56          5   10866
  6   45   32   35   43   65          6   10873
  7   56   34   38   56   75          7   10880
  8   86   45   48   68   97          8   10887
  9  120   75   88   94  110          9   10894
 10  140   89   97  106  120         10   10901
 11  168  102  110  132  154         11   10908
 12  190  110  128  154  178         12   10915
 13  224  100  121  131  191         13   10922
 14  180   94  112  125  160         14   10929
 15  128   82   96  102  112         15   10936
 16  114   67   78   88   95         16   10943
 17   85   45   56   59   76         17   10957
 18   63   34   44   49   54         18   10964
 19   35   23   36   38   17         19   10971
 20   23   11   18   19   19         20   10978
 21   12    4    8    9   10         21   10985
```

Output in HTML form:

Obs	A	B	C	D	E	WEEK_NO	DATE
1	10	8	12	15	17	1	10838
2	21	15	12	11	26	2	10845
3	25	16	13	18	22	3	10852
4	32	16	13	11	41	4	10859
5	36	24	23	45	56	5	10866
6	45	32	35	43	65	6	10873
7	56	34	38	56	75	7	10880
8	86	45	48	68	97	8	10887
9	120	75	88	94	110	9	10894
10	140	89	97	106	120	10	10901
11	168	102	110	132	154	11	10908
12	190	110	128	154	178	12	10915
13	224	100	121	131	191	13	10922
14	180	94	112	125	160	14	10929
15	128	82	96	102	112	15	10936
16	114	67	78	88	95	16	10943
17	85	45	56	59	76	17	10957
18	63	34	44	49	54	18	10964
19	35	23	36	38	17	19	10971
20	23	11	18	19	19	20	10978
21	12	4	8	9	10	21	10985

We could Plot A*Date B*Date C*Date D*Date E*Date /Overlay; but this would not generate a legend.

We need to rearrange the data for the Y*X=Z form.

Rearranging Data

Program Editor - gr028

```
Command ===>
00001 *** Rearranging The Data;
00002
00003 data epidemic;
00004     set saved.flu;
00005     keep cases category week_no date;
00006     category = 'A'; cases=a; output;
00007     category = 'B'; cases=b; output;
00008     category = 'C'; cases=c; output;
00009     category = 'D'; cases=d; output;
00010     category = 'E'; cases=e; output;
00011 run;
00012 title;
00013 footnote;
00014 proc print data=epidemic;
00015     format date date7.;
00016 run;
```

Obs	WEEK_NO	DATE	category	cases
1	1	03SEP89	A	10
2	1	03SEP89	B	8
3	1	03SEP89	C	12
4	1	03SEP89	D	15
5	1	03SEP89	E	17
6	2	10SEP89	A	21
7	2	10SEP89	B	15
8	2	10SEP89	C	12
9	2	10SEP89	D	11
10	2	10SEP89	E	26
11	3	17SEP89	A	25
12	3	17SEP89	B	16
13	3	17SEP89	C	13
14	3	17SEP89	D	18
15	3	17SEP89	E	22

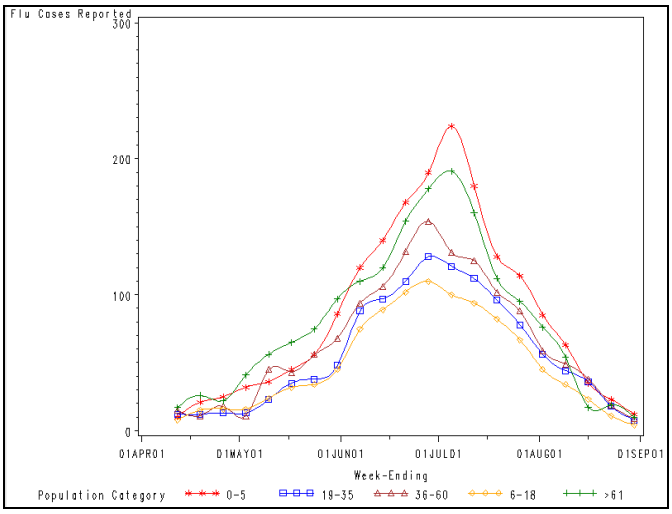
We can now Plot Cases*Date=Category;

For each Category, A, B, C, D, E a separate line will be drawn on the graph.

Program Editor - gr029

```
Command ===>
00001 *** Automated Legend Example;
00002
00003 goptions reset=global reset=all;
00004 proc format;
00005    value $age 'A'='0-5'
00006               'B'='6-18'
00007               'C'='19-35'
00008               'D'='36-60'
00009               'E'='>61';
00010 run;
00011 symbol1 v=star     c=red    i=spline;
00012 symbol2 v=square   c=blue   i=spline;
00013 symbol3 v=triangle c=brown  i=spline;
00014 symbol4 v=diamond  c=orange i=spline;
00015 symbol5 v=plus     c=green  i=spline;
00016 proc gplot data=saved.epidemic;
00017    plot cases*date=category;
00018    label date='Week-Ending'
00019          category='Population Category'
00020          cases='Flu Cases Reported';
00021    format date date7. category $age.;
00022 run;
00023 quit;
```

The output is displayed below.



The five Symbols statements generated will match with the five lines drawn, one for each value of category.

Always review the SAS log for notes and messages. They inform you about plots that lie outside the available range.

Introduction to Charts

SAS/GRAPH can create different chart styles.

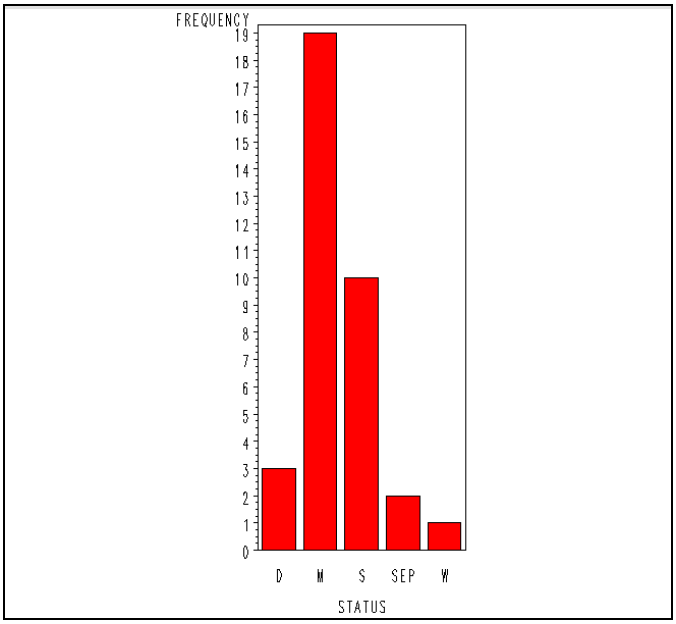
Let's illustrate with a few examples.

Vertical Bar Chart Example

Submit the following code:

Program Editor - ch4_1

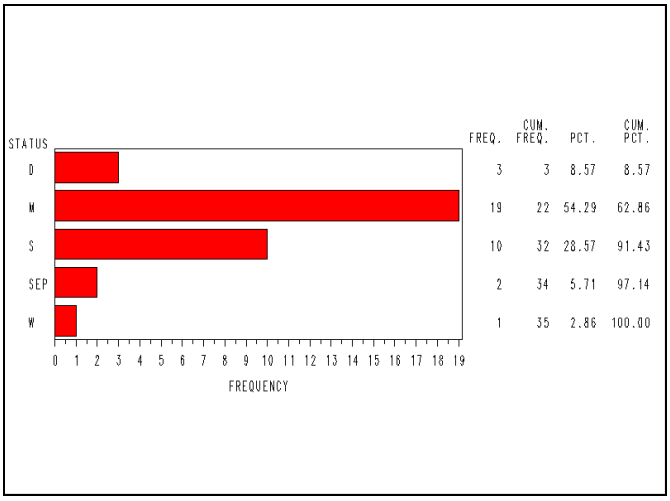
```
Command ===>
00001 goptions reset=global reset=all;
00002
00003 proc gchart data=saved.demograf;
00004    vbar status;
00005 run;
00006 quit;
```



Horizontal Bar Chart Example

Program Editor - ch4_2

```
Command ===>
00001 goptions reset=global reset=all;
00002
00003 proc gchart data=saved.demograf;
00004    hbar status;
00005 run;
00006 quit;
```



Notes:

Unless otherwise specified, SAS displays the frequency of the selected variable for display, by default.

Each bar or block represents a value of a variable, either character or numeric.

Bars can be grouped, sub-grouped and various patterns and colors can be used to enhance the presentation.

By default, Horizontal Bar Charts display graph statistics on the graph area next to the graph.

Pie charts can be of two types, Pie Charts and Star Charts.

Terminology

Physical Forms

The physical form of a chart is determined by the type specified in SAS code:

```
Proc Gchart Data=SAS_data_set;
  Vbar Variable / Options ;
```

The Vbar statement requests a vertical bar chart.

The variable after Vbar determines the number of bars the graph will contain.

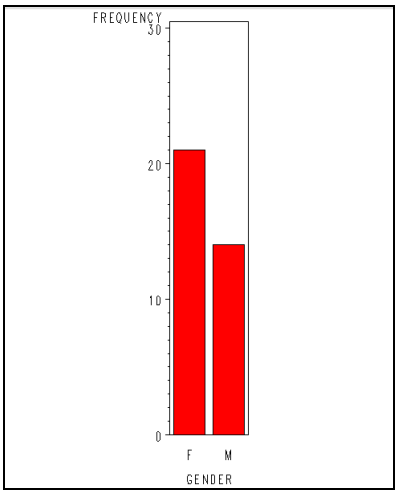
Options on the statement control other aspects of the graph.

For example, the Demograf data set has a variable named gender, which has two values: F and M.

If we specify the following:

```
Program Editor - gr038
Command ===>
00001 *** Vbar Example;
00002
00003 goptions reset=global reset=all;
00004
00005 proc gchart data=saved.demograf;
00006    vbar gender;
00007 run;
00008 quit;
```

The following chart is displayed.



By default, the vertical axis displays the frequency or the number of observations in the data set.

In this case, there are 21 females and 14 males.

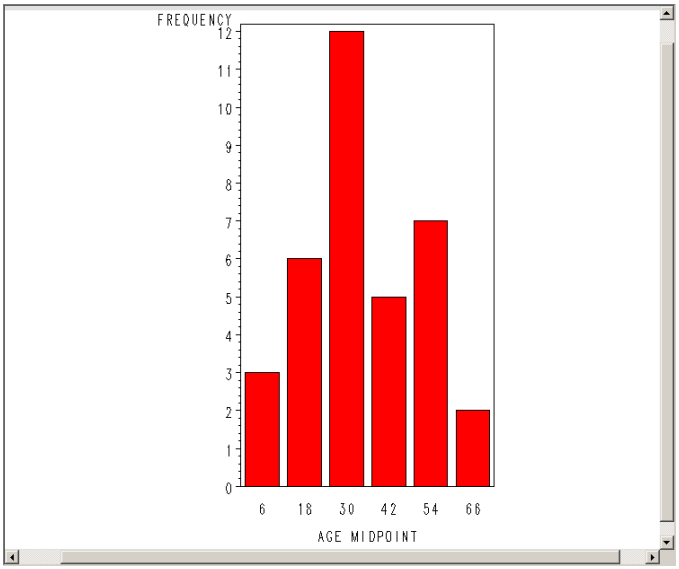
Midpoints

For numeric variables, SAS/GRAPH will chart the midpoints of a data range.

Sometimes this produces unexpected results.

Let's consider the same vertical bar chart as before, but use a numeric variable such as Age that takes on values from 15 to 65.

```
Program Editor - gr039
Command ===>
00001 *** Midpoints Example;
00002
00003 goptions reset=global reset=all;
00004
00005 proc gchart data=saved.demograf;
00006    vbar age;
00007 run;
00008 quit;
```



As seen from the graph, it does not produce one bar for each value in the data.

The data range (2 to 65) is divided into ranges, and the midpoint of each range is charted:

Range	Midpoint
2-12	6
12-23	18
24-35	30
36-47	42
48-59	54
60-71	66

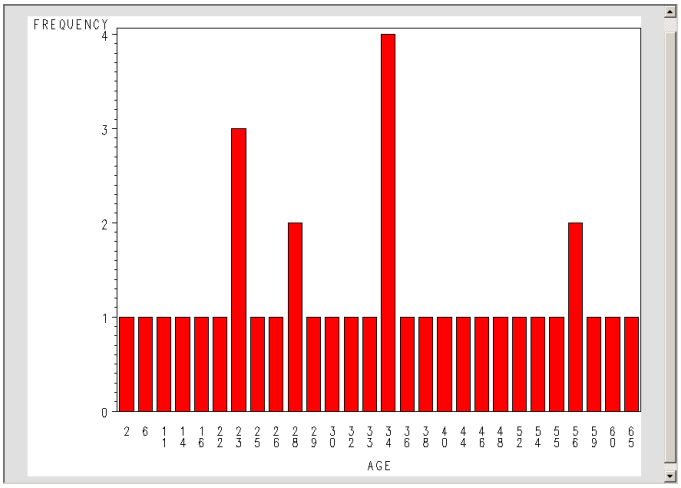
DISCRETE

The discrete option suppresses the calculation of these ranges and forces the procedure to produce one bar for each value in the data.

Submit the code below for illustration purposes.

```
Program Editor - gr040
Command ===>
00001 *** Discrete Example;
00002
00003 goptions reset=global reset=all;
00004
00005 proc gchart data=saved.demograf;
00006    vbar age / discrete;
00007 run;
00008 quit;
```

The graph will be displayed, as follows:



The Discrete option is valid for all types of charts.

Valid Options for GCHART

Some options are common to the Vbar, Hbar, Block, Pie and Star Statements, while others are specific to the type of chart being drawn.

The table below lists options the types of chart(s) to which they can be applied.

Option	Vbar	Hbar	Block	Pie	Star
ANNOTATE	*	*	*	*	*
CAXIS	*	*	*		
COUTLINE	*	*	*	*	*
CTEXT	*	*	*	*	*
BLOCKMAX			*		
PATTERNID=BY	*	*	*		
GROUP	*	*	*		
MIDPOINT	*	*	*		
SUBGROUP	*	*	*		
LEGEND	*	*	*		
NOLEGEND	*	*	*		
NOHEADING			*	*	*
FREQ	*	*	*	*	*
G100	*	*	*		
SUMVAR	*	*	*	*	*
TYPE=CFREQ	*	*	*	*	*
CPERCENT	*	*	*	*	*
FREQ	*	*	*	*	*
MEAN	*	*	*	*	*
PERCENT	*	*	*	*	*
SUM	*	*	*	*	*
DISCRETE	*	*	*	*	*
LEVELS	*	*	*	*	*
MIDPOINTS	*	*	*	*	*
MISSING	*	*	*	*	*
GROUP	*	*	*	*	*
SUBGROUP	*	*	*		
DESCRIPTION	*	*	*	*	*
NAME	*	*	*	*	*
GSPACE	*	*			
SPACE	*	*			
WIDTH	*	*			
CFRAME	*	*			

Option	Vbar	Hbar	Block	Pie	Star
FRAME	*	*			
CFREQ	*	*			
CPERCENT	*	*			
FREQ	*	*			
MEAN	*	*			
NOSTATS		*			
PERCENT	*	*			
SUM	*	*			
GAXIS	*	*			
MAXIS	*	*			
NOAXIS	*	*			
RAXIS	*	*			
MINOR	*	*			
ASCENDING	*	*			
DESCENDING	*	*			
NOZERO	*	*			
AUTOREF	*	*			
CLIPREF	*	*			
NOBASEREF	*	*			
REF	*	*			
CFILL				*	*
MATCHCOLOR				*	*
ANGLE				*	*
EXPLODE				*	
FILL				*	*
INVISIBLE				*	
PERCENT				*	*
SLICE				*	*
VALUE				*	*
NOGROUPHEADING				*	*
OTHER				*	
ACROSS				*	*
DOWN				*	*
NOCONNECT					*

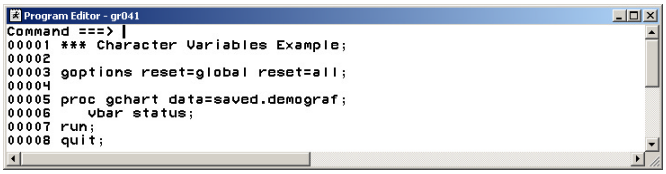
Bar Charts

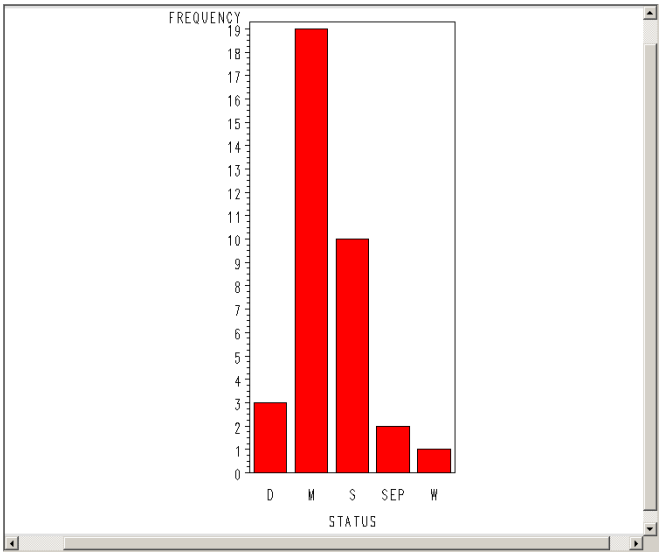
Vertical Bar Charts

The Vbar Statement produces Vertical Bar Charts, also known as Histograms.

Character Variables

The discrete option does not add value when used with a character variable since each value in the data is given a bar.



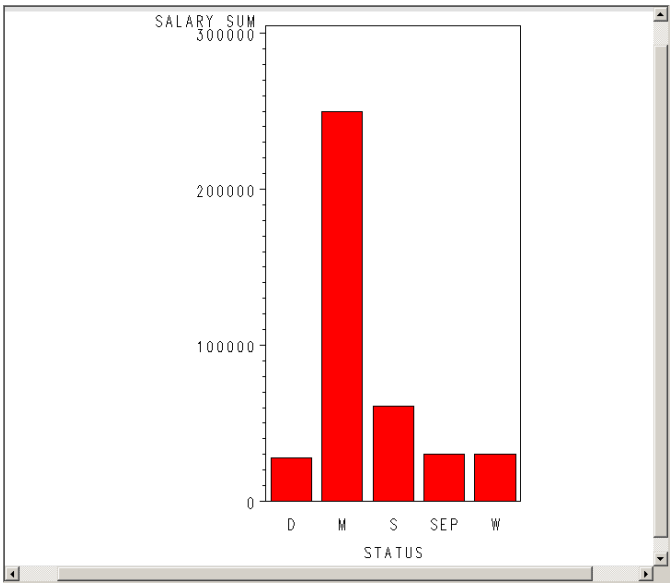


The Response Axis

The response axis is controlled with the SUMVAR (summing variable) option.

```
Program Editor - gr042
Command ===>
00001 *** Response Axis Example;
00002
00003 goptions reset=global reset=all;
00004
00005 proc gchart data=saved.demograf;
00006    vbar status / sumvar=salary;
00007 run;
00008 quit;
```

By default, the sumvar option displays the sum for the specified variable.



Statistics for SUMVAR

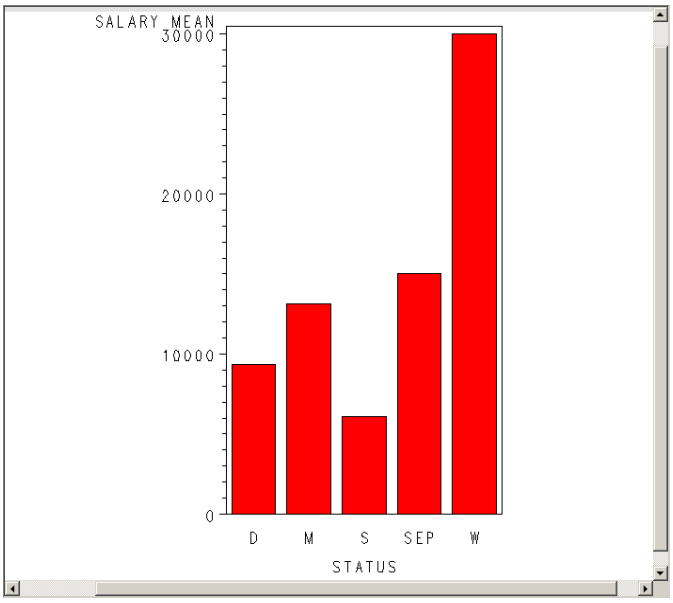
By default, the statistic displayed for Salary is SUM.

The response axis can display several different statistics, such as Mean, Sum (total), Freq, Cumulative Percent, Cumulative Frequency.

The option controlling statistics is TYPE.

Type and Sumvar work together to control which variable and statistic is displayed on the response axis.

```
Program Editor - gr043
Command ===>
00001 *** Statistics For Sumvar Example;
00002
00003 goptions reset=global reset=all;
00004
00005 proc gchart data=saved.demograf;
00006    vbar status / sumvar=salary type=mean;
00007 run;
00008 quit;
```

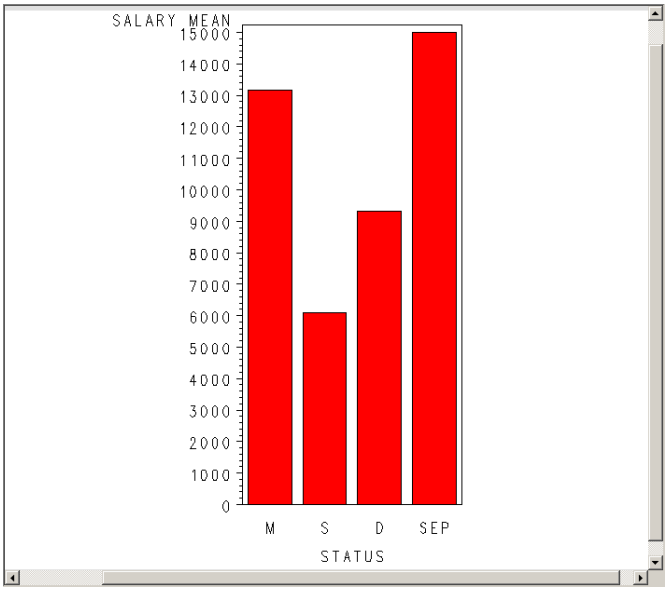


Controlling the Bars

The Midpoints option allows specification of which bars to chart. Use quotation marks around character values.

The usual SAS shortcuts can be coded when using Midpoints, e.g. 10 to 100 by 10.

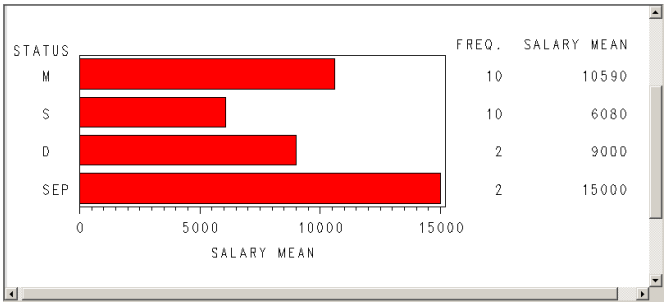
```
Program Editor - gr044
Command ===>
00001 *** Controlling The Bars Example;
00002
00003 goptions reset=global reset=all;
00004
00005 proc gchart data=saved.demograf;
00006    vbar status / sumvar=salary type=mean
00007 midpoints='M' 'S' 'D' 'SEP';
00008 run;
00009 quit;
```



Selecting Observations

The Where clause can be used to subset certain observations you wish to chart.

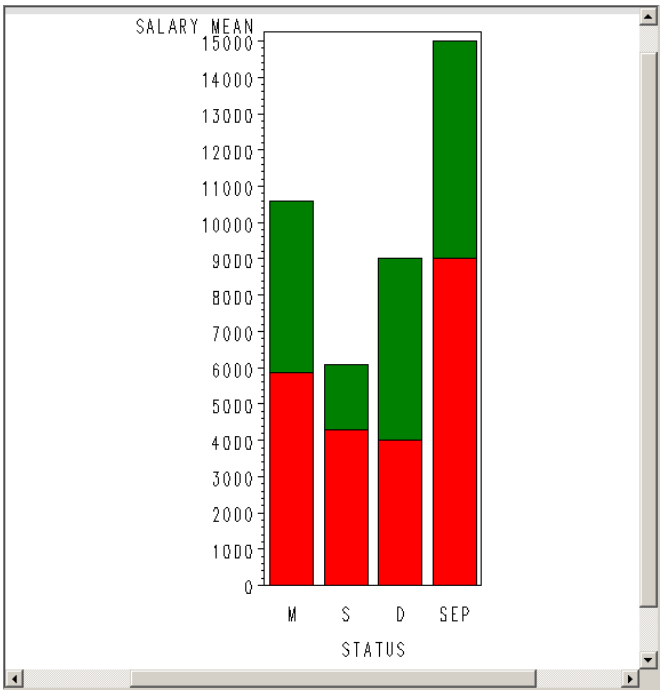
```
Program Editor - gr045
Command ===>
00001 *** Selecting Observations Example;
00002
00003 goptions reset=global reset=all;
00004
00005 proc gchart data=saved.demograf;
00006    hbar status / sumvar=salary type=mean
00007    midpoints='M' 'S' 'D' 'SEP';
00008    where children le 2;
00009 run;
00010 quit;
```



Sub-Dividing the Bars

The SUBGROUP option allows us to use another variable to divide the bars:

```
Program Editor - gr046
Command ===>
00001 *** Sub-Dividing The Bars Example;
00002
00003 goptions reset=global reset=all;
00004
00005 proc gchart data=saved.demograf;
00006    vbar status / sumvar=salary type=mean
00007    midpoints='M' 'S' 'D' 'SEP'
00008    subgroup =gender;
00009    where children le 2;
00010 run;
00011 quit;
```

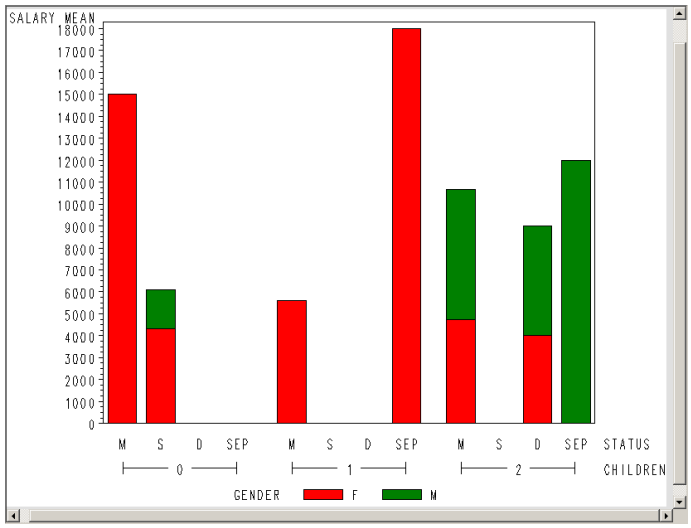


Note that an automatic legend is produced when subgroup is specified.

Grouping Bars

In addition to sub-dividing bars, they can also be grouped together using the GROUP option. This is displayed below.

```
Program Editor - gr047
Command ===>
00001 *** Grouping Bars Example;
00002
00003 goptions reset=global reset=all;
00004
00005 proc gchart data=saved.demograf;
00006    vbar status / sumvar=salary type=mean
00007    midpoints='M' 'S' 'D' 'SEP'
00008    subgroup =gender
00009    group=children;
00010    where children le 2;
00011 run;
00012 quit;
```



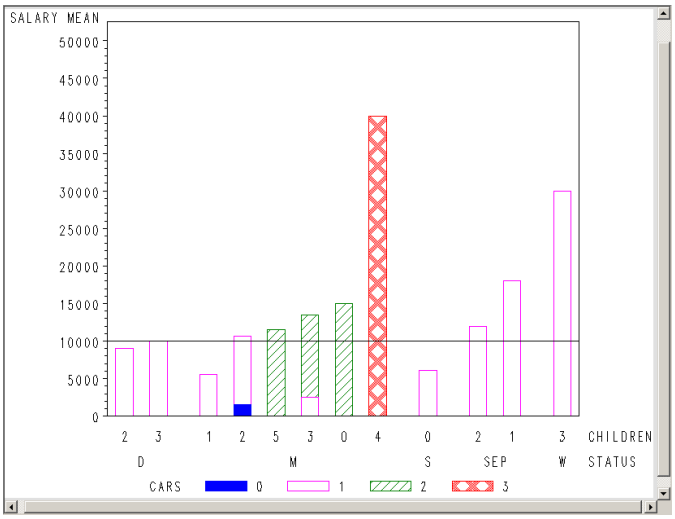
Additional Options

Additional options are available that can control the appearance of the report.

The following example illustrates Frame, Gspace, Space, Ref, Patternid, Nozeros, Ascending, Sum and Raxis.

```
Program Editor - gr049
Command ===>
00001 *** Other Options Example;
00002
00003 goptions reset=global reset=all;
00004
00005 pattern1 v=S  c=b;
00006 pattern2 v=E  c=m;
00007 pattern3 v=R1 c=g;
00008 pattern4 v=X5 c=r;
00009
00010 proc gchart data=saved.demograf;
00011    vbar children / frame gspace=2 space=2 ref=10000
00012                    sumvar=salary discrete type=mean
00013                    patternid=subgroup
00014                    subgroup=cars group=status
00015                    nozeros ascending sum
00016                    raxis=0 to 50000 by 5000;
00017 run;
00018 quit;
```

The above code has four pattern statements (lines 5-8) and the pattern id (line 13) changes patterns by sub-group.



The graph used four different patterns.

The space between the bars and the space occupied by the bars is set to two.

A reference line is drawn at 10,000, and the bars are drawn in ascending order.

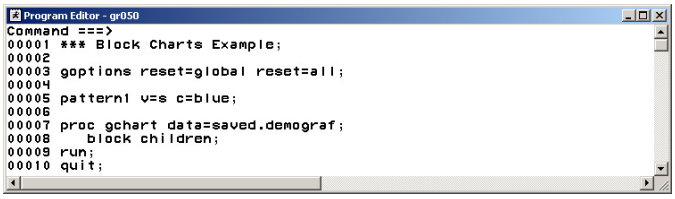
No space is left for non-existent bars, and patterns change across the sub-groups.

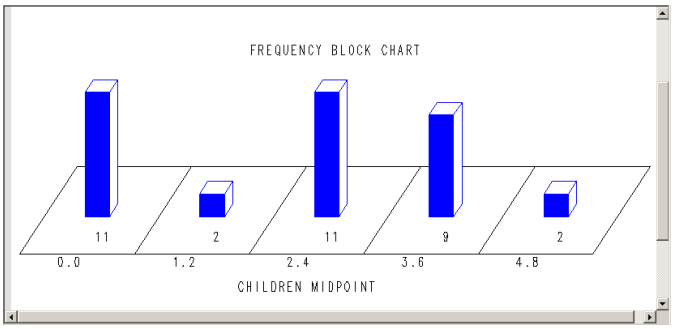
The response axis is ordered with the Raxis option.

Block Charts

Block Charts are specified with the Block statement.

They use more space on the graphics area than a Vbar or Hbar and you may need to increase Hpos and Vpos. This is displayed on the next page.

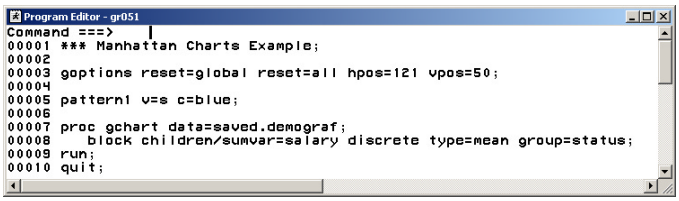


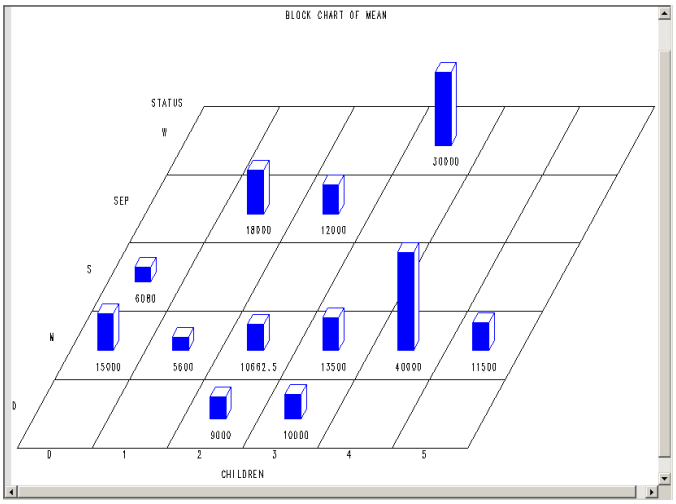


Manhattan Charts

The GROUP option produces Manhattan Charts.

Note how the use of the discrete option changes the graph.





Also called Manhattan Charts, this Grouped Block Chart displays the data across the two axes.

Note the values of Hpos and Vpos.

The original values of Hpos=80, Vpos=62 have been increased by 50% each.

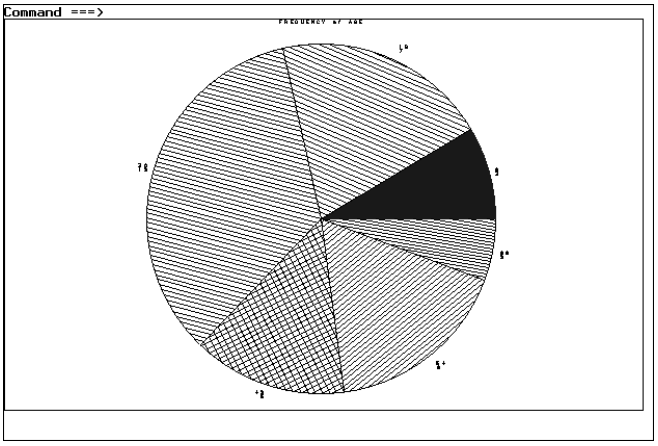
The aspect ratio then remains the same.

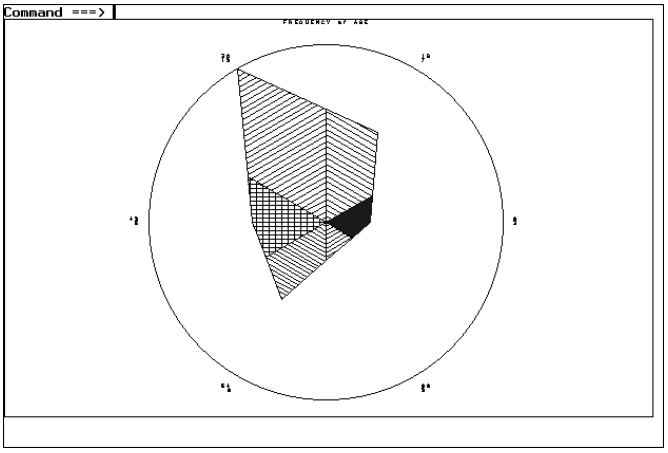
Calculations: 80+40 = 120 62+31=93

Pie Charts

Pie Charts can be displayed using pies and stars.

Both types are capable of displaying data in a circular pie form.





Pie and Star charts are specified in a fashion similar to Vbar, Hbar and Block Charts:

Star Variable
Pie Variable

Default Option: OTHER

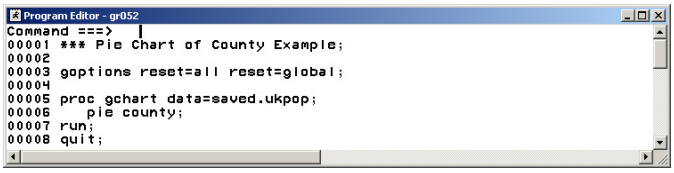
The option OTHER specifies a value of 4%, by default.

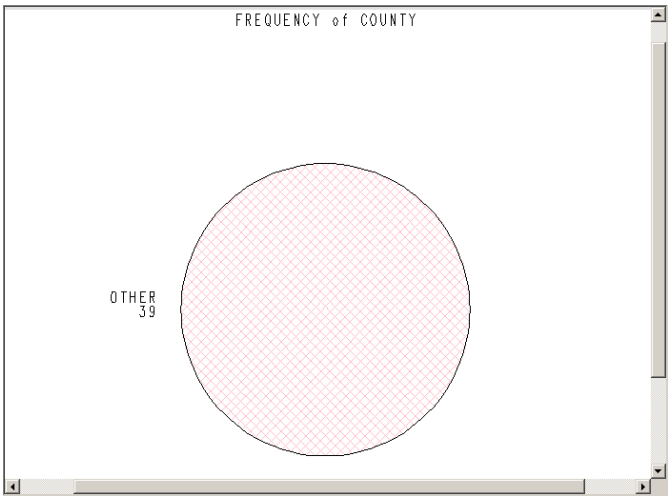
If you do not change this value, then any slice of the pie, which contributes less than 4%, is grouped together with all the OTHER small values.

This will be illustrated in the following examples.

Pie Chart of County

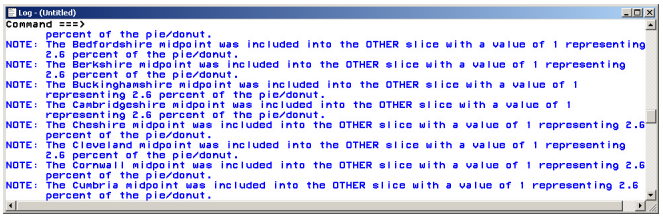
Let's create a Pie Chart displaying the counties of the UK.





When Pie charts are drawn for numeric variables, the Discrete option needs to be applied.

Since all the values in the data were under 4% of the total, they have been grouped together into the OTHER category.
Review the log.

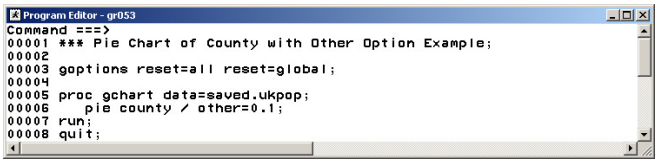


The notes state that all the counties were included in the OTHER slice because none of them represented more than 2.6% of the total pie.

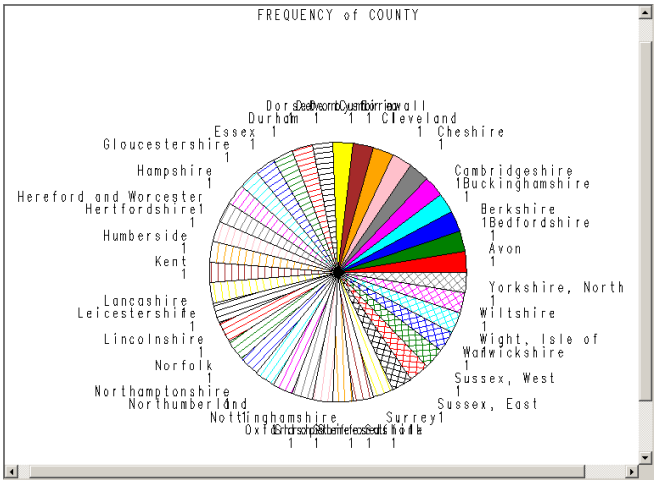
In such a situation, we incorporate the other option where we specify the value of the other slice.

We need to adjust the other slice in relation to the values that we would like to display as slices of the pie.

Let's set the value for Other at 0.1. This is demonstrated below.

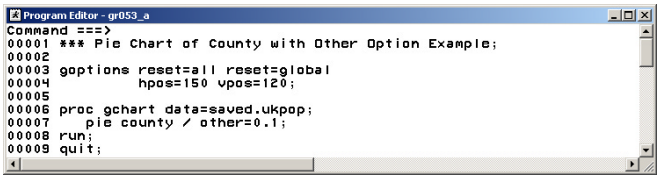


The output is displayed below:

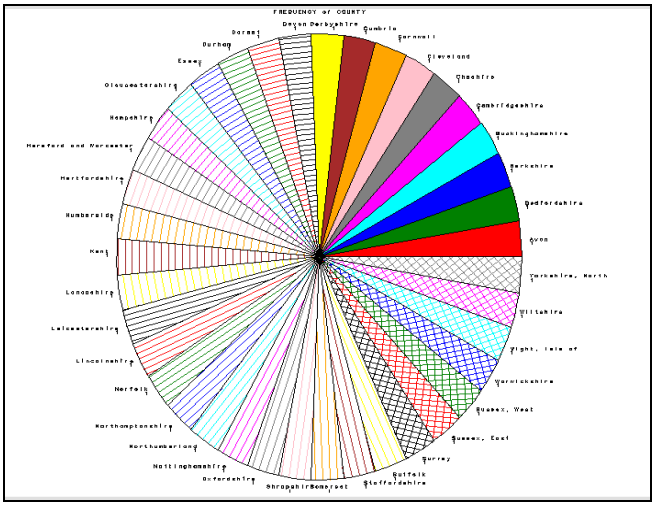


All the slices of the pie are displayed. The display is not clearly legible.

Let's increase the hpos to 150 and vpos to 120 and re-submit the code.



This addresses the spacing issue and makes the graph legible.

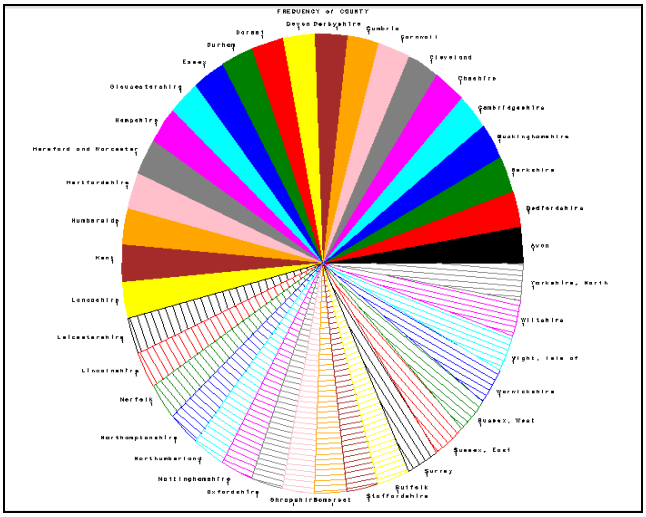


Now let's fill all the slices in the pie with solid colors.

Let's add a pattern option, as follows:

```
Program Editor - gr053_b
Command ===>
00001 *** Pie Chart of County with Other Option Example;
00002
00003 goptions reset=all reset=global
00004          hpos=150 vpos=120;
00005          pattern1 v=s;
00006
00007 proc gchart data=saved.ukpop;
00008    pie county / other=0.1;
00009 run;
00010 quit;
```

We see that about half the total number of slices in the pie are solid. This is because the large number of slices requires more than one pattern specification.

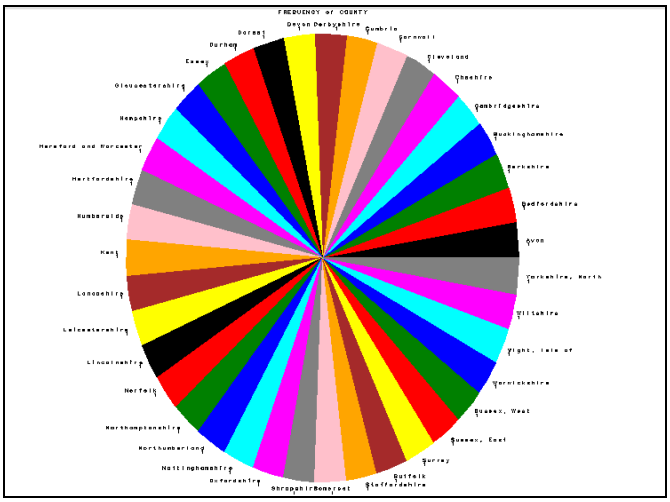


We need to keep adding pattern statements (pattern1, pattern2, pattern3) until all the slices are solid.
In this example, we need to add three pattern statements to make all the slices in the pie solid colors.

There are so many slices in this pie that they have exhausted three patterns.

```
Program Editor - gr053_c
Command ===>
00001 *** Pie Chart of County with Other Option Example;
00002
00003 goptions reset=all reset=global
00004          hpos=150 vpos=120;
00005          pattern1 v=s;
00006          pattern2 v=s;
00007          pattern3 v=s;
00008
00009 proc gchart data=saved.ukpop;
00010    pie county / other=0.1;
00011 run;
00012 quit;
```

Finally, we see that all the slices have been assigned solid colors.



Review the log and you will see messages stating that not all the slices in the pie were labeled.

```
Log - (Untitled)
Command ===>
WARNING: Text boundaries for the pie/donut slices overlap.
NOTE: Two or more pie/donut slice labels have been overwritten. This can occur
      several adjacent small slices. To avoid overwriting labels you can use on
      following techniques:
      1. Increase GOPTIONS HPOS= and VPOS= to make the text smaller.
      2. Reduce the size of the text with GOPTIONS HTEXT=.
      3. Use a different labeling method and perhaps a legend.
      4. Use the OTHER= option on the PIE/PIE3D/DONUT statement to group small
         single category.
      5. Use the MIDPOINTS= or ANGLE= options on the PIE/PIE3D/ DONUT statement
         order of slices or starting angle for the first slice so that two smal
         fall next to each other.
      6. Use a VBAR or HBAR chart instead of a PIE/DONUT chart.
      7. Use the graphics editor to reposition or resize the labels.
      Consult the proper documentation for details on any of these techniques.
```

This is addressed in the following section.

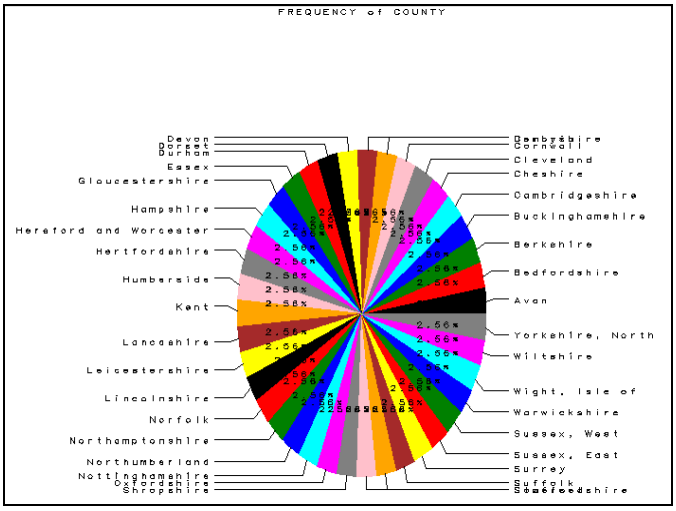
Labeling just the Midpoint

Let's take the previously submitted code and add the slice, value and percent options.

```
Program Editor - gr054
Command ===>
00001 *** Labeling Options Example;
00002
00003 goptions reset=all reset=global hpos=85 vpos=85;
00004          pattern1 v=s;
00005          pattern2 v=s;
00006          pattern3 v=s;
00007
00008 proc gchart data=saved.ukpop;
00009
00010    pie county / other=0.1 slice=arrow value=none percent=inside;
00011 run;
00012 quit;
```

Submit the code.

We see that the output is displayed as follows.



Regional Percentages

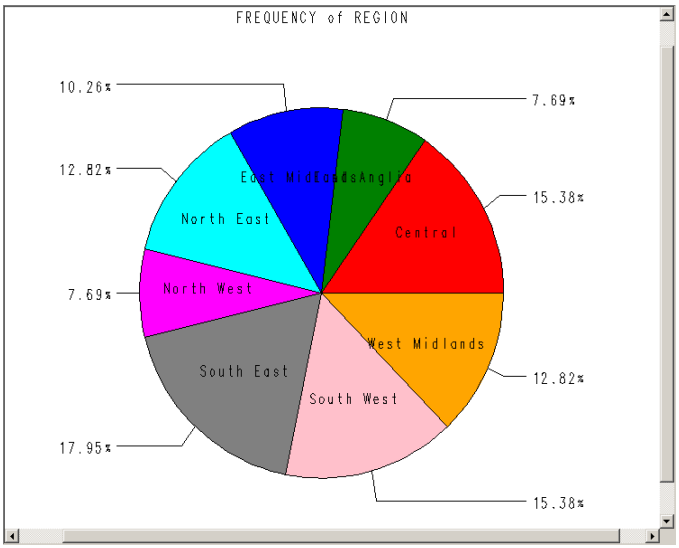
In the previous examples, by default the frequency was displayed on the charts.

When the SUMVAR option is absent, the frequency, or the number of observations in the data set, is charted.

Let's label each slice of the pie, as displayed in the following code.

```
Program Editor - gr055
Command ===>
00001 *** Regional Percentages Example;
00002
00003 goptions reset=all reset=global;
00004
00005 proc gchart data=saved.ukpop;
00006    pie region / other=0.1 slice=inside value=none percent=arrow;
00007 run;
00008 quit;
```

We see that there are no warning messages in the Log.



Contour Plots

Contour plots present output in three-dimensional forms.

There are two procedures used for plotting in three dimensions:

1. GCONTOUR
2. G3D

In GCONTOUR the response to two independent variables is displayed as different contour lines

G3D is a 3-dimensional perspective representation, either as a 'sheet' of joined points or a scatter plot.

In this section, we will demonstrate how to produce the various shapes of a plot.

It is beyond the scope of this course to explain all the options and features available.

The next sections explain how to create basic shapes and enhance plots using features like patterns, axis and legend specifications and definitions of different plotting symbols.

The data set used for our examples is typical of monitoring drug dosages.

The two independent variables used are the dosage, which is recorded in milligrams and the frequency of dosage, which is shown in hours.

The response to the dosage and frequency is measured in terms of the patients' pulse rate.

Here is a sample of the data:

```
Program Editor - gr072
Command ===>
00001 *** Contour Data Example 1;
00002
00003 proc print data=saved.contour;
00004 run;
```

Obs	DOSE	REGULARY	PULSE
1	40	4	115
2	40	5	130
3	40	6	130
4	40	7	134
5	40	8	140
6	50	4	115
7	50	5	120
8	50	6	122
9	50	7	128
10	50	8	140
11	60	4	95
12	60	5	99
13	60	6	115
14	60	7	120
15	60	8	133

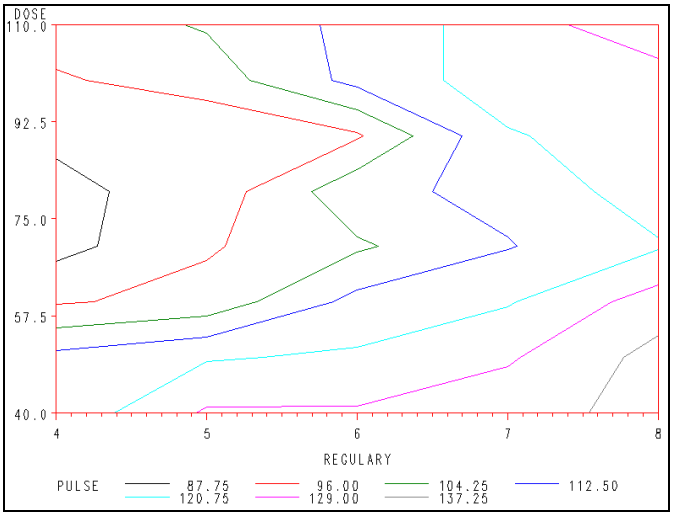
Dose is measured in milligrams and regulary is measured in number of hours.

The goal here is to lower the patients' pulse rate as much as possible based on the combination of dosage and regulary.

The first example shows the default contour plot produced with the Y*X=N form of plot statement.

The vertical axis is the Y, the horizontal is the X and the contour, or the response axis, is represented by values of the N.

```
Program Editor - gr073
Command ===>
00001 *** Contour Example 2;
00002
00003 goptions reset=all reset=global;
00004
00005 proc gcontour data=saved.contour;
00006    plot dose*regulary=pulse;
00007 run;
```



The SAS System has chosen the line types, axis gradations and contour steps.

We can infer a pattern from the data that the patient response is getting better with increased dose and regularity.

We can also see that dosages greater than around 75 milligrams at 4-hour intervals (regulary) do not decrease the pulse rate - in fact, it increases again.

The goal is to find the point at which the pulse rate is minimized. This trend will be shown in many ways in the following graphs.

This example shows that for a given regularity of dosage, increasing dosage results in a decrease in pulse rate. After reaching a certain dosage level pulse rates start to rise again.

The most effective dosage seems to be 75 mg at four-hour intervals.

Increasing the dosage to 110 mg has a less positive response.

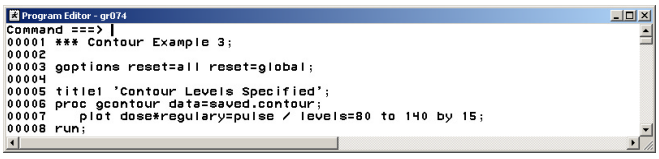
LEVELS= Option

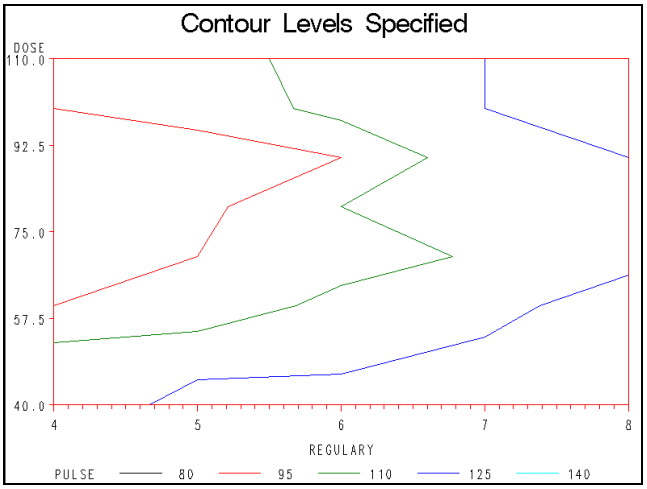
The first way we can enhance the plot is to specify the contour levels.

This is done by the LEVELS= option on the plot statement.

As usual, options on a plot statement follow a /.

The levels are going to start at 80 and increase in increments of 15 up to 140. The legend reflects this information.



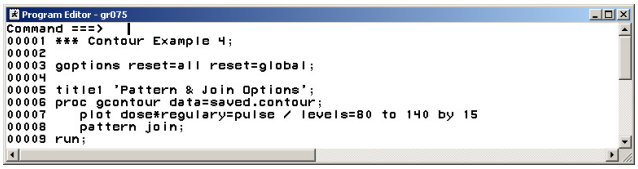


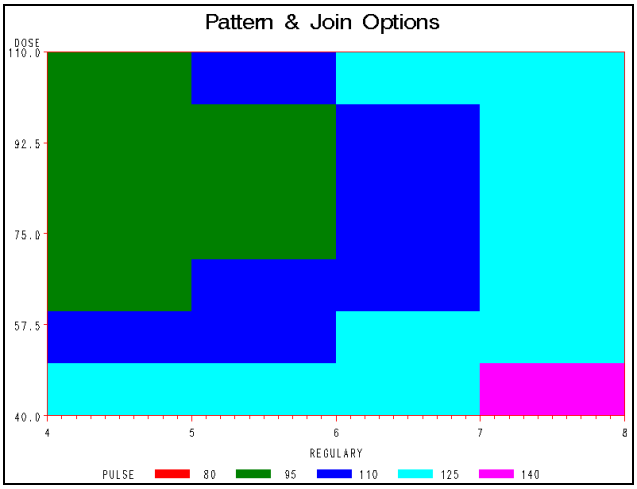
Note that the start value (80) is below any value in the data, and a note in the LOG states the same.

Pattern Option

The pattern option displays the graph as 'bricks' of X and Y combinations. The contour passes through it to be patterned according to value.

The join option joins together areas of equal response.





3-D Plots

PROC G3D can depict three-dimensional graphs.

PROC G3D generates two types of plots - 'sheets' of joined points across a surface and scatter plots.

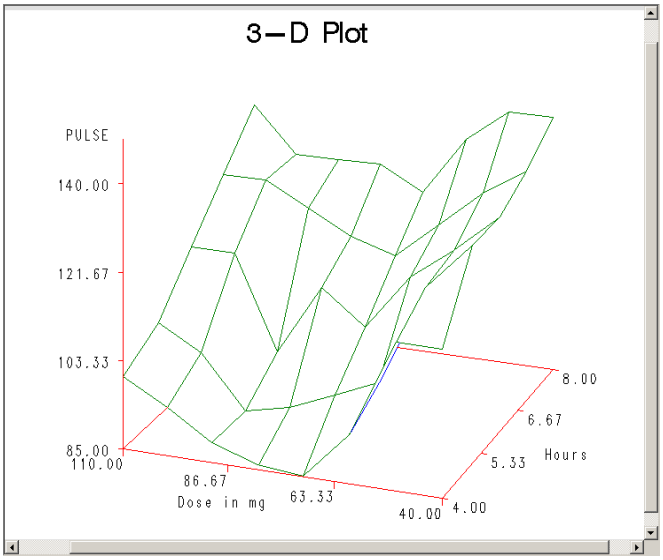
Note that if any interpolation is required between the joined points of the sheet, the data must first be processed using PROC G3GRID.

Using the same drug dosage data, we can display the relationship between the two independent variables (DOSE and REGULARY) and the dependent response variable (PULSE).

Using the code below, we get a three-dimensional presentation of the information where the exact points are joined together and graphed.

```
Program Editor - gr076
Command ===>
00001 *** G3D Example 1;
00002
00003 goptions reset=all reset=global;
00004
00005 title1 '3-D Plot';
00006 proc g3d data=saved.contour;
00007    label dose='Dose in mg' regulary='Hours';
00008    plot dose*regulary=pulse;
00009 run;
```

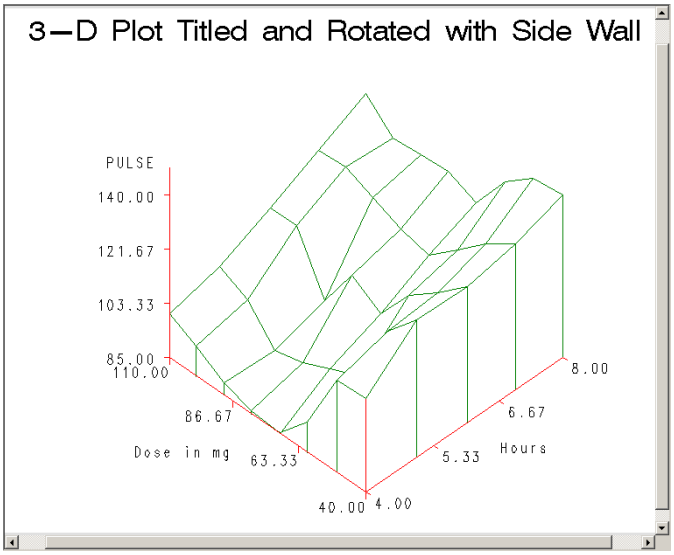
The example shows the very simple plot statement, again in the form Y*X=N.



Our first enhancement is to rotate the plot around the Z-axis using the rotate option and tilt it towards the observer by tilting around the Y-axis.

Side 'walls' will also be displayed for the sides of the plot.

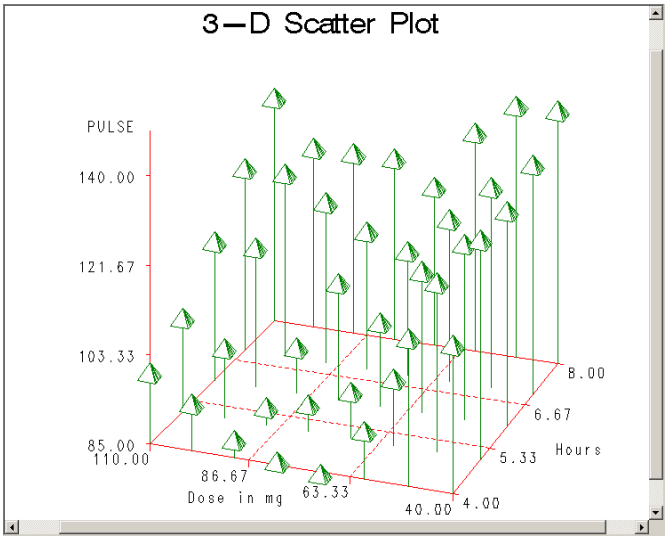
```
Program Editor - gr077
Command ===>
00001 *** G3D Example 2;
00002
00003 goptions reset=all reset=global;
00004
00005 title1 '3-D Plot Titled and Rotated with Side Wall';
00006 proc g3d data=saved.contour;
00007    label dose='Dose in mg' regulary='Hours';
00008    plot dose*regulary=pulse / rotate=45 tilt=45 side;
00009 run;
```



The next example shows a very simple example of the scatter plot.

```
Program Editor - gr078
Command ===>
00001 *** G3D Example 3;
00002
00003 goptions reset=all reset=global;
00004
00005 title1 '3-D Scatter Plot';
00006 proc g3d data=saved.contour;
00007    label dose='Dose in mg' regulary='Hours';
00008    scatter dose*regulary=pulse;
00009 run;
```

To produce this form simply substitute SCATTER for plot and specify the graph in the Y*X=N form.
\



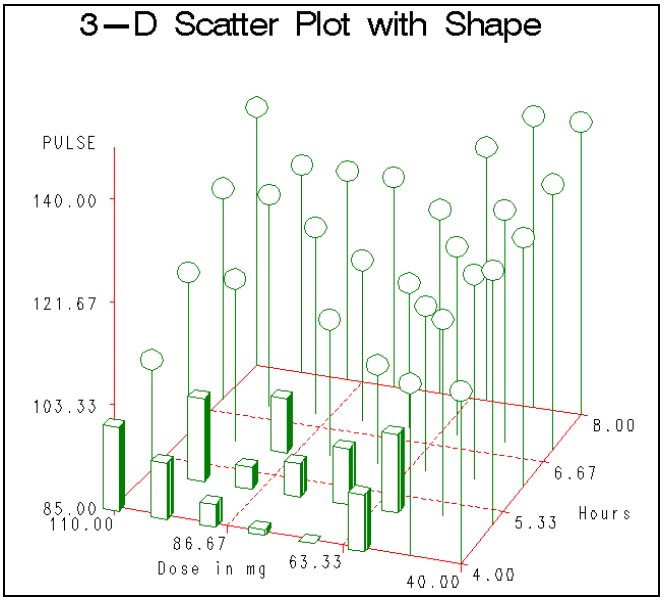
The pyramid shapes are produced by default.

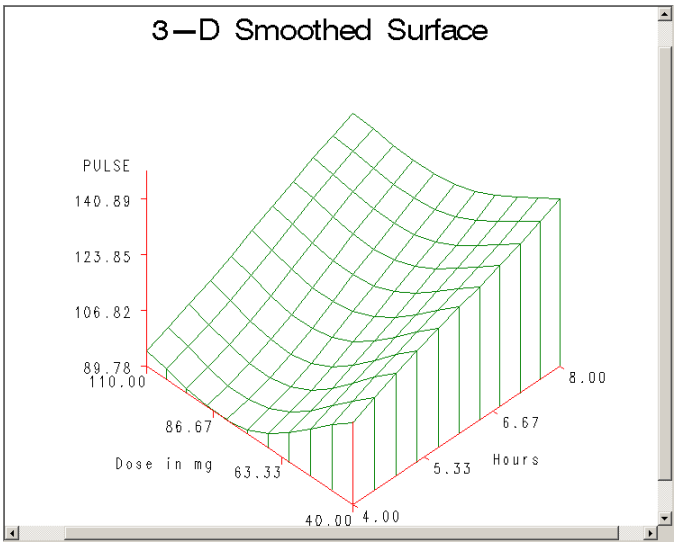
We can change the pyramid shape by using the shape option and specifying a variable in the input data set.

In the next example, we have specified a prism for values of PULSE of 100 and below.

Values above these values are assigned the balloon shape.

```
Program Editor - gr079
Command ===>
00001 *** G3D Example 4;
00002
00003 goptions reset=all reset=global;
00004
00005 data contour2;
00006    set saved.contour;
00007    if pulse le 100 then shape='PRISM';
00008                    else shape='BALLOON';
00009 run;
00010 title1 '3-D Scatter Plot with Shape';
00011 proc g3d data=contour2;
00012    label dose='Dose in mg' regulary='Hours';
00013    scatter dose*regulary=pulse / shape=shape;
00014 run;
```





G3GRID Procedure

We can use the G3GRID to smooth the values in our input data set and produce an improved G3D.

See the user guide for the amazing algorithm!

```
Program Editor - gr080
Command ===>
00001 *** G3GRID Example;
00002
00003 goptions reset=all reset=global;
00004
00005 proc g3grid data=saved.contour out=smooth;
00006    grid dose*regulary=pulse/smooth=1 spline;
00007 run;
00008 title1 '3-D Smoothed Surface';
00009 proc g3d data=smooth;
00010    label dose='Dose in mg' regulary='Hours';
00011    plot dose*regulary=pulse / tilt=45 rotate=45 side;
00012 run;
```

Internet, Intranet & the Web

Section Chairs

Caroline Bahler
Meridian Software, Inc.

Jimmy DeFoor
Brierly and Partners

Internet, Intranet & the Web

HTML for the SAS® Programmer

Lauren Haworth, Genentech, Inc., San Francisco

➢ ABSTRACT

With more and more output being delivered via the Internet, a little knowledge of HTML can go a long way toward improving the appearance of your output. This paper introduces some simple HTML coding techniques that are useful for SAS programmers. The paper will show how your SAS output is converted into HTML, and demonstrate HTML tricks that you can use in your SAS code to dress up your SAS HTML output.

The examples in this paper are based on SAS version 8.

➢ INTRODUCTION: HOW TO CREATE HTML OUTPUT FROM SAS

There are several ways you can move your SAS output to the web. If you are going to be doing a lot of web publishing, you may want to investigate the SAS/InterNet® product. However, if you just want to publish the occasional table, report, or printout on the web, you can do this with SAS/BASE®.

To put your output on the web, you need to convert standard SAS output into HTML files. HTML stands for HyperText Markup Language, and it is the common language understood by web browsers like Netscape and Internet Explorer.

To create HTML-formatted results, all you have to do is use the Output Delivery System and specify HTML as the output format. A simple example is shown below:

```
ODS HTML BODY='myfile.htm';
   PROC PRINT DATA=TEMP NOOBS LABEL;
      VAR JobType Salary;
   RUN;
ODS HTML CLOSE;
```

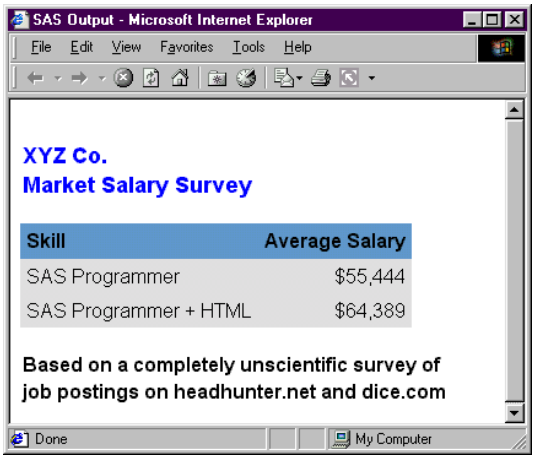
➢ WHAT YOUR OUTPUT LOOKS LIKE ON THE WEB

When you create an HTML file from your SAS output, the usual SAS output format of text and numbers separated by spaces is "marked up" with HTML "tags." These tags tell Internet browsers how to display your output. They identify headers and footers, specify fonts and type sizes, reference images, and point to other locations on the web (hyperlinks).

For example, the following PROC PRINT output …

```
XYZ Co.
Market Salary Survey

                          Average
Skill                     Salary

SAS Programmer            $55,444
SAS Programmer + HTML     $64,369

Based on a completely unscientific
survey of job postings on
headhunter.net and dice.com
```

… would look like this on the web …



… and would contain the HTML code shown in Figure 1. The figure is considerably simplified, with a number of formatting tags removed for the sake of clarity.

FIGURE 1: HTML CODE

```
<HTML>

<HEAD>
<TITLE>SAS Output</TITLE>
</HEAD>

<BODY>

<TABLE><TR><TD>XYZ Co. </TD></TR>
<TR><TD>Market Salary Survey</TD></TR></TABLE>

<P>

<TABLE>

<TABLE><THEAD><TR><TD>Skill</TD><TD>Average Salary</TD></TR></ THEAD >

<TBODY><TR><TD>SAS Programmer</TD><TD>   $55,444</TD></TR>
<TR><TD>SAS Programmer + HTML</TD><TD>   $64,389</TD></TR>
</TBODY></TABLE>

<P>

<TABLE><TR> <TD>Based on a completely unscientific survey of</TD></TR>
<TR><TD>job postings on headhunter.net and dice.com</TD></TR></TABLE>

</BODY>

</HTML>
```

<HTML> </HTML> surrounds all code, and identifies it to the browser as HTML code.

<HEAD> </HEAD> surrounds your header. In this section, you will find general information that applies to the entire file. The <TITLE> </TITLE> tags identify the text to display in the browser's title bar when the page is viewed.

<BODY> </BODY> tags enclose the "guts" of your HTML output. All of the content that will be displayed on the page is contained between these two tags.

<TABLE> </TABLE> tags enclose a table that is used to align the two titles for this output. The <TR> </TR> tags mark the beginning and end of each table row. The <TD> </TD> tags mark the beginning and end of each table cell in the row. Table tags are also used to align the results and the footnotes.

<P> </P> marks a break between paragraphs. It puts a large break between the lines above and below. If you don't want such a large break, use the <BR> tag

The <THEAD> </THEAD> tags mark the beginning and end of the heading row of the table. Text in these cells is displayed in bold in the browser.

The <TBODY> </ TBODY > tags mark the beginning and end of the body rows of the table. Text in these cells is not displayed in regular weight (not bold) in the browser.

As you review the information in Figure 1, you will see that the HTML file is composed of the basic text of the SAS output (shown in italics), surrounded by numerous HTML tags enclosed in angle brackets (shown in bold face).

If you look closer, you will also see that HTML tags come in pairs. If you look at the very first tag on the page: <HTML>, you will see that there is a matching </HTML> tag at the very bottom of the file.

Figure 1 explains the meaning of each of the tags in the file. This is just a small sampling of HTML tags; there are dozens more. However, if you can understand the tags used here, you will understand 80-90% of the tags you will find in more complicated web pages.

➢ TRICK #1: ADDING A HYPERLINK TO YOUR FOOTNOTE.

Now that you know a little about HTML tags, it's time to turn this knowledge to our advantage. We can use a couple of HTML tags in our FOOTNOTE statement to add a hyperlink to our output. This is a useful trick because it allows you to annotate your output. You can use the footnote to list the source of your data, and then attach a hyperlink so when the viewer clicks on the footnote, it automatically links them to the web site for that source.

To do this, you need to learn a new HTML tag. The tag for a hyperlink is:

```
<A HREF="xxx">text</A>
```

The "xxx" is where you put the information on where you want the hyperlink to go. This needs to be a valid URL (Universal Resource Locator) like "http://www.sas.com" or "http://www.mycompany.org/sales.htm".

The "text" is where you put the text for the hyperlink. This is the text that will be displayed on the page (with an underline to indicate that this is a hyperlink). For example, you could use "SAS Institute" or "January 2001 Sales Figures."

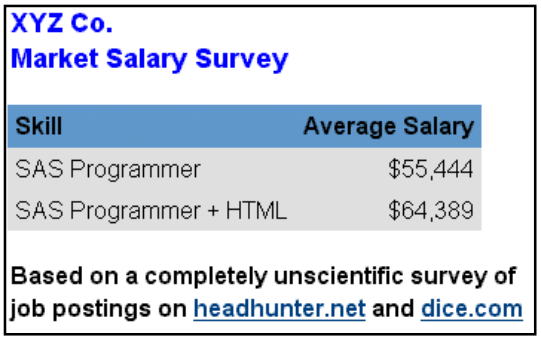
To see this code in action, we can add the following hyperlinks to the footnote statement from our previous examples. The original FOOTNOTE statement for the second line of the footnote was:

```
FOOTNOTE2 'job postings on
  headhunter.net and dice.com';
```

The revised FOOTNOTE statement with hyperlinks added is:

```
FOOTNOTE2 'Job postings on
  <A HREF="http://www.headhunter.net">
  headhunter.net</A> and
  <A HREF="http://www.dice.com">
  dice.com</A>';
```

When you run this code, SAS passes the HTML tags along with the rest of the text for the footnote. When the file is viewed with a browser, these tags are interpreted, and the footnote is displayed with a hyperlink. This is how the file looks when viewed with a browser:



One warning: version 8.1 will sometimes fail to correctly convert the HTML tags in your footnotes. Any tags you put in titles work fine. The problem has been corrected in the 8.2 release.

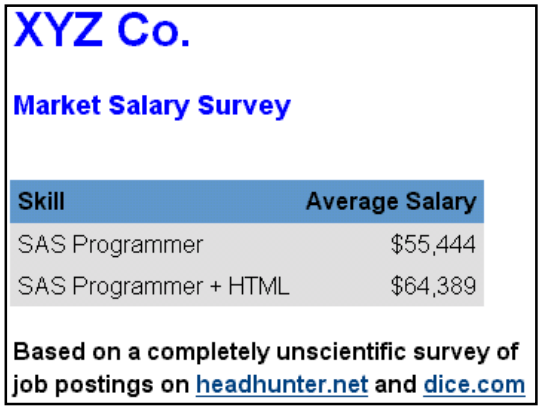
➢ TRICK #2: CHANGING THE TITLES

Another HTML tag that we can use to our advantage is the header tag. This tag identifies the level of the header. Lower levels (<H1> or <H2>) create big, bold titles. Higher levels create smaller titles. SAS picks a size for your titles based on the ODS style you are using.

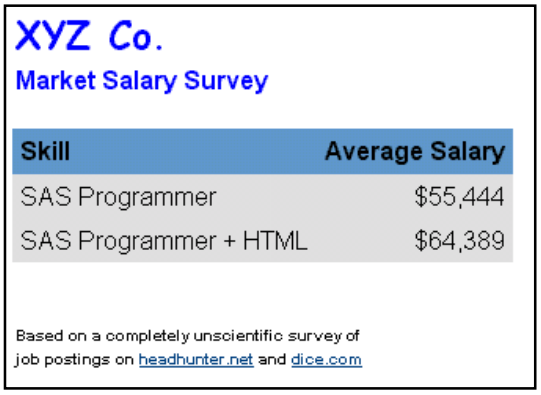
However, if we want a bigger title, all we have to do is add header tags to the text in our titles. The following code shows how this is done:

```
TITLE "<H1>XYZ Co.</H1>";
TITLE "<H2>Market Salary Survey</H2>";
```

This code uses the <H1> </H1> tags on the first title for a really big title, and the <H2> </H2> tags for the second title for a slightly smaller title. The results are shown below:



The results are shown below:



➢ TRICK #3: CHANGING THE FONTS

The titles in the previous example are certainly bigger, but they're not very attractive. This is because each web browser has a default font and point size for each heading level. Unfortunately, the fonts are pretty basic.

This example will show how to specify the font and point size for each title, instead of just using the standard heading definitions.

To do this, you need to learn a new pair of HTML tags: <FONT> </FONT>. What these tags do is control the appearance of the text between the two tags. You can specify the typeface, color, and size, among other attributes.

For this example, we're going to put some tags into our titles to select two new typefaces and point size settings. The syntax is:

```
TITLE "<FONT FACE='Comic Sans MS'
    SIZE=5>XYZ Co.</FONT>";
TITLE2 "<FONT FACE='Arial'
    SIZE=3>Market Salary Survey</FONT>";
```

Notice how the two parameters for the FONT tag are used. The FACE= tag is used to specify the typeface. The SIZE= tag is used to specify the font size. A SIZE setting of 3 is the standard font size used for the rest of the text on the screen. By using a larger setting for the first title, it stands out from the rest of the text.

The technique can also be used on footnotes. In this case, it would be nice if the footnote were smaller, so it wouldn't draw so much attention. The following code changes the footnote to the typeface Arial and the size to 1, which is much smaller.

```
FOOTNOTE "<FONT FACE='Arial' SIZE=1>
        the text goes here </FONT>";
```

➢ TRICK #4: CHANGING THE FONT COLOR

In addition to controlling font size and typeface, the FONT tag can also be used to change the color of your fonts. If you don't like these bright blue titles, you can change them to a more aesthetically pleasing color.

The syntax for assigning a font color is:

```
<FONT COLOR=color>
```

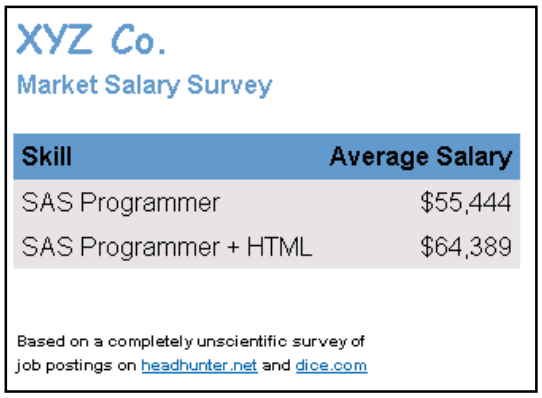
There are several ways you can define the color. First, you can use a color name. For example, to get a red title, you could assign the font to COLOR="red". However, there is a limitation here. Browsers vary in terms of which color names they understand, and also in how those colors are rendered. "Red" may be bright and orangish in one browser and dull and bricklike in another browser.

To get exactly the color you want, the best way is to specify the color using RGB codes. These are hexadecimal values that indicate the amount of red, green, and blue to use to create your color. You can look up these codes in the SAS/GRAPH manual, or use a "Color Picker" on the internet (just use your favorite search engine to find one, there are dozens of free color pickers out there).

For this example, we'll use the RGB code for a lighter blue that matches the table heading to create a new color for our titles. The code for the title statement is:

```
TITLE "<FONT COLOR='#639ACE'
        FACE='Comic Sans MS' SIZE=5>
        XYZ Co.</FONT>";
TITLE2 "<FONT COLOR='#639ACE'
        FACE='Arial' SIZE=3>
        Market Salary Survey</FONT>";
```

The new output is shown below. The titles are now the same color as the table heading. (You'll have to take the author's word on this, as it's hard to see the difference in this black and white output).



➢ TRICK #5: ADDING A LOGO TO THE PAGE

One of the most powerful aspects of the HTML is its ability to display images along with the text on a page. Images are added to the page using the <IMG> tag.

This is a very simple tag to use. All you have to do is specify the image source by listing the file name for the image. Ideally, you should use a web-friendly image format like GIF or JPG for your images.

There are two ways you can add this logo to your output. The first is to add an <IMG> tag to your TITLE statement. The following code shows how to display an image in the top left corner of the page by adding an <IMG> tag to the first TITLE statement.

```
TITLE "<IMG SRC='xyzlogo.gif'
    HEIGHT=40>";
```

This IMG tag uses parameters to specify the file name to be used and the size to which the image will be expanded or compressed. It is important to specify a size, or you may end up with an image being displayed either too small to see or too large to fit on the screen. Picking the right size usually involves some trial and error.

The other way to add a logo is to create a new style definition. This approach allows you to add a logo at the top of every page of output, without having to edit all of the TITLE statements.

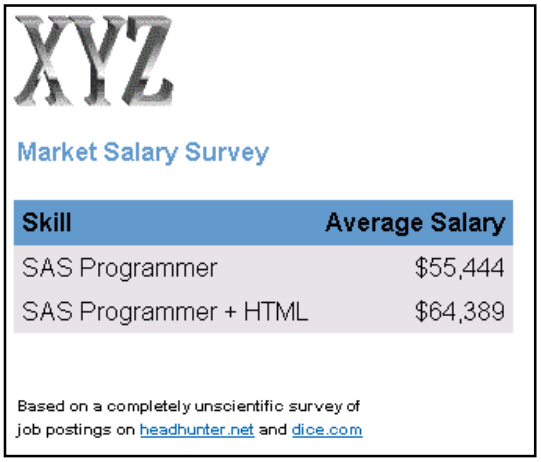
The code to create the new style definition is shown below:

```
proc template;
  define style MyHTMLStyle;
    parent=Styles.BarrettsBlue;
      replace Body from Document /
      prehtml="<IMG SRC='logo.gif'><br>";
  end;
run;
```

The PREHTML attribute is used to add the same <IMG> tag we used in the TITLE statement. The only difference is that putting it in the style means that it will be used at the top of every HTML page.

Now you can use this new style by adding STYLE=MyHTMLStyle to your ODS HTML statement.

Whether you use the TITLE approach or create a new style, your results will look like the output shown below.



➢ TRICK #6: ADDING AN IMAGE TO THE BACKGROUND

Each of the previous examples has shown some basic black text on a white background. However, if you've spent much time surfing the Web, you probably know that it's possible to use all sorts of colors and images for the page background.

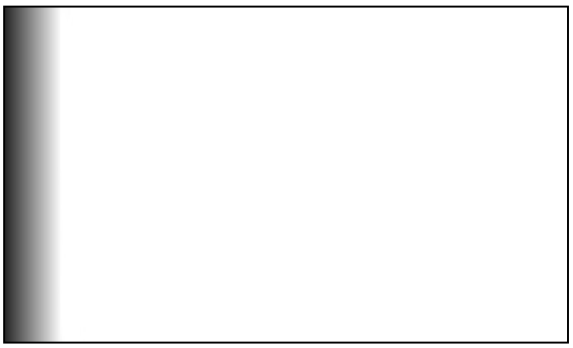
This next example will show how you can add a background image to your output.

To add a background image, we need to modify the <BODY> tag. We are going to add two new attributes to that tag.

First, we need to tell the browser what image to use. This is done using the following tag:

```
<BODY BACKGROUNDIMAGE=image>
```

The image is identified by file name, just as we did with the logo in the previous example. Again, web-friendly formats like GIF or JPG are best. The image we will use is shown below.



The other change we are going to make is to widen the left margin. The reason for this is that this image has a dark pattern on the left side, and we don't want our results to be hidden in the page background. If your background image does not detract from your results, you can skip this step.

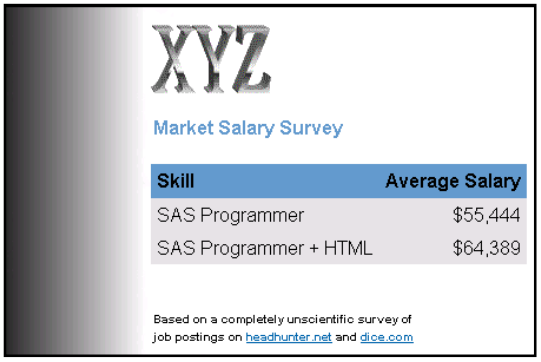
The syntax for a margin change is:

```
<BODY LEFTMARGIN=number>
```

To apply these two new settings, we are going to create a new style. Just as in the previous example, we'll use PROC TEMPLATE.

```
proc template;
  define style MyHTMLStyle;
    parent=Styles.BarrettsBlue;
    replace Body from Document /
      prehtml="<IMG SRC='logo.gif'><br>"
      backgroundimage='xyzback.jpg'
      leftmargin=125;
  end;
run;
```

We're adding attributes to the <BODY> tag by adding attributes to the Body style element. Notice how the style attributes are named to match the <BODY> tag attributes. This is true of a number of the SAS style element attributes. The new results follow. If you look back to the original version of this table on the first page of this paper, you will see that this is quite an improvement.



If you looked at the HTML code that was generated for this output, you would see that ODS has added two new parameters to the <BODY> tag in the file. The new body tag that controls the background is as follows.

```
<BODY BACKGROUND='xyzback.jpg'
      LEFTMARGIN=125>
```

Although you didn't have to specify this tag in order to use it, it's still helpful to review the code to see what tags ODS has used to build your output.

➢ Trick #7: Adding a Navigation Bar

The previous examples have shown how to add some fairly simple HTML code to your results. This next example will show how you can add some fairly elaborate features using the tags we have already learned.

This example will add a navigation bar at the bottom of the page. This is a series of links back to other parts of your web site. It's a useful tool to help users move around your site without getting lost.

To add the navigation bar at the bottom of the page, we will use the POSTHTML attribute of the Body style element. This works just like the PREHTML attribute we used earlier.

The hyperlinks that make up the navigation bar will be organized into a single-row table. The HTML code we will use is:

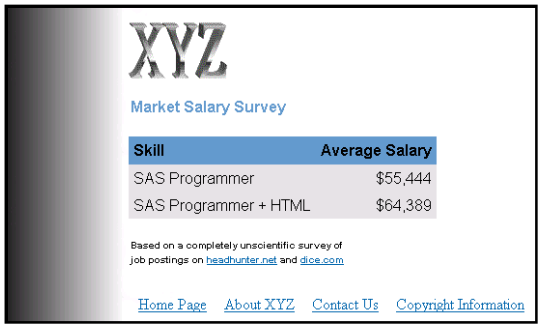
```
<BR><TABLE CELLPADDING=8><TR>
<TD><A HREF='home.htm'>Home Page
</A></TD>
<TD><A HREF='about.htm'>About XYZ
</A></TD>
<TD><A HREF='contact.htm'>Contact Us
</A></TD>
<TD><A HREF='copyright.htm'>Copyright
Information</A>
</TD>
</TR></TABLE>
```

These tags create a table with one row and four columns. Each column contains a hyperlink to a different part of the web site. The CELLPADDING is set to 8 pixels to put some space between the links.

This code is added to our style as follows:

```
proc template;
  define style MyHTMLStyle;
  parent=Styles.BarrettsBlue;
  replace Body from Document /
    prehtml="<IMG SRC='logo.gif'><br>";
    backgroundimage='xyzback.jpg'
    leftmargin=125
    posthtml="<BR><TABLE CELLPADDING=8><TR>
      <TD><A HREF='home.htm'>Home Page
      </A></TD>
      <TD><A HREF='about.htm'>About XYZ
      </A></TD>
      <TD><A HREF='contact.htm'>Contact Us
      </A></TD>
      <TD><A HREF='copyright.htm'>
      Copyright Information
      </A></TD>
      </TR></TABLE>";
  end;
run;
```

The new results are shown below:



➢ CONCLUSIONS

These examples are just a brief introduction to the world of HTML. You can use these tricks to spice up your HTML output.

However, if you're going to be creating a lot of HTML output from SAS, I strongly recommend that you take the time to learn more about HTML. Although SAS gives you great tools to create HTML output, you will find it a lot easier to use these tools if you can understand the resulting output.

Start with reading up on ODS styles and templates. These tools will give you a lot of control over your output. You'll want to pay special attention to the documentation on the PREHTML and POSTHTML attributes.

The references section lists several good tutorials and books to help you learn more about HTML. Also, see the Appendix for a listing of common HTML tags.

Finally, the next time you're out surfing the web, take a look at the HTML code behind the pages you're viewing. You can learn a lot by viewing other people's web sites.

Netscape and Internet Explorer both have menu options to view the source code for the page on the screen. You won't recognize all of the tags you see, but the basic tags presented here will show up again and again.

➢ REFERENCES

Free downloadable HTML tutorial:

http://www.pagetutor.com/pagetutor/index.html

Another tutorial (aimed at kids, but it's a lot of fun)

hotwired.lycos.com/webmonkey/kids/

Good books for HTML beginners:

Castro, Elizabeth, *HTML 4 For The World Wide Web Visual Quickstart Guide*, Peachpit Press

Powell, Thomas, *HTML: the Complete Reference*, McGraw Hill

Lemay, Laura, and Denise Tyler, *Teach Yourself HTML 4 in 24 Hours*, SAMS

➢ ACKNOWLEDGEMENTS

➢ CONTACTING THE AUTHOR

Please direct any questions or feedback to the author at: info@laurenhaworth.com

APPENDIX: SELECTED HTML TAGS

➢ BASIC TAGS

<html></html> Creates an HTML document
<head></head> Sets off the title and other information that isn't displayed on the Web page itself
    <title></title> Puts the name of the document in the title bar
<body></body> Sets off the visible portion of the document
    <body bgcolor=?> Sets the background color, using name or hex value
    <body text=?> Sets the text color, using name or hex value

➢ TEXT TAGS

<pre></pre> Creates preformatted text
<h1></h1> Creates the largest headline
<h6></h6> Creates the smallest headline
**** Creates bold text
**** Creates italic text
<**strong>** Emphasizes a word (with italic or bold)
<**font size=?>** Sets size of font, from 1 to 7
<**font color=?>** Sets font color, using name or hex value
<**font face=?>** Sets font typeface, using name (in quotes)

➢ LINKS

**** Creates a hyperlink

➢ FORMATTING

<p> Creates a new paragraph
<**p align=?>** Aligns a paragraph to the left, right, or center
<**br>** Inserts a line break
<**blockquote></blockquote>** Indents text from both sides
<**ol>
** Creates a numbered list
    <**li>** Precedes each list item, and adds a number
<**ul>
** Creates a bulleted list
    <**li>** Precedes each list item, and adds the bullet

➢ GRAPHICAL ELEMENTS

<img src="name"> Adds an image
<**img src="name" height=? Width=?>** Sets display size of an image
<**hr>** Inserts a horizontal rule

➢ TABLES

<table></table> Creates a table
<**tr></tr>** Sets off each row in a table
<**td></td>** Sets off each cell in a row
<**th></th>** Sets off the table header (a normal cell with bold, centered text)

➢ TABLE ATTRIBUTES

<table border=#> Sets width of border around table cells
<**table cellpadding=#>** Sets amount of space between a cell's border and its contents
<**tr align=?> or <td align=?>** Sets alignment for cell(s) (left, center, or right)
<**tr valign=?> or <td valign=?>** Sets vertical alignment for cell(s) (top, middle, or bottom)

Delivering Information Everywhere using JSP and SAS®

Bryan Boone, SAS® Institute Inc., Cary, NC
Pat Herbert, SAS® Institute Inc., Cary, NC

ABSTRACT

With the advent of the Internet and the increasing mobility of its users, it has become necessary to allow immediate access to data and reports from non-desktop (wireless) devices such as palm pilots and phones. In this paper we will show how SAS technology in conjunction with JSP and Servlets will keep your users connected to their business information on demand.

INTRODUCTION

The face of business is changing and corporate executives and managers must be mobile in order to meet the fast-paced changes in customer and market demands. To meet these requirements, executives are turning to wireless devices, such as the cell phone and Personal Digital Assistant (PDA), to remain in touch with the office while attending to customer needs.

The Cahners In-Stat Group forecasts that the number of non-desktop (wireless) users will surpass 1.3 billion by 2004. Additionally, it is projected that more than 1.5 billion handsets, PDAs, and Internet appliances will have wireless capability by the end of 2004.

Wireless communications and the Internet are becoming increasingly intertwined. Using the combination of SAS and Java technology, the mobile executive can have important sales and customer data delivered wirelessly to his or her cell phone, PDA or other non-desktop device.

WIRELESS TECHNOLOGY

Wireless Application Protocol (WAP) is a global standard for applications over wireless networks. It is just one of the wireless and handheld device standards that SAS is supporting to enable the mobile user to access his or her data more efficiently. In this paper we will be using WAP as the basis for many examples but the reader should keep in mind that other standards are supported equally. Within the WAP standard, Wireless Markup Language (WML) is used to create pages that can be displayed in a WAP browser. The WML markup language contains hierarchies of screens ("decks") and links between those screens ("cards"). WML is designed to display content on wireless devices such as phones, pagers, and PDAs.

A typical WAP request for a cell phone would entail:

1. The user presses a phone key that initiates a request.
2. The phone browser sends the request to a WAP gateway using the WAP protocol.
3. The WAP gateway creates an HTTP request for the specified URL and sends the request to the designated web server.
4. The web server then processes the HTTP request as it would any other HTTP request.
5. The web server returns the requested WML deck using either a static WML deck or a dynamically generated WML deck.
6. The WAP gateway verifies the HTTP header and WML content and encodes them to binary form. The gateway then creates a WAP response containing the WML and sends it to the phone browser.
7. The phone browser receives the WAP response, processes the WML response and displays the first card of the WML deck to the user.

JAVA TECHNOLOGY

Java is a software language that is platform neutral. It is said that Java is "Write Once, Run Anywhere ™ ". That is, an application developed in Java can be deployed to any machine that supports the Java™ Virtual Machine. Two important technologies that build on Java are Java™ Servlet and JavaServer Pages ™ (JSP ™). When used in conjunction with wireless technology, developers can use their knowledge of these technologies to provide dynamic delivery of data such as WML. Java's component-based technology, JavaBeans ™, makes it easier to build web pages using JSPs and Servlets. JavaBeans separate the user interface from the application logic. This enables the page designer to focus on writing the presentation layer while allowing the application developer to generate the dynamic content portion of the page using Java and JSP. Additionally, JavaBeans provides an integration standard. This is important for developing solutions in heterogeneous environments within the enterprise or across the Internet. SAS Institute Inc. adheres to this standard and offers component-based JavaBeans that allow easy access to complex SAS resources.

SAS TECHNOLOGY

SAS Institute Inc embraces Java technology. AppDev Studio™ the first Java-based development solution to be tailored for the information delivery environment, is a complete suite of application development tools for building thin-client Java applications. Applications written using AppDev Studio™ can tap into SAS resources through webAF™.

SAS's component technology, webAF™, makes it easy to use a standards-based approach to access SAS from the Web. webAF software is a Java framework that enables access to SAS/AF® objects, tables (data sets), multidimensional databases (MDDBs), and other SAS computing resources. webAF® provides data models that enable developers using JavaServer ™ Pages (JSP ™) to create dynamic content that maximizes the capabilities of the SAS System.

The power of webAF's JSP support lies in its InformationBeans™ and TransformationBeans. InformationBeans encapsulate SAS data by presenting it as webAF data models. The webAF data models are then, in turn, consumed by TransformationBeans, which transform the model into appropriate presentations. A key to integrating SAS into wireless technology lies in the server-side processing of the HTTP request. Here the TranformationBeans display SAS data for the appropriate wireless enabled device or PDA.

CONCLUSION

Changes in today's business market require corporate executives and managers to be more mobile. At the same time, cell phones and PDAs are becoming more common in the workforce and have a tremendous potential to provide more information to those mobile professionals. Wireless technologies facilitate the use of JSPs and Servlets on the web server, where the user's request is ultimately evaluated. SAS technologies, particularly InformationBeans™ and TransformationBeans found in webAF, deliver SAS data wirelessly to cell phones, PDAs and other non-desktop devices, keeping the user connected to his or her business at all times – providing "The Power to Know ™".

The complete copy of this paper can be found at
http://www.sas.com/usergroups/sugi/sugi26/sipapers

CONTACT INFORMATION

Your comments and questions are valued and encouraged.
Contact the authors at:

Bryan Boone
SAS Institute Inc.
SAS Campus Dr.
Cary, NC 27513
(919) 531-6039 (Voice)
(919) 677-4444 (Fax)
Bryan.Boone@SAS.com

Pat Herbert
SAS Institute Inc.
SAS Campus Dr.
Cary, NC 27513
(919) 531-7618 (Voice)
(919) 677-4444 (Fax)
Pat.Herbert@SAS.com

Using SAS/IntrNet™ Software

Kevin Davidson
FSD Data Services, Inc.

ABSTRACT

This paper will demonstrate the best methods of running SAS programs with SAS/IntrNet software. Examples of SAS programs executed via the web will be shown. Installation issues will be discussed, such as the optional settings of the Application Dispatcher and the efficiencies of each (the Application Dispatcher is the intermediary between the web browser and the SAS programs that retrieve or update data.) One topic will be customizing the broker configuration file and PROC APPSRV (the procedure which invokes the Application Server). Another will be the options for invoking SAS programs.

INTRODUCTION

SAS/IntrNet software allows you to build dynamic Web applications and gives you the ability to dynamically query your data, generate reports on demand, and perform many of the same analysis that you would run in an interactive SAS session. Obviously, taking the power of SAS software to the Web has tremendous advantages for a wide variety of companies. The purpose of this paper is to discuss some of the options within SAS/IntrNet software that make for an efficient operation and show how you can use the built in options to keep a fairly easily maintained web server.

In the ensuing pages, the following issues will be discussed:

Customization options:

Decide on socket, pool, or launch services

Broker.cfg
- Timeout option
- Administrator name and e-mail address
- Fieldwidth for textarea boxes
- Exporting fields
- Creating fields using SET statement
- Load Manager

PROC APPSRV
- ADMINPW option
- Unsafe option
- Short macro for simplifying the ALLOCATE, DATALIBS, and PROGLIBS statements.
- Request init
- Statistics
- Using datalib to use aggregate naming
- Log statement

A brief review of the plusses and minuses of 4 different program submit styles

SAS/IntrNet SOFTWARE

SAS/IntrNet software is composed of a number of components including Application Dispatcher, HTMSQL, and the MDDB Report Viewer. An Application Dispatcher server is a SAS session that serves out requests from thin clients and returns results via the browser. Parameters are passed to the Application Dispatcher and can be accessed in the form of macro variables. One of the parameters is '_program' which tells the Application Dispatcher which SAS program to run (usually either a standard .sas program or a compiled macro).

CUSTOMIZATION OPTIONS:

Deciding on type of service

The on-line documentation for SAS/IntrNet does an excellent job of presenting the pros and cons of the three different types of services. For discussion purposes, the following text on services is adapted from the on-line documentation.

A service can be a socket, pool, or launch service. The features, advantages, and disadvantages of each of these service types are discussed below.

Socket Services

Socket services consist of one or more Application Servers that run continuously, servicing client requests. Socket services are generally started whenever a machine is restarted (either manually or by an operating system mechanism for starting processes at boot or login time).

The service usually runs until the machine is shut down. Socket services are relatively simple to configure and manage and are adequate for most applications

Advantages

- Socket services are supported on all SAS/IntrNet platforms. Other service types are not supported everywhere.
- The server is already running by the time a client request appears, so clients do not have to wait for a server to start.
- The administrator has explicit control of resources allocated to the service: the administrator can control how many servers are run on each system and which resources are allocated to each server.
- Increasing load can be handled by adding more servers to the service.

Disadvantages

- Servers must be started and stopped manually or by the operating system. No automated start-up and shutdown service is provided by SAS/IntrNet software.
- No dynamic scaling to meet increasing loads is provided. A fixed number of servers are available to handle all client requests. A few long-running requests can slow the entire service for all clients.

Pool Services

Pool services consist of a pool of Application Servers shared by clients. Based on system loading the servers are started and stopped by the Application Load Manager. Numerous options are provided to fine-tune the operation of a pool service. Pool services combine some of the advantages of socket and launch services.

Advantages

- Servers are started as needed. If all servers in the service are busy, the Load Manager can start an additional server.
- Servers can be reused by new clients once they are started. A started server remains in the pool until an idle timeout is reached and the server is stopped.
- Unlike launch services, pool services can be on a different system than the Web server and can be distributed across multiple server systems.
- Using the SAS Spawner, servers can be started under specific usernames to control access to system resources.

Disadvantages

- Installation and configuration are more complex for pool services. The Application Load Manager must be installed. The SAS Spawner must be installed in most cases.
- Client requests might have to wait for a new server to start, although this is typically no worse (and could be better) than waiting for currently executing requests to complete in a socket service.

Launch Services

A launch service starts a new Application Server for each client request. An existing server is reused only for applications that use sessions or the `_tmpcat` catalog for IDS output. Most of the features of launch services are better provided by pool services (a new feature for Version 8). Launch services are not generally recommended for new installations.

Advantages

- Server start-up is automatic for each request. Once the launch service is configured, little or no additional administration is necessary.
- Requests run in a separate server, so a long-running request will not block access to the service for other clients.
- Many requests can run in parallel, assuming that the system will support the load.
- Ill-behaved applications that "crash" or "hang" a server will not affect other client requests.

Disadvantages

- Launch services are started by the Application Broker and must run on the same system as the Web server.
- Each new request incurs the resource overhead and delay of starting a new server session.
- Launch services are not suitable for high user loads. There are no settable limits on the server load. The service will attempt to start a new server for each new client. In an extreme case, 200 simultaneous users could cause 200 servers to be started, likely causing extreme "memory thrashing" and very slow response for all users. Most Web servers have limits on the number of simultaneous CGI requests that could help to control this problem.
- Each launch service request must incur the additional time for starting a SAS session.
- Launch services are not supported on CMS, OpenVMS, and OS/390 platforms.
- Launched servers can be difficult to shut down. A launched server that creates a session or `_TMPCAT` catalog will continue running until an idle timeout is reached. These servers cannot be shut down other than by interrupting the server process.

There are a number of things that can be done to customize your SAS/IntrNet configuration. Here we will focus on customizations that can be made within two files, broker.cfg and PROC APPSRV. Broker.cfg is the main configuration file.

Broker.cfg - The broker configuration file is the configuration file for the Application Broker. Among the items that you might find useful to alter are:

1) One of the options that can be set is the 'timeout' period, after which the end-user will see an error message. To manually set this option, put a line similar to the following anywhere in your broker.cfg:

Timeout=60

This sets the timeout option to 60 seconds. You will want to carefully analyze the expected times that your programs may take before setting this value. You need to take into account the speed of the machine, speed of connectivity, and amount of traffic. It is a good idea to periodically check the run times of your web program submissions to determine the causes of any slowdowns. Should an end-user experience a timeout they will see something similar to this:

> **Timeout error**
>
> The program ran longer than its allotted timeout period (60 seconds). This could happen if the timeout is too short, if the server is unavailable or busy, or there was an error invoking the SAS server. Note: program may still be running.
>
> *Kevin Davidson, KevinD@FSD.nu*
>
> *This request took 6.38 seconds of real time (v8.2 build 1391).*

Note that the Timeout option can also be set as a PROC APPSRV option.

2) Administrator name and e-mail address

In case of an error this name and e-mail address will appear so that an end-user can alert you. This data can easily be altered by changing the lines that read:

Administrator "Kevin Davidson"
AdministratorMail "KevinD@FSD.nu"

3) Field width of text area blocks

Any text area fields that you may set up in your HTML are chopped up into character block fields according to an option set in the configuration file. The default used to be 80 and although the SAS supplied comments in the 8.2 configuration file still indicate that the default is 80, it does not appear to be so (at least under Windows). Suppose your HTML page has an input field such as the following:

<textarea name=comment cols=80 rows=4 wrap=virtual>
This is a pretty long text field. We want to see where it wraps. The zero in 80 is the eightieth character in this line. We want to see if it splits it given the default length that is specified in the broker.cfg file.
</textarea>

There is an option in the broker.cfg called 'Fieldwidth'. If the Fieldwidth is set to 80, then upon submitting the above text to the Application Dispatcher, the symbols made available to SAS will look like the following:

comment=This is a pretty long text field. We want to see
where it wraps. The zero in
comment0=3
comment1=This is a pretty long text field. We want to
see where it wraps. The zero in
comment2=80 is the eightieth character in this line. We
want to see if it splits it
comment3=given the default length that is specified in the
broker.cfg file.

As you can see, the longer string is cut up into smaller fields of roughly 80 characters each. The 1st field listed above is the original input field name. The 'comment0' value of 3 indicates that the original variable was split into three different blocks. The three different blocks are represented by the numeric suffix being added to the original input field name.

You can alter this setting by entering a value such as the following:

Fieldwidth 32767

This will allow text area strings to be exported to SAS as one long character string (you will of course want to make use of the 'compress' data set option if you create extremely long character variables in your data set).

4) Exporting fields

If you examine the broker.cfg file, there is a series of potential export variables which you can make available to your SAS session (they will appear in the 'SYMBOLS' section of your logs). In version 8.2, the list of available export variables are as follows: Simply uncomment those that you want to make use of.

What is it?	Export variable name
CGI version	Export GATEWAY_INTERFACE _GATEWAY
Web server hostname	Export SERVER_NAME _SRVNAME
Web server name/version	Export SERVER_SOFTWARE _SRVSOFT
HTTP version	Export SERVER_PROTOCOL _SRVPROT
Web server port number	Export SERVER_PORT _SRVPORT
GET, POST, etc.	Export REQUEST_METHOD _REQMETH
Extra path info after script	Export PATH_INFO _PATHINF
Local filename of PATH_INFO	Export PATH_TRANSLATED _PATHTRN
Dup of _URL	Export SCRIPT_NAME _SCRIPT
Directory from which Web documents are served [unreliable]	Export DOCUMENT_ROOT _DOCROOT
Query string for GET requests [duplicate of user macro parms]	Export QUERY_STRING _QRYSTR
User's DNS name if known	Export REMOTE_HOST _RMTHOST
User's IP address	Export REMOTE_ADDR _RMTADDR
Usually Basic	Export AUTH_TYPE _AUTHTYP
Username if authenticated	Export REMOTE_USER _RMTUSER
RFC931 id if supported	Export REMOTE_IDENT _RMTID
HTTP POST type	Export CONTENT_TYPE _CONTTYP
HTTP POST length	Export CONTENT_LENGTH _CONTLEN
Email address of user making request [unreliable]	Export HTTP_FROM _HTFROM
MIME types UA likes [unreliable]	Export HTTP_ACCEPT _HTACPT
Cookies	Export HTTP_COOKIE _HTCOOK
Browser name	Export HTTP_USER_AGENT _HTUA
Referring page if known	Export HTTP_REFERER _HTREFER

As an example, if we have the "_HTUA" set to export, then your SAS programs will be able to make use of the variable and you can also automatically store the value in the STATISTICS data set (see below). Near the top of your log, you would see something similar to:

Thu, 31 May 2001 11:18:23.687 Request 12 Program is proglib.myprogram.sas
Thu, 31 May 2001 11:18:53.858 Request 12 ended okay (30.50 seconds)
========== Request 12 Symbols ==========
_SRVNAME=www.fsddatasvc.com
_RMTADDR=216.91.105.82
_HTUA=Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 4.0)

5) Creating fields using a SET statement

In addition to the SAS supplied export variables, you can also create your own. For example, if you have multiple web servers running Application Dispatcher(s), you may want to display in the log (and/or in the STATISTICS data set) the name of the server. If you add a line to your configuration file such as:

Set machinename kevinspc

This statement will display the variable 'MACHINENAME' in the log with a value of 'kevinspc'.

6) Load Manager

The load manager is not required for socket services but is recommended. It is required for the use of pool services. It is a good idea to run the load manager so that some software intelligence is used to direct requests to an inactive port, rather than being shipped to a port that is already active and thus put into a queue. Another benefit is that if all dispatchers are busy, then the load manager will hold the request until a dispatcher is open. In the case of pool services, it will launch a new dispatcher. Once the load manager (an .exe file) is loaded, there is simply one line within the broker.cfg file which activates it:

LoadManager www.fsddatasvc.com:6301

PROC APPSRV

This procedure invokes the application dispatcher and contains a number of customization options which can be used to your benefit. The procedure can be run within a larger SAS program if you want to create macros and/or set macro variables. At our company, we create a file

called 'appstart.sas' which contains the PROC APPSRV procedure. Here are a few of the options and statements that you might find beneficial.

1) ADMINPW

The Adminpw=xxxxx. Allows you to run admin programs and restricts others from being able to shut down your dispatchers. There are several SAS supplied administrative programs such as PING, STATUS, and STOP (you can create your own as well). If you set up an administrative password, then these administrative programs cannot be run unless the password is provided. An example of how to implement this option is something like this:

```
proc appsrv port=6013
adminpw='MYADMINPASSWORD';
```

2) UNSAFE

The unsafe option strips unwanted characters from fields which end users submit. The most often stripped characters are ampersands, semicolons, and quotes. Use this with caution as you might be surprised where these values are needed.

```
UNSAFE='&"%;'
```

There is an appsrv_unsafe function that allows you to pull the original value even if the unsafe option is utilized.

3) Short macro for simplifying the ALLOCATE, DATALIBS, and PROGLIBS statements.

If your applications dispatcher references a number of libraries, your PROC APPSRV can get quite long and cumbersome. Below is a short macro which can simplify the process.

```
%let drive_direct=f:;
%let drive_share=c:;

%macro libfile(ref,path,server);
  allocate library d&ref "&drive_direct\&path";
  allocate library s&ref "&drive_share\&path" &server;
  allocate file p&ref "&drive_direct\&path";
  datalibs d&ref s&ref p&ref;
  proglibs p&ref;
%mend;

proc appsrv port=6013
adminpw='MYADMINPASSWORD';

%libfile(mylib,\mydir,%str(server=servname.myserver));

%libfile(mylib2,\mydir2,%str(server=servname.myserver)
);
```

4) REQUEST statement

If you wish to run a program either before or after each requested program is run, you can specify the REQUEST statement with either or both of the INIT and TERM arguments. An instance of where this might be useful is if you want to initialize a set of macro variables or run an options statement to apply to all requests. If you are using compiled macros across multiple dispatchers, you need to point the program to them using an options statement to avoid a lock being put on the catalog. To do so, you need to submit an options statement with the 'mstored' and 'sasmstore' options specified. You can then store that options statement in a .sas program (e.g. myinit.sas) and then have it submitted before each request with a statement such as:

```
request init=mylib.myinit.sas;
```

5) STATISTICS Statement

The STATISTICS statement is an extremely useful to for collecting statistics on individual requests made to your dispatchers. By default, the data set is set up to collect the following pieces of information:

Variable Name	Variable Type	Description
Obstype	Character length 1	R = request, I = Internal, U = startup, D = shutdown, T = trace
Okay	Character length 1	1 = request ran okay, 0 = error
Duplex	Character length 1	H = half duplex, F = full duplex
Http	Character length 1	1 = http request, 0= normal broker request
Program	Character length 32	_PROGRAM variable
Peeraddr	Character length 16	Peer address
Hostname	Character length 20	Node name of the server
Username	Character length 12	_USERNAME variable, if any
Entry	Character length 32	_ENTRY variable, if any
Sessionid	Character length 12	_SESSIONID, if any
Service	Character length 12	Service name
Starttime	Number	Time the request started
Runtime	Number	Run time of the request
Port	Number	Server port number

Bytesin	Number	Number of input bytes (read from client)
Bytesout	Number	Number of output bytes (written to client)

As long as the variable is available when the request is made (either by having set it to be exported or creating your own variable), the variable gets added to the data set. There are options that you can set to control the frequency with which the request queue gets written to the data set. You can easily add or drop variables from the data set just as you would any other data set (assuming of course that the application dispatcher is closed as it puts a lock on the data set). You can use SAS/SHARE software to so that multiple dispatchers can write to a single data set. The following statement is a sample usage of the STATISTICS statement:

```
statistics data=mylib.mystats (DLDMGACTION=repair);
```

Note that the data set option is a useful option that we have found prevents manual intervention when the data set is damaged due to an unfortunate shutdown. One of the great utilities of this data set is that you can then write SAS utility programs as needed to analyze the data on a real time basis. For example, you can create reports to show you which programs are running slowly, what the frequency of the various programs is, what the usage by hour and by day of the week is, etc.

6) Using aggregate syntax (member name syntax) with %include

Note that since you are essentially setting up filerefs within your PROC APPSRV statement, you can utilize aggregate syntax for use with %include. This saves you the trouble of having to include the path name. For example, to %include a file named 'myfile.sas' you would simply use the following code:

```
%include mydir(myfile);
```

7) LOG statement

The LOG statement gives you some control over what gets written to your application dispatcher log as well as the naming of the log. If you are not using Web Hound and wish to be able to track happenings via the logs, it is highly suggested that you display as much information as possible. To do so, specify the following:

```
allocate file logfile "d:\mydir\%nrstr(%m%d%y)_1.log";
log display=all symbols=all file=logfile;
```

The 'display=all' option tells the dispatcher to write the log information from all requests to the log file. 'Display=error' would write the log information only when the request ended in an error. 'Display=none' would never write any log data. The same three options (all, error, and none) are available for the symbols argument which controls the listing of the client request passed values within the log. The syntax shown in the example above creates a file with a name of "mmddyy_1.log" where mmddyy is month, day and year. There are numerous other options available for naming the log file.

Running your SAS programs:

Whenever the application dispatcher is summoned, one of the pieces of information that it must receive is something that tells it what to do. This piece of information is in the form of an input field with a reserved name of '_program'. This information can be one of 4 types of programs:

You can run 4 different types of SAS programs. Three of the types reside in catalogs.

1) SAS programs - _program=*library.program*.sas

Pros – easily maintained
Cons – relatively slow as the code needs to be compiled
Generates larger dispatcher log files

2) SCL entries - program=*library.catalog.program*.scl

Pros – precompiled code
Cons - SCL code has its origins in the full-screen enviroment and thus much existing SCL code needs modification to run stand-alone.
Source code is not stored with the compiled macro

3) MACRO entries - compiled macros
_program=*library.catalog.program*.macro

Pros – precompiled code
Cons – Can be difficult to maintain as must compile them and copy them to a catalog
Source code is not stored with the compiled macro.

4) SOURCE entries - .sas programs stuck in a catalog with a different extension
_program=*library.catalog.program*.source

Pros – fairly easy to maintain although not as easy as running SAS programs
Cons - relatively slow as the code needs to be compiled
Generates larger dispatcher log files

CONCLUSION

Determining how you set up your Application Dispatcher within SAS/IntrNet can be a daunting task. As usual with SAS software, there are many ways to get things done. The administrator and programmer are given a wide range of options. A little forethought into what type of statistics you want to gather, how much security you want or need, and how you plan to maintain your SAS/IntrNet web site will go a long way in helping you make decisions that you are comfortable with for the long haul.

Kevin Davidson, Ph.D.
FSD Data Services, Inc.
2020 Southwest Freeway, Suite 206
Houston, Texas 77098
713-942-8436
kevind@fsd.nu
www.fsd.nu

Paper P304

Building a SAS Intranet Site

Tim Williams, PRA International, Charlottesville, VA

ABSTRACT

Intranets are now common in both large and small organizations. These internal websites provide a way to present vast quantities of information in an organized and searchable form. This paper describes the construction and implementation of an intranet site developed by the Analysis Programming staff of PRA International in Charlottesville, VA. Methods for content development and display of SAS coding examples are illustrated. Management of SAS coding projects is discussed in terms of how a database-driven website can increase the efficiency of a programming team and lead to a better deliverable product. It is hoped that this paper will stimulate discussion on how intranets may better serve SAS programmers in the pharmaceutical and other industries.

INTRODUCTION

When I recently joined PRA International, a well-established Contract Research Organization (CRO), I found I had a great deal to learn about the industry. While I was an experienced SAS user, this was my first exposure to the world of pharmaceutical research in the United States. I needed to learn new standard operating procedures and coding conventions. Experienced personnel were often required to assist me in locating information and explaining policies. The situation called out for a well-organized, indexed and searchable resource for staff.

At the time of my employment, PRA was embarking on the implementation their of intranet. Initial development of static HTML pages had begun (Phase 1) with a planned move to dynamically generated content during Phase 2. These early developmental stages provided an excellent opportunity for departments to define their needs, conceptualize site content, and contribute to decisions regarding choices of technology.

This paper describes the initial design of a static HTML intranet site for the Analysis Programming Department at PRA. It also includes a discussion on how processes will be streamlined when a dynamic, database-driven site is implemented. The paper will not discuss issues such as website style, navigation, scripting, or comparisons between technologies and approaches These issues are beyond the scope of the paper and are often dependent upon company policy and individual preferences.

DEVELOPMENT STRATEGY

Winning the support of management and staff was critical to the project. The staff already had heavy workloads, so it was crucial to build a strong case for the departure from client-oriented project work. Managers and staff met early in the project to determine what processes and procedures could be targeted for rapid content development and benefit most from web integration.

Our staffing and workload constraints lead us to adopt the staged delivery model of software development (McConnell, 1996). Staged delivery, also known as incremental implementation, makes content available to users in successive stages as the site is developed. In this way our staff and management saw tangible benefits of the intranet without having to wait for its full implementation. The first content was purely informational; it told the company who the Analysis Programming Department is and what they do. Next, we quickly realized our second goal of providing a resource for department staff. Early successes helped to strengthen support for later, more ambitious aspects of development.

Our overall objective was to start with static content while structuring the site to facilitate the switchover to database-driven dynamic pages (using ASP and Oracle) in the near future. Several staff were already familiar with basic HTML or were able to learn it in a short period of time. The anticipated delay in moving to dynamic content will allow staff to obtain the skills and training they need.

DEVELOPING CONTENT

The PRA Webmaster developed HTML templates, style sheets, and initial site structure for all departments. These templates provided an excellent launch pad for rapid development because they freed individual developers from the time and effort required when starting from scratch. Individual departments selected development teams to identify content requirements and start site construction.

We organized available materials into a series of easy-to-navigate HTML pages to provide immediate benefits to staff. Many of the vital components already existed in electronic form and were easily linked to HTML pages. Such items included:

- Standard operating procedures
- FDA Electronic Submission Guidelines
- Coding conventions
- Project specific information

HTML content was searchable directly after publishing to the web server because we used Microsoft FrontPage 2000 and published to a Microsoft Internet Information Server 4.0. (IIS 4.0) with the FrontPage Server Extensions installed. Additional benefits over the pre-existing documents were gained by including hyperlinks to related site content. For example, links to a Table of Acronyms has proven to be a useful tool for employees new to the industry.

KEY CONTENT

Essential content was identified by meeting with interested department staff and storyboarding the website. We defined the following areas for early development:

- Training in company-specific SAS Coding Methods
- Standard Operating Procedures
- Table of Acronyms
- SAS Content
    - SAS help documentation
    - External SAS resources

- Search facility for SAS discussions (see below)
- SAS functions reference
- Standardized "in-house" SAS macros and code modules
- Code examples contributed by staff

SEARCHING THE SAS-L ARCHIVES AND COMP.SOFT-SYS.SAS

One of the most popular items on our website is the ability to conveniently search both the UGA SAS-L Listserv and **comp.soft-sys.sas** through the website through the Google search facility at http://groups.google.com. Google Inc. acquired the Deja.com Usenet Discussion Service in February 2001. The service provides a means to search millions of Usenet messages dating back to 1995.

Instead of providing simple links to the internet home pages of these searchable archives, we placed the search forms within our intranet, thus removing the additional step of loading the search page from the remote server. Our staff simply enters the search text along with other optional parameters and the form is then submitted to the remote search engine. The browser returns the search results page when processing is complete.

The code below illustrates how simple it can be to incorporate such functionality into a web page. In this example, **comp.soft-sys.sas** is searched for articles containing a list of all the words entered on a form:

```
<form method=GET
   action= "http://groups.google.com/groups"
   name=f>

  <h3>Search comp.soft-sys.sas</h3>
  for all of these words:
  <input type=text value="" name=as_q
    size=25>
  <br>
  <input size=30 type=hidden
     value="comp.soft-sys.sas" name=as_ugroup>
  <input type=submit value="Google Search">

</form>
```

At the time of this writing, groups.google.com supports the following fields on their "Advanced Groups Search" form:

- All words in a list (as in the example above)
- Any of the words in a list
- An exact phrase
- Subject
- Author
- Message ID
- Language

Further options are also available to limit the search to a range of dates, return only a specific number of items, and sort the results by either relevance or date.

Developers can easily add these fields to their own form by locating the field names on the "Advanced Groups Search" page at http://groups.google.com. Simply use the View | Source menu on Internet Explorer when the remote page is loaded. A similar method was used to implement a form to search the UGA SAS-L Listserv from our intranet.

PRESENTING SAS CODE ON THE INTRANET

Formatting of the original code lines should be preserved when presenting SAS code examples on the web. This allows users to cut-and-paste code from the web into a SAS code editor. Similarly, contributors to the website need to cut-and-paste SAS code into FrontPage (or another HTML editor) while preserving formatting. Users of the website should be able to quickly identify SAS code through the use of distinct fonts and colors. All of these elements are available through the use of HTML tags and style sheets.

<PRE> TAG

The HTML tag <PRE> defines a section of text that the browser will render in exactly the same character and line spacing as it appears in the HTML source document. It is this tag that allows the programmer to cut-and-paste SAS code from the SAS editor (or other text editor) directly into HTML while preserving spacing and indentation of the code lines. Some authors use this tag in conjunction with the <BLOCKQUOTE> tag that helps to delimit the code section by applying special indentation and formatting. Our approach was to instead use our own user-defined tag <CODE> to encapsulate SAS code blocks and render formatting through application of styles to that tag.

STYLE SHEETS AND THE <CODE> TAG

Cascading Style Sheets (CSS) provide a powerful and easy way to provide a consistent font, font size, color and other properties for HTML pages, sections of pages, or an entire website. The web author can not only control the presentation attributes of standard HTML tags but can also implement user-defined tags for their own use. Styles can be applied to HTML tags using inline styles, document-level styles, and external style sheets. Use of external style sheets is often preferred because it allows the web author to apply global style changes to a website by editing only one file. Styles "cascade" down from the external sheets through local document styles down to inline styles.

We developed a style sheet for SAS program code to enhance the style sheet supplied by the PRA Webmaster. We chose to distinguish the program logs, outputs, comments, and required program parameters from surrounding text on the web page. We applied additional styles to code linked to documentation through either a mouse-over or a mouse-click. Our new style sheet was linked to all documents containing SAS code using the following line in the header section of HTML pages:

```
<link rel="stylesheet type="text/css"
href="../SASCode.css">
```

Styles were then applied to the user-defined <CODE> tag by specifying different values for the CLASS parameter as shown in the table below.

Classes of the user-defined <CODE> tag

CLASS=	Used for display of:
comment	Comments within code, typically code lines with /* */ , * or %* comment styles
change	Program code that should be changed by the user prior to implementation. E.g.: a random number seed.
explain	Text explaining a code segment
footnote	Designates footnotes that are not part of the code but are used to label code for explanation in a footnote

log	SAS log
out	SAS output
prog	SAS program code

The following code illustrates the use of the <PRE> and <CODE> tags :

```
<pre>
<code class="prog">
data random1(drop=i);
  do i = 1 to 10;
    random=ranuni(
       <code class="change">54321</code>);
    output;
  end;
run;
</code>
</pre>
```

In addition to uniquely rendering the types of SAS code being presented, it is also useful to identify sections of code that contain explanations or documentation available through mouse-overs or hyperlinks within the code itself. Styles were therefore defined for use with the HTML <A> tag to enable users to identify these areas on the screen. Typically, these <A> tags will appear enveloped within <CODE> tags for the presentation of SAS code and output.

Classes of the HTML tag <A>

CLASS=	Used for display of:
example	Code is hyperlinked to additional examples illustrating the same concept or program statements.
explain	Code displayed with this parameter is hyperlinked to an explanation or documentation
footnote	Code is hyperlinked to additional information in a footnote.
mouseExplain	An explanation of the code will appear on the page when the mouse pointer is moved over the code section.

The code below illustrates the use of the <A> tag to display the `ranuni` function as an HTML hyperlink which, when clicked, takes the user to the HTML document sasFunctions.html that contains additional documentation about the function.

```
<pre>
<code class="prog">
%let rannum=<a
href="./sasFunctions.htm#ranuni"
class="explain">ranuni</a>(1234);
</code>
</pre>
```

USING SAS TO GENERATE AND UPDATE HTML PAGES

Many papers have been presented in past conferences describing how to generate HTML output from SAS prior to the introduction of ODS in Version 8 (Bahler 1999, 1998 ; Pope 1997). SAS can also be used to update existing HTML files that are subsequently published to the intranet. One way to accomplish this is to develop a template page containing HTML comment tags that delimit a section where a SAS program will insert output. No text between the start of the HTML comment tag "`<!--`" and the end of the tag "`-->`" is displayed by a browser, so the comment tag provides a convenient and invisible marker for use by a SAS program.

[Top of the HTML document omitted]

```
<h2>SAS Generated Table Follows</h2>
<!-- Start SAS Insertion -->
  <!-- this text will be replaced by SAS -->
<!-- End SAS Insertion -->
<em>Table generated using SAS 6.12</em>
```

[Bottom of the HTML document omitted]

It is then a simple task to develop a SAS program that reads the HTML page one line at a time and writes it back out until the insertion point is found. When the program finds the insertion point a routine is called that inserts SAS output into the HTML document.

Once the SAS program has finished inserting HTML, it will read in (but not write out) lines from the source HTML file until it reads in a line denoting the end of the insertion area:

```
<!-- End SAS Insertion -->
```

Using this approach, any text that exists in the HTML template between the start and end of the insertion point is not written to the HTML output file. When the end of the insertion point is found, the SAS program reads in and writes out each line until the end of the input file is reached.

The completed HTML output file must then be published to the web server before it can be viewed. Although inconvenient, this method has provided a way to extract data from Oracle databases for display on our intranet, using SAS as an intermediate step.

THE CHALLENGES OF MICROSOFT FRONTPAGE

Microsoft has made many improvements over early versions of FrontPage but it can still be problematic for the HTML source code enthusiast. Care must be taken when developing the "template" pages mentioned above; opening them into FrontPage may result in pages with lines longer than the 200 character limit of SAS version 6.12 and earlier. The FrontPage visual interface also has an undesirable habit of inserting many proprietary tags and redundant formatting tags that make editing of the source code difficult.

However, if a developer employs patience and ingenuity, FrontPage can make the presentation of SAS code easier (and thereby encourage other programmers to submit code examples to the website). One such example is the ability to easily apply the user-defined tags for formatting SAS code. Ideally, a developer would select code with a mouse in FrontPage's visual editor, then click on a button to apply the custom style to the selection.

This functionality can be added to FrontPage by defining macros in the Visual Basic Editor and assigning these macros to buttons

on a toolbar. An example of a Visual Basic macro that applies the <CODE CLASS="prog"> tag reads as follows:

```
Sub CODEProg()
  ' Apply the <CODE CLASS="PROG"> tag
  ' to multiple selected lines
  Dim objTxtRange As IHTMLTxtRange
  Dim myHTML As String
  'Range of selected text
  Set objTxtRange =
  ActiveDocument.selection.createRange
  myHTML = "<code class=" & """prog""" & ">"
  & objTxtRange.htmlText & "</code>"
  objTxtRange.pasteHTML (myHTML)
End Sub
```

FUTURE DIRECTIONS

An intranet will truly come into its own when it uses dynamic content. The next step in our development process is to move toward dynamic content generation using Active Server Pages (ASP) and a database back-end.

In our department we have well-defined quality assurance methods for code development, team coordination, and quality control. These three elements currently exist as separate entities that could be joined together using web technology.

Our SAS programs contain a standardized header section that includes information such as :

- Author
- Program abstract
- Date of code completion
- Date of code validation
- Author of code validation
- List of input and output files
- List of macros called
- Notes
- Amendments

Headers serve not only as a guide to our programmers but also to our clients who often request the source code as part of the deliverable product. Much of the program header information is repeated in a Microsoft Excel "Table of Programs" (TOP) spreadsheet. Our programmers use the TOP to coordinate code development while providing additional documentation for each SAS program.

After programming is complete, each program is validated using a rigorous Quality Control Checklist. As a series of MicroSoft Word documents (one for each SAS program) the QC Checklists represent a third element that disconnects and adds redundancy to the process (FIGURE 1).

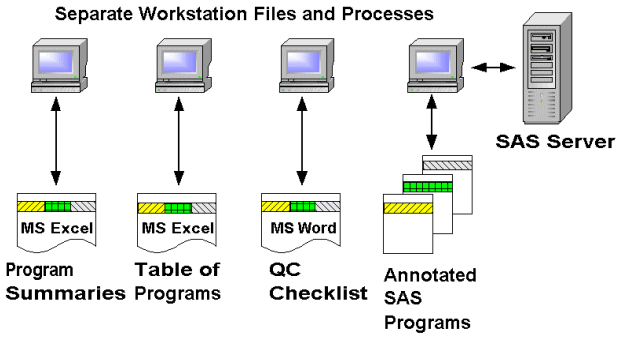


Figure 1 : SAS programs and related files prior to database integration.

Related elements of the coding process can be integrated into a web-based system (FIGURE 2). A centralized database will hold information about each program for display and updating through dynamically generated web pages. These pages will allow developers to enter information about a project and its associated files. QC documents will no longer exist as disconnected elements, but will share fields with the Table of Programs and SAS Headers in a database. Redundancy of data entry will be greatly reduced and the entire process will become streamlined due to the centralization of information. Programmers will be able to spend more hours on client oriented project work and less time laboriously filling out redundant fields during code validation and project management.

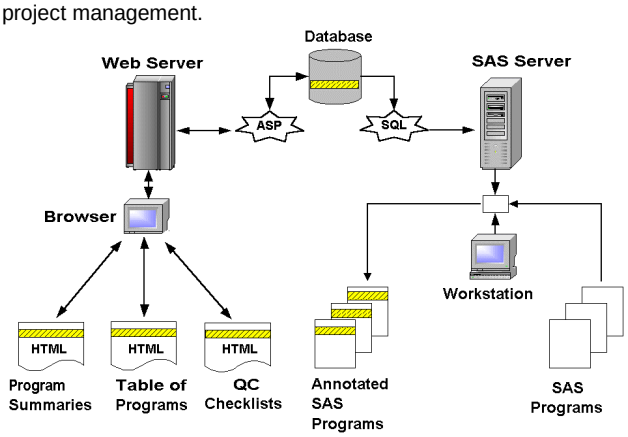


Figure 2 : Integrated database approach to managing SAS programs and related files

The approach to the SAS program headers requires further investigation. One option may be to have a reduced header during code development that contains only a few key fields needed to link it to the database. One could foresee the placement of a hyperlink that, when clicked in the editor, would launch the database interface providing a means to view, add and edit information about the program. Once coding is complete, a separate "annotator" SAS program could link to the database and extract information about each SAS program in a project. The annotator program would sequentially read in each SAS program and insert detailed information from the database into the header comment section. Each program, with its updated, detailed header would be written back out to a production directory for shipment to the client and archiving.

CONCLUSION

Implementation of an intranet can greatly improve existing processes and add new ones that increase department efficiency. The Analysis Programming site has become a model for other intranet sites in our organization and has facilitated communication between departments and regional offices. However, it remains a challenge for staff to find the time to supply content, even with several tools available to facilitate their contributions. Use of our intranet will increase dramatically once our standard operating procedures become incorporated into a dynamic, database-driven website that integrates coding, quality control, and project management.

REFERENCES

Bahler, C et al. 1998. SAS and HTML - HTML Publishing Using SAS. Proceedings of the Twenty-Third Annual SAS Users Group International Conference. Paper 38.

Bahler, C, S. Calhoun, E. Kistner. 1999. Essentials for Static and Dynamic Web Publishing - SAS HTML Formatting. Proceedings of the Twenty-Fourth Annual SAS Users Group International Conference. Paper 158.

Jennett, M. 1999. FrontPage 2000 Developer's Guide. Osborne/McGraw Hill. 709 p.

Kauffmann, J , K Spencer and T Willis. 2000. Beginning ASP Databases. Wrox Press. 824 p.

McConnell, S. 1996. Rapid Development: Taming Wild Software Schedules. Microsoft Press. 647 p.

Musciano, C and B Kennedy. 1998. HTML: The Definitive Guide, Second Edition. O'Reilly & Associates Inc. 531 p.

Pope, P. 1997. Using the SAS System for Large Volume HTML Document Production. Proceedings of the Twenty-Second Annual SAS Users Group International Conference. Paper 215.

ACKNOWLEDGMENTS

I wish to thank the Analysis Programming staff at PRA Intl. for their continuing feedback and assistance. Analysis Programmers Larry Broach, Jennifer Potter, Dev Patel and Senior Manager Lee Walke were instrumental in developing site content and layout. Special thanks to Leslie Cagley, PRA Intranet Webmaster, for assisting with site setup and always having an answer for my countless questions.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Tim Williams
PRA International
4105 Lewis and Clarke Drive
Charlottesville, VA 22902
Work Phone: (804) 951-3504
Fax: (804) 951-3001
Email: WilliamsTim@PRAIntl.com
Web: www.praintl.com

Paper P305

Sounds Like a Good Idea, But What's the ROI?

John E. Bentley, First Union National Bank

Abstract:

Too many good ideas for using SAS® Software remain just that: good ideas that are never implemented. Why? Getting an idea approved and funded as a project largely depends on showing that it has real value, and most organizations select projects based on ROI—Return on Investment. ROI is the net profit that a project generates in the form of either additional revenue or reduced expenses. This paper will introduce SAS users to concepts related ROI, present some essential information about calculating ROI, and suggest how to write a Project Justification Document that includes an ROI estimate. The goal is to provide some insight and suggestions on how to get a project approved.

Disclaimer: The views and opinions expressed here are those of the author and not those of First Union National Bank. First Union National Bank does not necessarily subscribe to any philosophy, school of thought, definition, methodology, approach, or process the author describes.

The Gravy Train is Over

The days of easy money are over, if you were lucky enough to enjoy them. The business cycle has turned and the bottom line focus has returned. For everyone from Corporate CEO's to programmers, controlling costs and increasing revenue have re-assumed their preeminent place in the corporate strategic plan.

This focus on profitability will continue for quite a while. Several quarters ago it became clear that the US economy was slowing after two incredible years of Internet-fueled growth. Companies generally over-invested in technology during those years, seeking revenue generation at the expense of profitability. As marginal returns on technology spending fell, companies slowed their investment in IT and, consequently, the demand for equipment declined and equipment supplier were hurt.

As companies retrench they cut back in all areas, including advertising, travel, and training, spreading the pain to non-IT companies. Hiring slows and layoffs are announced. As this paper was being written in Spring 2001, many economists and business leaders were forecasting an economic turn-around in the second half of the year, but others said that these conditions would exist into 2002.

In this economic climate, savvy business leaders realize that the continued investment in strategic technology can be the single biggest differentiator between a thriving business and one that's just treading water or even going under. Smart CEOs are positioning their companies for the next upswing in the business cycle. Even so, it's no surprise that as part of their cost-cutting efforts organizations are examining IT projects more closely and trying to separate those with the greatest potential for improving the bottom line from those with little or no potential.

There is much evidence showing that IT spending has been virtually out-of-control. A recent case study by The Beta Group focuses on a large financial services firm in which 60 percent of the IT systems development and support budget was being spent without an evaluation of the business impact of the expenditure. An analysis revealed that almost $30 million was spent annually under an "entitlement budget" for legacy system support. The Beta Group estimates that most companies can redeploy 10 to 20 percent of their IT budget by evaluating the business impact of those expenditures and reallocating dollars to higher-return projects and initiatives.

A survey of corporate IT executives by ComputerWorld in Spring 2001 found a new focus on deriving real business value from IT spending. In the past, the value derived from e-business technology, for example, was secondary to having an e-business initiative. Projects with Internet, web, OLAP, or BI in its name were almost guaranteed funding. A John Deere e-business manager, though, typifies the new approach to IT project review: "Don't loose sight of the fact that IT is a tool in the context of the larger business plan."

Project Justification and ROI

Here's a hypothetical chance conversation in an elevator between a Division CIO and a SAS application developer:

CIO: Bret, I've just come from a meeting where your manager said that you have an idea about making our inventory reports available over the corporate Intranet. It sounds like it will save money.

Bret: Yes, it is a great idea… we'd be using cutting edge technology and I know everyone on my team would like to do it.

CIO: Cutting-edge technology? Hmmm. Well, it still sounded like a good idea. I've asked that you send in a project justification that includes an ROI estimate.

Bret: Huh?

In recent years, SAS programmers and developers have largely been insulated from concepts related to profitability. For many IT professionals who have fewer than five or six years' experience, this renewed focus on business value might be a new theme. For the more experienced, it may be only a vague memory. For everyone, well, get used to it. During tight economic times all IT projects will get closer scrutiny to see if they improve the bottom line. "Improving the bottom line" may mean different things in different organizations, but it always includes justifying a project in terms of having a positive ROI, and the sooner the better.

An ROI estimate is an indicator of profitability showing by how much—in either dollars or as a percent—new revenues and/or operational cost reductions exceed the investment. ROI is derived from a comparison of project costs to project benefits. Basically, it shows the profit that can be attributed to your project after it is implemented and after the project costs have been deducted. Quite often a technique called payback analysis is also incorporated; in this, the length of time it takes to recover the initial cost of a project is factored in. Given the pace of technological change, though, budget managers are looking for shorter and shorter payback periods.

There are four generally accepted techniques for estimating IT project ROI, and spreadsheet packages will do all the calculations. By using the same set of techniques across projects for estimating potential profitability, management can accurately compare projects and select the ones with the highest value-added.

A criticism of all four techniques is that while they compare benefits to costs focus on current expenses and minimize

maintenance costs. Also, they do nothing to insure that the technology selected and related expenditures are appropriate for the project.

Table 1. Measuring IT project profitability

Technique	Description
Payback analysis	The length of time it takes to recover the initial cost of a project, with or without regard to the time value of money.
Return on investment	A comparison of profitability indicators, where net new revenues or operational cost reductions from the project exceed the investment.
Net Present Value	The present value of the expected future cash flow generated by the project minus the cost. Useful for comparing projects.
Internal rate of return	The discount (interest) rate at which net present value is zero. Usually an organization will set a benchmark rate such as 25 percent so that only the highest value-added projects are funded.
Economic value added	Based on the concept that profitability should account for the cost of equity capital as well as debt and other means of capital acquisition. Companies invest only in projects that will return more than the total cost of capital.

There are three generally accepted models that utilize the ROI techniques presented in Table 1. These models help organizations assess and prioritize technology investments.

Table 2. Models Guiding IT Project Selection

Model	Description
Enterprise ROI	First identify enterprise-level financial targets for new revenue or cost reduction. Then allocate those targets to broad initiatives, which set targets for specific projects. The success of any given project, however, is often dependent on a concurrent project.
Productivity	The project is justified on the basis of employees "doing more with less." Uses goals such as "managers will use 10% less time searching for information." Result is difficulty in auditing and measuring actual benefits. Often difficult to accurately attribute financial gains to a specific project.
IT Value/Utilization	Focuses on individual project economics and maximizing value of IT investments by managing execution and implementation. Premise is that the technology aligns with the existing IT strategy and has low-cost maintenance.

To improve profits, your project must either (a) generate higher revenues or (b) reduce costs, but preferably both. The project can be something completely new, like delivering customer profiles on-demand to a PDA, or an enhancement to an existing system, such as automating the validation process for an existing production report. As long as it adds real measurable value, it's a candidate for approval.

ROI elements

Many items can be used in an ROI estimate but both costs and benefit estimates are needed. You need to take a snapshot of how you're doing business now and then compare that to an estimate of how you'll be doing business when your project is in production. It's nearly impossible to present an ROI estimate without using a baseline to estimate the cost-benefit differential.

One important point to remember*: Costs are incurred up-front during project design, development, implementation, and maintenance ~~a~~all through the life cycle. Benefits don't accrue until the project goes into production*.

Costs

The first two parts of a four part series by William Roetzheim in Software Development Magazine provide valuable information on estimating project costs. Costs can be grouped into three broad areas.

- Time
- People and skill sets
- Hardware and software

Time is obvious. How many people do you need and how many person hours will it take? Multiply people by their hourly rate (or an average). Don't use a best-case scenario, but don't use a worst-case scenario either. Take a hard look at the project and make a good, solid, realistic estimate.

People and skill sets. Do you have the skills the project needs? If not, you'll have to acquire them by either training existing staff or hiring contractors. If you have to train or bring in outside help there will initially be some non-productive time, so be sure to adjust your time estimate to allow for people coming up-to-speed.

Your people-related costs should be adjusted to account for certain "environment factors" that are beyond the scope of this paper but are detailed in the Roetzheim series. Keep in mind that you need the people and skills not only to develop and implement the project but also to maintain and upgrade it over its life cycle.

Hardware and software costs are also obvious. Either you have what you need or you don't. But make sure you know what you need and what you have before submitting your ROI estimate. Finding out a month into the project that you need $10,000 for DASD will blow away your credibility and may sink the project. Even if you don't need extra DASD this year, what about next year? Forecast your needs three years out.

Benefits

Benefits should be calculated after the costs. It might quickly become clear that the costs are prohibitive. There are four categories of benefits derived from IT:

- Time Savings
- Personnel Savings
- Operational Savings
- Revenue enhancement

As a simplified example, lets say that we have an idea to automate a set of ten weekly reports that require fifteen person hours to generate, validate, copy, and distribute through interoffice mail to thirty-five people. Because the process has been stable for a number of months, we want to automate everything: the report generation and validation processes, change the output from listings to spreadsheets, and then email the spreadsheet to the recipients. If there are no validation problems (and there haven't been), the

entire process will happen Sunday afternoon and the fifteen person hours are reduced to zero.

Time savings are the amount of time saved by the user community. It is a sort of "time to market" concept and can be difficult to quantify. In our example, the users are the people receiving the reports. We will have to talk with them to get a feel for the value (if any) that they will derive from having the reports in spreadsheet format at 8 am Monday instead of Tuesday when the interoffice mail is delivered.

Personnel savings include improvements in staff utilization such as allowing a staff member to redeployed from one area to another, reducing the number of temporary hires, or slowing the rate of new hiring. Deferred training costs should be included here. Average annual salaries can be used in this estimate. In our example, the person who modifies and submits the program each week and the person who validates the results can be reassigned to other work. Here we save 15 person hours, so we multiply 15 hours by the average hourly salary. But we should reduce that number by an efficiency factor of 30% because not all time saved will be redirected productively.

Operational savings can be unexpectedly high as you "speed up" a system to run a job faster or "scale up" to handle more work in the same amount of time. To estimate savings in this area, the Systems Administrator can provide an estimate of how much it costs to run the system on an hourly basis. In our example, the project may require a substantial rewrite of the existing report generation code and that may allow us to make the code more efficient. Time saved here, though, will have to be subtracted from the time it takes the new validation routine to run.

Operational savings also include deferred software and hardware costs. It may be that one of our motivating factors for our project is that we heard a rumor that the IT group was considering purchasing an OLAP software package to do what our project does. Since we already have SAS Software, we don't have to purchase the new package, install and implement it, or train users. We also save the paper that the hardcopies are printed on. Finally, if the reports deal with inventory, there may be savings attached to being able to place a restock order Monday morning instead of Tuesday afternoon.

Revenue enhancement from an information analysis or delivery project can be difficult measure. If a manager uses your information along with other information (and his own experience) to make a decision that generates $250,000 more revenue, it's difficult to assign a percentage to any one piece of information that he used. But if your project will result in faster turn-around or higher response for a marketing campaign, it's easy to assign a dollar value to that "lift".

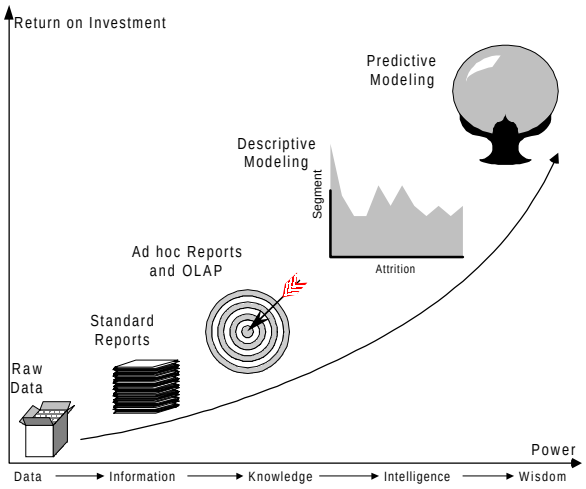
Which Projects Will Be Approved?

The surest way to getting a project approved is to learn your organizations priorities and strategies and then propose a project that supports them. A project that delivers well-defined business results that support strategic goals should always make the first cut in a budget review. Finding a partner on the business side to help you develop, focus, and refine your initial concept will also improve your chances. Decision-makers are always more likely to approve projects that are jointly developed by business and IT staff. Further, projects that already have business support will get priority.

IT projects related to data and information differ in their business value. Capturing raw data is good, but in and of itself doesn't add value; analyzing and reporting adds value to data. Discovering patterns, running complex statistical analyses, and model building adds even more value.

As a joint SAS Institute/EMC White Paper describes it, there is a continuum along which data moves to first become information, then knowledge, then intelligence, and finally wisdom. It's at the higher levels that companies gain the highest ROI in information gathering, processing, reporting, and analysis. Figure 1 depicts how moving from having raw data to performing predictive analysis increases the power of information and, consequently, ROI.

Figure 1. Information Power = ROI



Information's value also increases as more people have access to it, and delivering the right information at the right time to the right people is extremely valuable. Applications that deliver information are high value-added projects, and the Web provides a perfect channel to expand information access to front-line supervisors and customer-facing staff. Applications that automate processes and tie together data analysis and delivery are also high value-added.

The Project Justification Document

A Project Justification Document is not a formal IT Proposal. IT Proposals are usually required only for major system implementations or modifications. It provides management with a comprehensive evaluation of a significant problem, suggests options, presents the preferred solution, and asks for a go-ahead. An IT Proposal is much more time and research intensive than a Project Justification and is usually a formal group project that includes team members outside the IT group. Our focus here is on a "quick and dirty" Project Justification Document that may be a precursor to an IT Proposal.

The Project Justification Document is exactly what it says: it justifies the project and says on why it is important. Let's define the terms.

Project:	An extensive undertaking requiring an extended, focused, concerted effort.
Justification:	A condition that demonstrates a concept or action to be right, valid, or just.
Document:	A written or printed paper that contains information presenting decisive evidence

Based on our definition of terms, you need to identify a basic message, the message contents, and a writing style. Your language and style should be should short, clear, and concise. More is not better. Remember that non-technical business and budget managers may be reading it.

- Be direct and unequivocal (don't say "in first quarter 2002, say "in March 2002",");
- Quantify when appropriate and possible (don't say " several dozen users", say "36 known users");
- Be specific (don't say "this project is important", say "the outcome of this project will enable <someone> to achieve their goal of <goal>.

The document should be only two or three pages long. Each section should have a short heading and be no more than two or three paragraphs and a table. Where possible, use bullet points instead of a paragraph. Here are some buzzwords that should be relevant to a software project.

1. Effectiveness – the ability to work smarter and do more with less.
2. Productivity improvements wherein fewer people do the job in a shorter time period.
3. Ease of use, requiring minimal training and having a steep learning curve.
4. Reuse of hardware and software that the organization already owns.
5. Scalability and the ability to accommodate growing volumes of data, demands for speed, and increasing numbers of users.
6. A neutral architecture that conforms to industry standards and supports multiple platforms.
7. Data exchange and integration with other applications.
8. Reuse and sharing of the project's components and modules within the development organization and for future projects.

Writing a Project Justification Document

Know Your Audience

The purpose of the justification is to persuade decision-makers that they should approve and fund your proposal by allocating time and money in the budget. But who are these decision-makers? It's entirely possible that your own supervisor or manager is not one of them. Instead, they may your organizations senior managers—your boss's boss—financial group managers, and budget analysts. Very likely, people who know and understand your technology are not among them, so you have to be very clear and specific in your writing.

What do they want to know? They want to know the same things you want to know when you are trying to decide whether or not to purchase a product or service for your own use:

- What specifically is it?
- What specifically will it do?
- How much does it cost?
- How much does it cost to maintain it?
- When will I get it?
- Can I trust the seller?

Write an Outline

Writing an outline is similar to flowcharting: everyone usually agrees that it's valuable, but it's usually not done. In this case, it helps organize your points, identify which information to include, and avoid things that obscure your message. Major sections of the outline contain:

- Alignment with a business objective or a strategic goal. ("Here is our goal, and this will help us achieve that goal.")
- Cost estimates. ("This is what it will cost you to do it and what it will cost to maintain it.")
- The approach. ("It will do what it will do.")
- Key deliverables and a schedule. ("This is what you will get and when you will get them.")
- A project control plan. ("This is how you know you are getting the work done on time and within budget.")

Flesh out the Outline

Align with a business objective or a strategic goal. Find out what objectives and goals are hot buttons with the decision-makers and think of ways that your idea will support those. Generally, this means doing things better, cheaper, or faster than they are now. You need to increase revenue or decrease costs. Start with a "Project Objective Statement":

> "To [implement, complete, activate, etc.] [what] by [month, year], which will enable [business group, function, or process] to [increase, decrease, etc.] [what - related to business objective].

Cost Estimates. The cost estimates here are neither exceedingly detailed nor agonizingly accurate. Although accuracy definitely counts, senior managers and financial analysts are comfortable with estimates and realize that you're projecting costs up to six months in advance. Always give a good faith estimate. Depending on how much you're asking for, you should probably round your estimates to thousands or hundreds of dollars. At a minimum, the cost estimate must include the total amount requested, broken down into personnel expenses and non-personnel expenses for a specific time period, usually the coming fiscal year.

Personnel expenses include the salary cost of employees who will be working on the project only the period of time that the employees will be assigned to the project. That is, if a person is working full-time on the project for three months, only one-quarter the annual salary is requested for the project. Also include cost estimates for contractors who will be needed for the project. (Obviously, by the time you get to this point you will need to know the skill sets that your project will take and the number of man-hours it will require.)

The amount for non-personnel expenses must include all other expenses associated with the project. This includes hardware and software acquisition and licensing, training, travel to the training, reference manuals, the cost of setting up work cubicles, long-distance phone calls, even for late-night dinners. Think of everything. Don't knowingly underestimate here and blind-side your managers later with a request for something you should have foreseen, like $2,500 to send a developer to Java training when your training budget was $49.95 for "Learn Java in 15 Days".

It's important that your cost estimates not only include costs to get the project completed, but annual estimates of what it will cost to keep it in production over its life-cycle. Is there an annual license renewal associated with the software, or as the database grows will you have to purchase more disk space? If so, get those costs out in the open now. It helps establish your credibility, and by showing that you've done the research you'll somewhat reduce the likelihood that anyone will question your numbers.

The Approach. Briefly explain how you identified the need for the project and the steps you'll take in the design, development, testing, and implementation phases. You should include a paragraph about how you've worked closely with a business partner in developing this solution. (How else would you know that this is the right solution, right?)

It's important to include a few sentences on any significant risks and how they will be prevented or mitigated. Be up-front about the risks, but don't dwell on them. If, for

example, you need a UNIX server with a specific configuration and you've found an underused box that fits your specs and your cost estimates assume that you'll get access it, you must say what the consequences will be if you don't get it—either you'll be delayed while you find another box or the cost goes up when you buy a box or the project is stopped. If you need a team member with a specific skill set like Java, what will be the consequences if that person isn't available when you're ready?

A description of how you will assure that the project will deliver results is also necessary. This could include a concise overview of the software development and/or project management methodologies that will be used, the extent of end user involvement in development and testing, project progress reporting, and post-implementation follow-up and evaluation. This section should not be exhaustive treatment of any of the items mentioned. The intent is to help decision-makers understand that you have thought through the project and understand what needs to be done to deliver what you are promising.

Key Deliverables and a Delivery Schedule. This is where you spell-out what they are getting for their money and when they will get it. Deliverables are tangible products that the project will generate; there are intermediate deliverables (e.g., a beta version of a software) and final deliverables (e.g., fully tested and documented software placed in production). Always include the final deliverable and delivery date. Don't over promise! A delivery date appears "overly aggressive" will make people question your credibility and ask more questions.

Depending on the project, intermediate deliverables could include:

- Business and System requirements documents
- Hardware acquisition specifications and installation
- Pilot project results
- Software installation
- Database design and implementation
- Evaluation and acceptance testing
- User documentation
- User training

The Project Control Plan. Briefly explain here how you will keep the project activities on track and insure that it will be finished on schedule. This section should include a few sentences each covering:

- Project plans and schedules. (Avoid details.)
- A short table of key milestones and dates.
- Key performance indicators for monitoring and tracking progress.
- The process for taking corrective actions when needed.
- What periodic meetings you will have, who will attend them, and when, how, and to whom you will communicate project status.

Double check the numbers and get a peer review

Budget numbers are closely checked so don't let a simple mistake in arithmetic undermine your proposal. Even though many of the people reviewing your proposal may not be IT professionals, they'll know enough to know when they're being sandbagged. Don't use clearly unrealistic estimates because credibility is the very first criteria a proposal must pass.

Get a peer review from someone who isn't familiar with the business issue you're address or the technology you want to use. After they read it, ask them to tell you what the project is, what it is for, what the business benefits are, and why it's important. If your reviewer can't answer these questions, the people reviewing your proposal won't be able to either.

Summary

An idea needs to be funded before it becomes a project, and funding depends on the anticipated Return on Investment (ROI). Calculating ROI is not necessarily difficult, but it's critical that the Project Justification Document presenting the ROI be clear, concise, and as accurate as possible.

Although the Project Justification Document may be only the first step in the approval process, it's a critical first step. For that reason, it's important to keep in mind that non-IT staff will very likely review your proposal and that the Document has to clearly describe the project, what it will accomplish, and how much it will cost in addition to how much it will add to the bottom line.

References and Resources

Hummingbird. "Enterprise IT Value: Beyond Data Warehousing and ROI."
www.humingbird.com

Information Discovery, Inc. "Measuring the Dollar Value of Mined Information".
www.datamine.aa.psiweb.com/infoval2.htm

Resources Management Systems, Inc., on-line tutorial. "Get Your IT Project Funded – 5 Steps to Improve the Odds".
www.rms.net/tut_proj.htm

Roetzheim, William H. "Estimating Software Costs."
Software Development Magazine, 4-part series, October 2000 to January 2001.
www.sdmagazine.com/articles/2000/0010/0010d/0010d.htm

SAS Institute and EMC, Inc. "SAS and EMC: Working Together to Turn Data into Knowledge." White Paper.

The Beta Group. "IT Budgets in Uncertain Times, Part 1: Managing The Legacy "Entitlement Programs". Working Note, 2001.

Contact Information

John E. Bentley
First Union National Bank
201 S. College Street
Mailcode NC-1025
Charlotte NC 28288

704-383-2686
John.Bentley2@FirstUnion.Com

About the Author

John Bentley has used SAS Software for fourteen years in the healthcare, insurance, and banking industries. For the past four years he has been with the Enterprise Information Group of First Union National Bank with responsibilities of supporting users of First Union's data warehouse and data marts and managing the development of SAS client-server applications to extract, manipulate, and present information from it. John regularly presents at national, regional, and special interest SAS User Group Conferences and local SAS User Group meetings.

Web-Intelligence: A Primer

Don Henderson, PricewaterhouseCoopers LLP
Ralph Mittl, PricewaterhouseCoopers LLP

Abstract

Web-Intelligence (also referred to as e-Intelligence) is the application of business intelligence and analytics to data resulting from activity at a web site. Given the advent of e-commerce, such data is becoming very important to organizations as they strive to:

- Understand who their customers and prospective customers are
- Provide appropriate content to their web site customers
- Leverage other information they have about customers and prospects from other channels

Much attention is given to concepts like web-site personalization engines, click-stream analysis, predictive modeling, web-site optimization (the list goes on).

In order to fully leverage this data, it is essential to have an understanding of what data are available and the types of questions that data can help answer. The next step is then to use a proper data warehouse platform to support the access, management and analysis of that data.

This paper will provide an overview of the key concepts involved in making effective use of web log data.

Basic Definitions

There are a number of metrics and terms that are discussed in the web-intelligence space. In order to leverage the data available from a web site, it is essential to understand exactly what the metrics are, and their corresponding limitations.

One of the most commonly discussed metrics is ***Hits***. When a user visits a web site, each request is logged, and each of those requests is counted as a hit. The Hits metric is of interest to those who manage web sites and are interested in load, demand, etc. However, Hits is a metric whose value can be misleading. For example, if a single web page has 15 graphics, then whenever a user makes a request to see that page, there are 16 hits. One hit is counted for the main web page, and one for each of the 15 graphics.

The ***Page View*** is an attempt to deal with this shortcoming. A page view counts the number of requests to load a single web page, regardless of the content on the page. This metric is often measured by excluding from the counting any graphic or multimedia files. However, there are a number of other confounding factors that impact page views. For example, consider an HTML page that is a frameset, with two embedded frames (or panels). The question is whether to measure **Page Views** for the individual panels (which are also HTML pages) or the frameset definition page. Thus, this could be considered a single **Page View** for the frameset or single Page Views for each frame (or pane).

The number of ***Visits*** or ***Visitors*** is a measure of how many times any user has viewed any page on a web site within some prescribed time period (a **Session**). Note that if a user leaves the site and returns after the prescribed time period ends, this is interpreted as two visits.

Summarizing these, consider the scenario where a single person visits a site and views 7 pages (no frames to keep the

example simple), each of which has 5 embedded graphics. The values for these metrics would be:

- ***Hits:*** 42
- ***Page Views:*** 7
- ***Visitors:*** 1

Next, let us consider the concept of a ***Unique Visitor***. A unique visitor is an identifiable individual who visits a web site and is typically a major interest to most business users. However, quite often, the term ***Visitor*** is (incorrectly) used inter-changeably with ***Unique Visitor***. Measuring unique visitors is problematic because, at its core, the web is an anonymous/stateless environment. Thus, any attempt to measure or quantify unique visitors requires the application of business rules.

The term ***Click-stream*** (or click-stream data) is often used and is also subject to many interpretations. Some uses of the term click-stream data refer to any/all the data collected from the activity of users/visitors requesting pages from a web site. In the context of its intended use, however, the term click-stream is intended to mean a data-based representation of the path or sequence of activity (e.g., pages viewed) while a user navigates through a web site. It is this pathing data that is typically of high interest (e.g., what click-stream paths through my web site resulted in more or higher-value purchases?) to most business users. It is also this type of data that is most desirable for Data Mining.

Cookies are widely used, reviled and misunderstood. A cookie is simply a text file that is stored on a user's local machine and contains data specified by a web-server. The data is specific to that server and the data stored in a cookie are only available to the server that provided them. A standard Set-Cookie script allows the web server to specify the following:

- **Name**
  An arbitrary string containing the name of the cookie.

- **Value**
  Any specified value, as a string, to be stored in the cookie.

- **Expires**
  The datetime the cookie should expire (Greenwich Mean Time). Note that a value can be provided that indicates that the cookie should never expire.

- **Domain**
  Domain name of the server(s) that can read the cookie. A domain cannot read a cookie unless the server that created the cookie grants permission explicitly.

- **Path**
  Pathname in the domain for which the cookie is valid.

The form and location of these cookies depends on the browser being used.

Cookies can be used to provide a variety of facilities, including:

- The storage of user preferences
- A unique user identification
- Market Basket data

While cookies themselves are somewhat innocuous, they have been used in the past to do things that many users feel to be inappropriate or invasive. Thus, the use of cookies must be carefully evaluated in any web environment.

A cookie is often used to identify you when you return to a web site (e.g. AMAZON.COM). To accomplish this, the cookie stored some identification key that the site could translate to your name or some other value you have provided to that site in the past. When you visit the site again, the data in the cookie is provided by

your browser to the web server and that information is then used to populate the page you see.

Web Log Data

Web servers log their data to log files, creating the most common source of data. There are a number of standards, the first being **C**ommon **L**og **F**ormat (or CLF). The successors, by and large, are the **C**ombined **L**og **F**ormat or the **E**xtended **L**og **F**ormat (ELF) which includes additional commonly used/needed fields.

The ELF Data

The ELF fields include, among others:

- **Host**
  Host is the Internet address of the browser or other agent making the HTTP request and the location where the response will be sent. The value for host is the numeric IP address (e.g., "124.11.121.11") and that is the value that will be typically seen in web logs. Most web servers can resolve this address into a text domain using an Internet query protocol called reverse DNS (**D**omain **N**ame **S**erver) lookup. This is a process whereby the IP is replaced by the domain name (e.g., www.microsoft.com). This makes logs more readable, but can increase the load on the server dramatically if done in real time. More commonly, if DNS lookup is needed, the IP lookup is integrated with the Data Warehouse ETL process.

  At first glance the results of DNS appear to be potentially very useful (e.g. what percent of my visitor are from SAS.COM). However, upon examining the data it is only useful for a limited set of reporting functions. For example, a **B2C** site will typically find that the vast majority (e.g., upwards of 80%) of the IPs resolve to AOL.COM. Thus, knowing the actual domain name provides little discriminatory value.

  As discussed later, this IP value is commonly used to identify visitors and sessions. The IP value remains constant during a browser session, and can be used to tie events together where a more reliable mechanism such as a cookie or server-generated session ID is not available.

- **Ident**
  The ident data element is an arbitrary identifier that can be supplied by client applications that support the identd (identity daemon) protocol. Most common browsers do not provide this value and so this field is rarely if ever used.

- **Authuser**
  Authuser is a user ID that the web server will prompt the user for if HTTP authentication has been enabled on the web server. The user must enter a valid user id and password before the web server will provide any pages to the user browser. Only the user id is stored in the web log as providing the password would be a security breach.

  Figure 1 shows an example of a prompt that the browser will display if the web server signals that HTTP authentication is required.

Figure 1. HTTP Authentication

Many people incorrectly associate the

Authuser field with requests that are made to a secure server using the HTTP secure sockets layer (SSL). Such URLs begin with HTTPS instead of HTTP and are commonly used at sites where sensitive data (e.g., credit card numbers) are entered. The Authuser value is logged whenever HTTP authentication is enabled and that value is written to every web log record even though the user is only prompted once.

- **Time**
  Time is usually the time when the web server completed the response to the HTTP request. It is usually is set to GMT (Greenwich Mean Time). For web sites that are supported by multiple servers, atomic clock should utilities be used for all servers to ensure synchronicity.

- **Request**
  The request field contains the actual request line from the browser, for example “Get /mypage.html HTTP/1.0”.

  In this example, GET is the HTTP method, the next section is the Uniform Resource Locator (URL), and HTTP/1.0 is the protocol version being requested by the client. The two most common request methods are GET, which requests an object from the server, and POST, which sends information from the browser to the web server.

- **Status**
  Status is the three-digit status code returned to the browser from the server and indicates whether the page was returned to the user’s browser (and, if not, why not). Example values include:

  - 200 (OK)
  - 302 (Moved Temporarily)
  - 404 (Not found)

- **Bytes**
  Bytes is the count of bytes returned to the client by the server. It is seldom used, but is important to certain sites. For example, a site providing video or audio files which are large would likely make more use of this field (e.g., how many of my first-time visitors are downloading files larger than 300K).

- **Referrer**
  Referrer is a text string that contains the referring page, i.e., the URL of the page that contained the link a user clicked on to get to the current page. The referrer field allows you to trace how a user got to a page by navigating backwards through the web log.

  If a page contains images or embedded pages (e.g., framesets or iframes), then the referrer value for the page component is the container page that includes the references.

  It is the referrer field that typically provides the data source for **click-stream data**.

- **User-agent**
  The user-agent is the name and version of the client software making the request, and the corresponding operating system. This can be used to ensure that only content that can be supported is sent to the client. Unfortunately, this is very difficult to accomplish, as there is still not a widely used industry standard. For example, both Internet Explorer and Netscape have their own proprietary version of Dynamic HTML, Cascading Style Sheets, etc. and their own extensions as well as supporting some but not all of the supposed industry standard data elements.

  On an Intranet (as opposed to Internet) site, this is typically a more manageable

problem as standards can dictate what the supported browser (and version) is.

In addition to browsers, the user-agent field can also contain values for search engines, spiders, or web-bots that crawl the Internet (or an Intranet) to find, index and catalogue text.

An application that makes an HTTP request can provide any value it wants for the user-agent field. This field can also be easily changed by the end-user in their registry or other appropriate system location. In developing their WebHound solution, SAS Institute identified well over 1000 different values for user-agent based on just examining the web logs for their external site.

Data Usage Problems

When using data to investigate an issue, it is important to understand the nature of the data, how it was collected, and how it can be used. There are a number of issues relating to the quality of web intelligence data that must be considered.

The Visitor Problem

At the core of any web-intelligence effort is the need to uniquely identify visitors. However, as described above, the web is an anonymous and stateless environment. Thus, the identification and definition of a unique user must be dealt within the design of the web site. For some environments, uniquely identifying a user is straightforward, while for others it can be next to impossible.

For an Intranet site that requires authentication to the web-server, the user's id is automatically captured by the web-server, is written to the standard logs, and is easily made available to any web application server.

For the typical Internet site, web server authentication is not an option. Thus, other techniques must be used to identify visitors.

Cookies are commonly used to identify unique users. When a user visits a web site, the web server can request that a cookie be created on the user's pc and store some anonymous but unique identifier in that cookie. Any future visit to that web site will include the cookie value and so it is possible to identify a unique user (though this technique does not provide any other information about who the user is). There are a number of problems with this option:

- Some users may disable cookies
- If multiple users share a single machine (e.g., a home PC), the unique visitor identification is really identifying a household and not an individual
- A user may roam (e.g., use more than one pc) and have different cookie values on different PCs.
- The cookie on a specific PC represents that PC and not the specific individual user.

Regardless, the use of cookies is a widely used technique that can suffice for many situations.

Another commonly used technique is to request that the user log in or identify themselves to the site. However, this requires that the web site provide some incentive for a frequent visitor to log in vs. browsing *anonymously*. It also requires that the web site be designed so the values are propagated as the user navigates through the web site.

The most widely used technique groups together log records with the same IP number and assumes that they represent a single visitor if the time between successive

requests is less than some value (30 minutes is typically used).

The use of the IP number to identify unique visitors has limitations. First and foremost, the IP is not tied or associated with a specific visitor. A few scenarios where multiple users could share a single IP include:

- **Multi-user Machines**
  More than one person is using the same machine to browse the web site, and since the IP is associated with the machine, multiple users are indistinguishable.

- **Proxy Servers**
  A proxy server is used by many organizations and ISPs to cache commonly requested content. When a user makes a request for a page, it is the proxy server that is making the connection to the web site and so the IP of the proxy server is logged. Thus all of the individuals going through that proxy server are indistinguishable.

- **IP Reassignment**
  If a user logs off (e.g., disconnects from their AOL account), his or her IP is now available for use by someone just logging on.

It is also possible for a single user to have two IPs. Many Internet Service Providers (ISPs) have a policy that if there is no activity for some time period (e.g., 15 minutes), the user's connection is terminated. Consider the scenario where a user is browsing and stops on a page, gets distracted, and returns 20 minutes later and continues to browse. When they click on a link, a new ISP session is begun and they likely have a new IP. Alternatively, for sites with multiple proxy servers, any given request from a client to an external site can appear to come from any one of these proxies.

The Click-stream Problem

The *holy grail* of web intelligence is the use of click-streams or pathing data in order to gain insight into customer behavior. For example, if you can identify patterns in the click-stream that result in a higher propensity to buy, or that increases cross selling of related products, the usability and profitability of a web site can be improved.

There are (at least) two stumbling blocks to making use of click-stream data. First is ensuring that the click-stream path is collected completely and correctly. A second issue is the size of the data volume.

Consider the problem of ensuring that the click-stream path is collected completely and correctly. The data that is available is based on activity/requests made to the web server. However, there are external factors that can prevent a complete click-stream (from the user's perspective) from being collected. The issue is caching of pages – either by the user's browser or by a proxy server.

Follow the click-stream path below:

1. user starts at page New Orleans
2. clicks on a link to Shopping
3. clicks on a link to Clothing
4. clicks on a link to Jax Brewery
5. browser back button, to return to New Orleans
6. clicks on the link to Shopping again
7. goes back to New Orleans
8. clicks on a link to Mardi Gras
9. clicks on a link to Mardi Gras America
10. decides to buy
11. and continues.

So the user's actual click-stream path is:

New Orleans:Shopping:Clothing:Jax Brewery:New Orleans:Shopping:New Orleans:Mardi Gras:Mardi Gras America, etc. . .

Due to the fact that the browser has cached the pages, when the visitor uses their back button, there likely is no record at the web server of that activity, so the data that is logged might be:

New Orleans:Shopping:Clothing:Jax Brewery:Mardi Gras:Mardi Gras America, etc. . .

Further, suppose that the pages **New Orleans** and **Shopping** (being the main pages at a popular site) were cached by a proxy server (and lets ignore the issue of identifying the visitor for this example). The resulting click-stream recorded at our web site might be:

Clothing:Jax Brewery:Mardi Gras:Mardi Gras America, etc. . .

An analysis of this data might yield the conclusion that this is **THE** path that leads to higher-value purchases when in fact, pages **Clothing** and **Jax Brewery** are digressions and, in fact, many visitors to Clothing and Jax Brewery may actually abandon the site. The important path might be **New Orleans:Shopping:Mardi Gras:Mardi Gras America**. However, that is not what is recorded in our click-stream.

A compounding factor is browser favorites. By saving an intermediate page in the click-stream path as a favorite and returning to it later, the actual path that the user took may be spread out over several sessions. As a result, when the user makes the decision to buy, it is a truncated click-stream that is recorded in the log.

Next, consider the issue of data volumes. The click-stream paths that can be recorded are both very long and very voluminous. The patterns of interest are likely a small subset of a much larger click-stream. If one considers the entire click-stream, then many of the paths are going to be unique. The cardinality of the data is very high and so any analysis of the data is difficult at best.

As yet, no one has identified an apposite solution to these problems. However, the problem can be made tractable by scaling back what is examined. Instead of examining all of the click-streams, a site should consider identifying a small subset of pages of interest, and then build the click-stream paths for just that subset. There are at least two techniques that should be considered:

1. **Pathing To**
   Identify an ending page of interest (e.g., checkout) and then examine the click-steams that lead to this page.

2. **Pathing From**
   Identify a starting page and examine the click-streams that start from this page.

The Web-Intelligence Platform

In order to provide a robust solution that integrates the data/information available from web logs, it is both necessary and appropriate to consider the design of both the web site as well as the data warehouse. Each of these will be covered briefly in the following subsections.

Web Site Design

Depending upon the scope and purpose of the web site, there are a vast array of techniques and methodologies that can be employed. While many are specific to the scope/purpose, some are more general in nature. Some of the common design considerations are included below.

- **Dynamic vs. Static Pages**
  Web sites can be composed of a combination of both dynamic and statically generated pages. The primary advantage of static web pages is that such pages can be served quickly and easily. Dynamic pages can be customized to the user, and tracking access to them is less likely to be impacted by caching issues (by either a proxy server or the user's local browser cache). Dynamic pages place a higher demand on the web server. Sites can be built using either of these techniques or a combination. There are a number of techniques that can be employed for dynamic page generation, including:

  - Common Gateway Interface (CGI)
  - Servlets
  - URL rewriting

- **Content Labels**
  Content labels for pages allow the page events to be classified and coded for later analysis. These labels can either be coded manually, or may be generated by application files or directory structures automatically. Such an index can then be integrated with the web log extract system to be used as part of the Extract-Transform-Load (ETL) system. For both static HTML and dynamic HTML pages, tables must be set up and rigorously maintained. Possible classifications for content labels include:

  - Page source (static, dynamic)
  - Page template (catalogue, index)
  - Page function (site index, product catalogue, FAQ, announcement)
  - Item code (product ID)

  And others as mandated by the business requirements.

- **Use of a Null Logging Server**
  There are numerous names for this technique. It has been made popular (and to some extent, infamous) by Doubleclick. The basic idea is to embed a reference on a page to an image (or a cgi reference, etc.) that is located on that server. By embedding such an HTML tag into a web page, a single web log record is written on the null logging server that contains the requested image as well as the page that requested it (in the referrer field). That data can then be directly used for **click-stream** analysis. One technique commonly used to do this is to use an HTML IMG tag to request a one pixel sized transparent .gif file. This results in a record being written to the server web log. By using a one-pixel .gif (also called a ***webdot***), there is a minimal impact on the download time because of the small size of the picture. Thus, it can be done relatively transparent to the user.

  Alternatively, instead of using a separate web server, the **webdot** can be served by the same web server. This requires more effort during the ETL process in order to identify the **click-stream**.

- **Unique URLs**
  If the site is part of an overall CRM or Marketing Automation solution, then it is likely that prospects, customers, and users have received an email with a link to the site. Such URLs can be made unique to both the user receiving them as well as the marketing campaign. In order to fully leverage such data, it is necessary to make sure that the entire browsing session can be linked to the page. One way to do that is to count on the use of the referrer field to build the **click-stream** path. Alternatively, the site can be designed to propagate the values on the original URL (e.g., to identify both the user and the campaign).

- **Exit Point**
  Tracking the last page of a site that the user visits is of very high interest and value. Unfortunately, it is not easy to identify. While the referring field will tell you what page the user navigated from, there is no way for the web log to contain the page for which the user left (though if the page is on the same site, it can be determined). For example, if your site contains links to external sites, when a user clicks on one of them, there is no information written to your server's web log recording that action by the user. Since this is useful and important information to have, many sites will employ what is called a redirect page. When the user clicks on such a link, they can be brought to a page (either an HTML page or a cgi, asp or jsp application) that indicates that they are leaving the site. Then the site either makes them click again or automatically transfers them after a short time. The record written to the web server log contains both a page on the current site (thus is it logged) as well as the site the user is being transferred to.

- **Entry Point**
  Tracking the first page a visitor visits is also important. Fortunately, this can be determined in a straightforward manner once visitors/session tracking has been addressed. The entry point will typically be the first page visited at the site for any selected time window that has either a URL for another site in the referrer field (this is useful data as it identifies where the visitor is coming from) or no value at all for the referrer field. The referrer field can be blank for a number of reasons, the most notable causes are:

  - The user directly keyed the URL in the browser address field, or
  - A previously saved favorite was used, or
  - The user received an email with the link, which they clicked on to access the site.

Warehouse Design

There are a number of vendors and products that can be used to create a warehouse to support analysis of web log data, such as:

- WebHound from SAS Institute
- eSite from Informatica
- WebTrends
- Accrue
- NetGenesis

They each have their strengths and weaknesses. Instead of trying to compare these offerings, this paper will focus on general design principles that should be factored into the decision for the Warehouse platform.

Any warehouse platform must be scalable and extensible and should include, at a minimum, the following features or capabilities:

- **Warehouse Centric**
  The platform should be fully integrated with a data warehouse environment and should be based upon a data model that addresses both the specifics of web log data, and is extensible so that site specific characteristics and fields can be integrated into the warehouse.

- **A Standard Web Log Data Model**
  The model should address the following constructs at a minimum:

  - **Hosts**
    A table that contains information about all the hosts (internal and

referring) that appear in the web log.

- **Pages/URLs**
  Each unique page or URL that is surfaced by the site should be identified and should be integrated with the Content Label functionality as earlier described.

- **Sessions**
  Each unique session should be identified and information about it should be available from a sessions table.

- **Visitors**
  Each visitor should be identified in a table. If IP is used to identify visitors, then the visitors and the session tables will be one in the same. If some mechanism (e.g., cookies) is used to identify visitors then the visitors table should be linked to the sessions table using a one-to-many relationship.

- **Paths**
  Each unique path should be stored in a table. It may also be appropriate to define a table of unique paths by session or visitor.

- **Integrated with Other Channels**
  In order to maximize the effective use of data collected from web logs, the warehouse platform must enable easy integration with other enterprise data (e.g., customer history, product information, etc.). If the web log warehouse contains only information about web-site access, its usefulness will likely be limited to optimizing the web site itself rather than providing additional data that can be integrated into a CRM environment.

- **Static Reporting**
  There must be a rich set of publishable reports that contain information about web-site access and usage as well as information of business value (e.g., how many sessions, visitors, most common entry/exit points, typical paths to purchase, etc.)

- **OLAP Tools**
  Since not all reports can be generated statically, seamless integration with OLAP tools to allow business and technical users to slice-and-dice the data for discovering trends and patterns is an essential component of any warehouse platform.

- **Data Mining**
  Web logs can provide massive amounts of data and so integration with data mining tools that can be used to mine and discover information contained in the web logs is important. Data mining the web log information is an ongoing activity and should be targeted to address specific questions. There must be an ability to create data mining databases specific to certain problems. Trying to automatically mine the entire web log data is likely to be not nearly as productive as mining subsets of the data extracted for specific questions.

- **Web-based Platform**
  Reports and access to the data should be web-based and deployed to any user with a browser (with appropriate security, as necessary). It is an oxymoron to require proprietary desktop applications to browse and mine web logdata.

Summary

This paper has provided a brief overview of the fundamental elements of web-intelligence. Questions for the authors can be directed to:

Don.Henderson@us.pwcglobal.com
Ralph.Mittl@us.pwcglobal.com

Case Studies in Data Management on the Web

Carol Martell, UNC Highway Safety Research Center, Chapel Hill, NC

ABSTRACT

The Highway Safety Research Center(HSRC) at the University of North Carolina in Chapel Hill uses the data management capabilities available through SAS/IntrNet® CGI Tools for several projects. This paper will examine three applications. The PBCAT Order application captures information, forwarding relevant fields to a fulfillment house. The Walk to School application collects data from the web and quickly resurfaces that data after human scrutiny. The PedBike Information System is similar to a problem tracking system. Questions are directed to a team of experts. Both questions and answers are entered into the system, creating a searchable database. This paper describes the structure of these projects, which incorporate base SAS®, SAS/SHARE® and two SAS/IntrNet components (htmSQL and the Application Dispatcher) in a Solaris environment.

INTRODUCTION

For each case study we begin with how the application behaves through a web interface. The parameter section describes project requirements that give direction to the system design. The solution overview describes the general organization and dynamics of the system. The data management section highlights coding techniques.

PBCAT ORDER SYSTEM

This software available free of charge on CDROM helps a customer categorize pedestrian and bicycle crashes. The order form is available on the virtual Web site hosted at Highway Safety (Figure 1).

Figure 1

The process begins when a customer submits an online order (Figure 2).

Figure 2

The order arrives at the fulfillment house in an email message:

Subject: PBCAT CD-ROM Order 868796
Date: Mon, 28 May 2001 12:45:53 -0400 (EDT)
From: somewhere@server.unc.edu
To: fulfillment@thatplace.com

Please send a PBCAT CD-ROM to the following person:

Ms Carol Martell
UNC HSRC
730 Airport Rd CB# 3430
Chapel Hill NC 27599-3430
USA
phone 919-962-2202
email carol_martell@unc.edu

This order was placed on 28MAY01

The customer receives acknowledgement email:

Subect: PBCAT CD-ROM Order 868796
Date: Mon, 28 May 2001 12:45:53 -040(EDT)
From: somewhere@server.unc.edu
To: carol_martell@unc.edu

Thank you for ordering the PBCAT CD-ROM.
We will mail it to:

Ms Carol Martell
UNC HSRC
730 Airport Rd CB# 3430
Chapel Hill NC 27599-3430
USA
phone 919-962-2202
email carol_martell@unc.edu

This order was placed on 28MAY01

Please wait 15 business days for delivery.
If you do not receive your cd within that time, please send email to:
contact@unc.edu and reference order number 868796.

If the customer does not receive the software, they may follow up

with the contact provided, who has access to a dynamic web page showing all orders placed (Figure3). Each order has links allowing address correction and reordering:

24MAY01	USA	IL	ingrid	weisenbach	update record	delete record	send email	reorder
25MAY01	USA	ID	Christopher	Bates	update record	delete record	send email	reorder
25MAY01	USA	NJ	Fran	Sanchez	update record	delete record	send email	reorder
28MAY01	USA	NC	Carol	Martell	update record	delete record	send email	reorder

Figure 3

PARAMETERS

The fulfillment house is external to HSRC, so orders must to be forwarded to them. Their legacy system only accepts manual input, so we could not code something to automatically feed into their system. Instead, order requests are submitted to them by email. Because HSRC plans email notification to previous recipients of future software releases, order information is retained.

SOLUTION OVERVIEW

A single SAS table houses the order data. New information is added to the table through the Web form in Figure 2. Unacceptable mailing addresses, obviously invalid email addresses, and inconsistent zip codes are returned to the browser for correction and resubmission. Email to order the software is automatically sent to the fulfillment house when a record is added. At the same time, an email message acknowledging the order is sent to the customer. In-house maintenance is available through the dynamic web page (Figure 3) that lists each record with links enabling various actions. From this page we can, for instance, correct an address and resubmit the order for a customer.

DATA MANAGEMENT

The Application Dispatcher is used for data entry. All the information from the web form is passed in to a SAS program as macro variables. In the SAS code, a temporary, one-observation table is created for the order. Data is checked for completeness and validity. The following sample code shows email address screening:

```
email="&email";
if email = '' then
 do;
 str='We need your email address';
 goto errorm;
 end;
if index(email,'@')=0 or
   index(email, '.')=0 then
 do;
 str='We need a viable e-mail address';
 goto errorm;
 end;
```

If problems are encountered, the customer is prompted to correct the problem and resubmit. Further processing must be prevented to avoid sending empty email messages. This is accomplished by using a macro variable (&quit):

```
errorm: do;
file _webout;
put 'Content-type: text/html' ; put;
put '<h1> Please provide all required
information: </h1>';
put str "<br>";
put "Please provide the missing information
and resubmit <br>";
put "Thank you </html>";
call symput('quit','yes');
stop; end;
run;

data _null_;
x=symget('quit');
if x='yes' then do;
                abort abend;
                end;
run;
```

If all required data items are present and acceptable, the temporary table is appended to the main table. Next, the order is sent:

```
data;
set temporarytable;

filename sendit email
"fulfillment@thatplace.com"
subject="PBCAT CD-ROM Order &order";

file sendit;
put 'Please send a PBCAT CD-ROM to the
following person:';
put ' '/salutation first_name last_name /
organization;
…
```

The acknowledgement message is composed similarly. The wording differs and the email filename statement contains the customer's email address in place of the fulfillment house email address.

Customer inquiries about an order are resolved through the in-house maintenance htmSQL page (Figure 3). This page lists each order record with four links to other htmSQL pages. The six-digit order reference number, created with a random number generator, is used to specify the record in the links. The four actions available for a record are update, delete, email and reorder. We examine the entire htmSQL code for the main page. We begin with the opening {query} directive pointing to the SAS/SHARE server:

```
{query server="myshareserver"}
```

Next we select all records from the table:

```
{sql}
select * from mylib.cdrom
order by date
{/sql}
```

We will present the results in a single HTML table, so we must first open the table and provide header information:

```
<font size=+2>
  PBCAT Order List for Maintenance</font>
<table border=0>
<tr>
  <td>order date</td>
  <td>country</td>
  <td>state</td>
  <td>first name</td>
  <td>last name</td>
  <td>update record</td>
  <td>delete record</td>
  <td>email customer</td>
  <td>reorder for customer</td>
</tr>
```

We want an HTML table row for every observation. We specify in

the {eachrow} section the HTML formatting and variable display for a single observation. This specification will be applied to each row returned from the query:

```
{eachrow}
<tr>
  <td>{&orderdate}</td>
  <td>{&country}</td>
  <td>{&state}</td>
  <td>{&firstname}</td>
  <td>{&lastname}</td>
  <td> <a href="update.hsql?o={&ord}">
          update record</a></td>
  <td><a href="remove.hsql?o={&ord}">
          delete record</a></td>
  <td><a href="emailem.hsql?o={&ord}">
          send email</a></td>
  <td><a href=
"...broker8?_program=a.reord.sas&o={&ord}">
          reorder for customer</a></td>
</tr>
{/eachrow}
```

The first five HTML columns above display variable values. The next three build the links with accompanying name/value pair. The last column is an Application Dispatcher call to a program that sends a reorder request to the fulfillment house.
Having composed the row, we close the {eachrow} section. Next we close the table and the {query} section:

```
</table>
{/query}
```

The update.hsql page is exactly like the original order form except that it is already populated with the observation's values. We can easily accomplish this by making a copy of the order form, naming it 'update.hsql'. We place a query to select a specific observation at the top.

```
{query server="myserver"}
{sql}
select * from mylib.cdrom where order="{&o}"
{/sql}
```

Then we surround the form with {eachrow} directives, supplying values for each field. For example, where the order form contained an input field for first name with no value supplied:

```
<input name="firstname" value="">
```

the update form contains:

```
<input name="firstname" value="{&firstname}">
```

The order number must be passed along in a hidden field. Finally, we change the _program name to a different SAS program - one that updates an observation's values instead of the one that adds a new observation.

The code for the htmSQL page to delete a record could contain an {sql} section as simple as the following:

```
{sql}
delete from mylib.cdrom where order="{&o}"
{/sql}
```

Weaving back and forth between htmSQL and Application Dispatcher allows the developer to take advantage of the strengths of each tool.

WALK TO SCHOOL 2001

This project is for a campaign to identify obstacles preventing children from safely walking to school. HSRC hosts a website (Figure 4) with an online registration application to help event coordinators in the USA plan and publicize their local event. Coordinators provide information and we surface it to the web.

Figure 4

Figure 5

When someone organizing an event registers online (Figure 5), an acknowledgement window appears in their browser window (Figure 6).

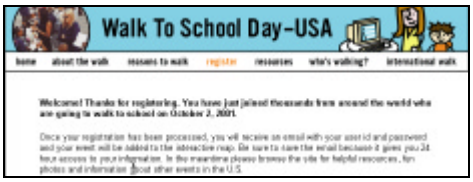


Figure 6

The record immediately becomes available for in-house screening (Figure 7).

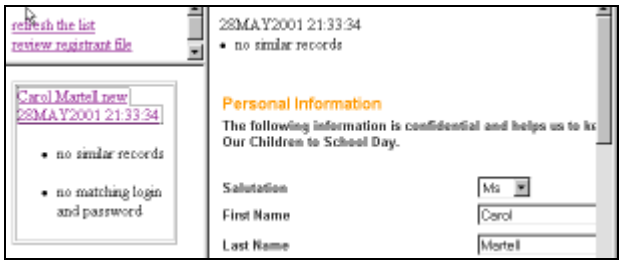


Figure 7

The screener can delete a bogus registration, approve it as is, or correct typos and approve corrected version (Figure 8). Upon approval, the registrant receives a 'welcome' email message (Figure 9). The message includes a link for updating their record. The link brings up a registration form already filled out with their information, ready for additions (Figure 10).

Figure 8

Figure 9

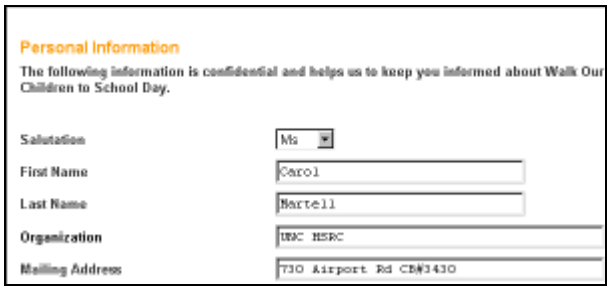


Figure 10

All previously approved registrations are also available for in-house review in an update mode (Figure 11).

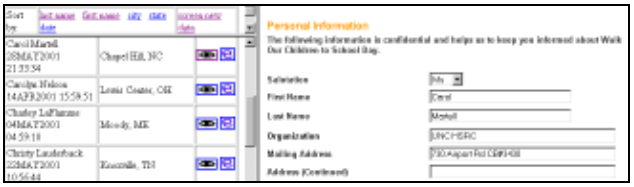


Figure 11

The image map on the web site is the starting point for the public to view event details for all registered events (Figure 12). States with registered events are highlighted. Within ten minutes of registration approval, a new registrant's state becomes highlighted on the map (Figure 13).

Figure 12

Figure 13

Clicking on the state brings up a list of communities with registered events (Figure 14), and clicking on a community name brings up event descriptions for that community (Figure 15)

Figure 14

Figure 15

PARAMETERS

This application is streamlined and otherwise improved from the previous year's registration application. Goals included avoiding problems that cropped up the previous year. Registrants reluctant to navigate a web site or to supply login and password can take advantage of the update link provided in the welcome email message. Those who mistakenly register anew instead of updating are flagged on the screening page, minimizing the chance of duplicating records. Registrants are allowed to choose their own passwords. The code to refresh the image map and the code to send the welcome email message is placed in frequently run scheduled batch SAS jobs. No Application Dispatcher is used in the system; it is written entirely in htmSQL and scheduled batch jobs. Presenting a record to be screened in an update form allows for on-the-spot typo correction. We introduce the use of frames to simplify navigation for data maintenance.

SOLUTION OVERVIEW

A single SAS table houses the screened registration data. Each unscreened registration or update is a separate table that is deleted after screening. To flag instances where the customer mistakenly registers anew rather than submitting an update, the dynamic screening page provides links to display similar preexisting registrations. Since passwords are user-defined, duplicate logon information is also displayed. New submissions and preexisting records are displayed side-by-side on the screening page for easy comparison. Both new and preexisting records are displayed in a populated form; each can be accepted, modified or deleted.

DATA MANAGEMENT

Record flags are used to accomplish tasks requiring a data step. A batch job is scheduled to run periodically that checks for the welcome message flag. All new registrations have this flag set. The job sends a customized message to the email address in each record having the flag set and then resets the flag to 0. A batch job publishes the image map of the US every 10 minutes. Other tasks are accomplished using htmSQL. The screening page frameset defines the layout seen in Figure 16.

```
<frameset cols="20%,40%,40%">
  <frameset rows="10%,90%">
    <frame src="menu.html">
    <frame  src="screen.hsql"
         name="thelist">
  </frameset>
  <frame name="incoming">
  <frame name="existing">
</frameset>
```

This code divides the page into three vertical columns (frames), the first of which is also divided horizontally. Each frame given a name can be the target of a link from elsewhere on the page. We use the convention of always displaying new data in the center column, and always displaying preexisting data in the far right column…hence the names 'incoming' and 'existing'. The bottom left frame displays screen.hsql, listing all new records for screening.

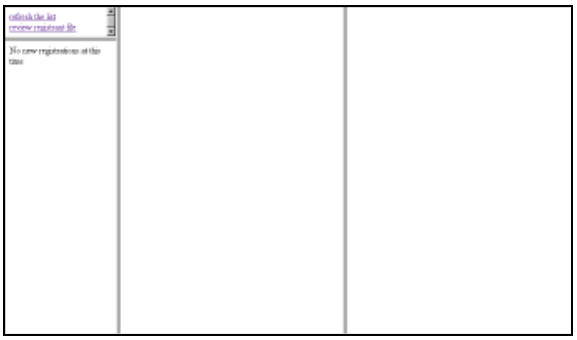


Figure 16

We examine the code for screen.hsql. We query the dictionary tables to find all tables other than the main registration table. If there are none, we display a message to that effect:

```
{sql}
select *  from
dictionary.tables where libname="WOCSLIB"
and memname ne "REGISTERED"
{/sql}
{norows}No new registrations at this time
{/norows}
```

For each registration found, we must look for similar names and watch out for duplicate name/password combinations. We use the nesting capabilities of htmSQL to accomplish our various tasks:

```
{eachrow}
  {sql}
    select fnameas f, lname as l, type as t,
    uid as unew, pword as pnew,
    "{&memname}" as filename
    from wocslib.{&memname}
  {/sql}
```

We build a link targeting the center column to display the new data. The variable type, aliased as t, specifies whether the record is a new registration or an update, taking on values 'new' and 'upd'. The link will resolve to either viewnew.hsql or viewupd.hsql. They must differ because if the data is new it should be added while updates replace an existing observation.

```
{eachrow}<a href=
         "view{&t}.hsql?f={&filename}"
          target="incoming">
         {&f} {&l} {&t} {&crdate}
         </a>
```

We now look for similar preexisting data, comparing the names using soundex and spedis functions. If we find none, we display a message to that effect. For each record found we construct a link to display the existing data in the far right column:

```
      {sql} select uid as u, pword as p,
            first_name as fn, last_name as ln
            from wmaint.registrants
         where
         (
           soundex(first_name)=soundex("{&f}")
           and
```

```
        soundex(last_name)=soundex("{&l}")
      )
      or
      (
        spedis(first_name,"{&f}")
        +
        spedis(last_name,"{&l}")
        <50
      )
    {/sql}
    {norows}no similar records{/norows}
    similar records already registered
    {eachrow}<a href=
             "viewold.hsql?u={&u}&p={&p}"
             target="existing">
             {&fn} {&ln}
             </a>
    {/eachrow}
```

We perform another search for exactly matching preexisting logon information (the login has been constructed by compressing together first and last names). Again we either announce that there are no matches or build links to these similar records, targeting the far right frame. We close the encompassing eachrow sections:

```
    {sql} select uid as ux, pword as px,
          first_name as fn, last_name as ln
          from wmaint.registrants where
         ux="{&unew}" and px="{&pnew}"
    {/sql}
    {norows}no matching login and password
    {/norows}
    already registered with login/password
    {eachrow}<a href=
             "viewold.hsql?u={&ux}&p={&px}"
             target="existing">
             {&fn} {&ln}
             </a>
    {/eachrow}
  {/eachrow}
{/eachrow}
```

HTML formatting to display these results is not included in the code above. The formatting we use organizes the gathered information about each new record into a table with a border. The similar names and matching login/password records are displayed as unordered lists. Consequently, the person screening data sees a box for each new record, with bulleted lists of preexisting data that should be used for comparison.

Data screeners are very happy with this solution. Providing access to all relevant information in a single Web page has proved to be extremely advantageous.

PEDBIKE INFORMATION SYSTEM

The use of frames for the Walk to School application was applied to another Web data management project. This project functions like a problem tracking system. A panel of experts answers questions posed by the public through a variety of avenues. Questions and answers are logged into the system through the web. The single Web page for this application is divided into five frames (Figure 17). The top left frame provides a menu. We will examine the avenues available there. Suppose someone named David Harkey poses a question. The first step is to determine whether or not he is already in the system. Entering a portion of his name, 'hark', and clicking 'find' (Figure 18) sends a list of matching names to the lower left frame. We see that a record is found for David Harkey (Figure 19).

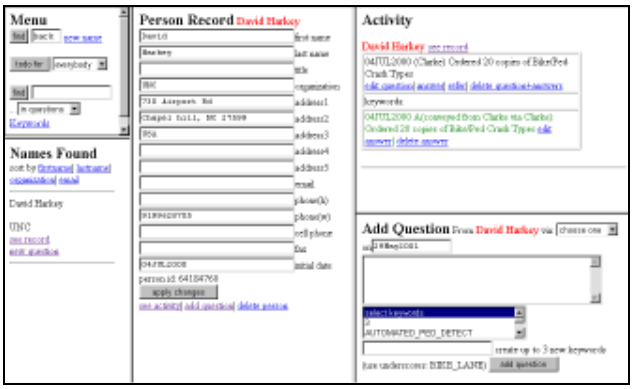


Figure 17

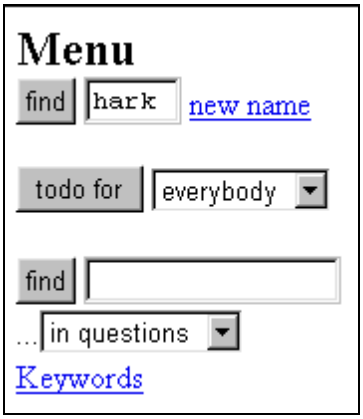


Figure 18

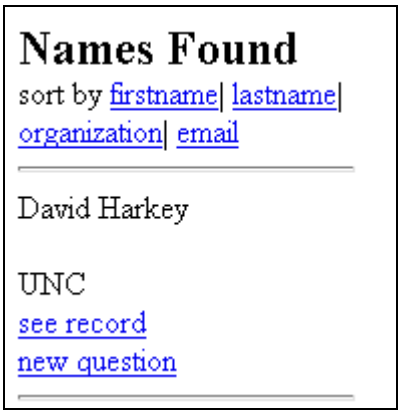


Figure 19

Clicking 'see record' sends his personal information record to the middle frame (Figure 20). It is displayed in a form so that information can be easily added or corrected.

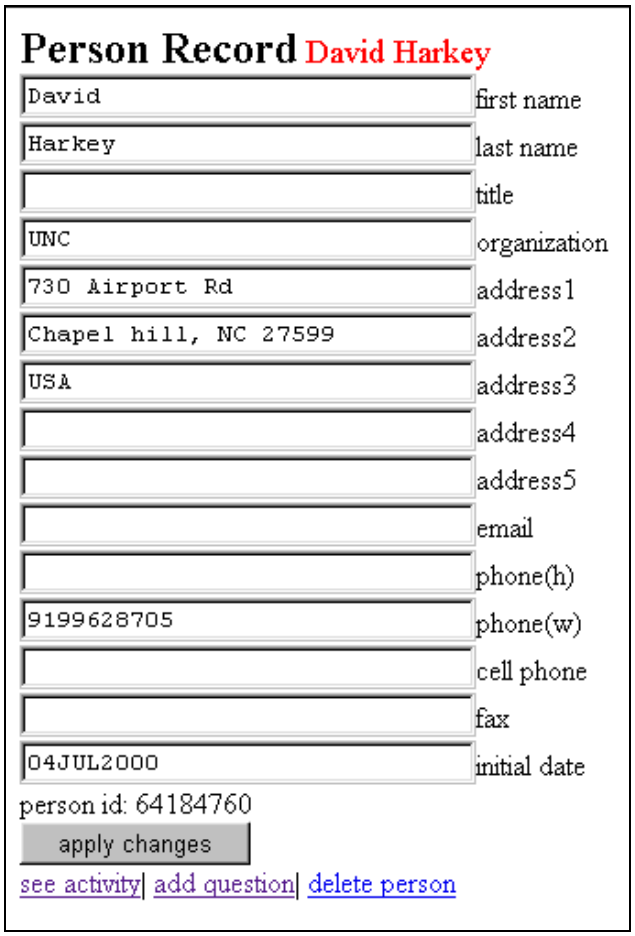


Figure 20

Clicking 'see activity' sends a list of David's questions with the answers and outstanding referrals to the top right frame (Figure 21). Clicking 'add question' sends a form to the bottom right frame (Figure 22). Clicking 'answer' in the activity frame sends a form to the bottom right frame (Figure 23). Clicking 'refer' in the activity frame sends yet another form to the bottom right frame (Figure 24).

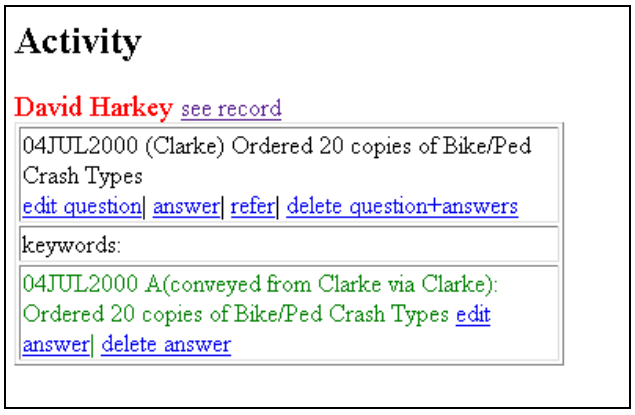


Figure 21

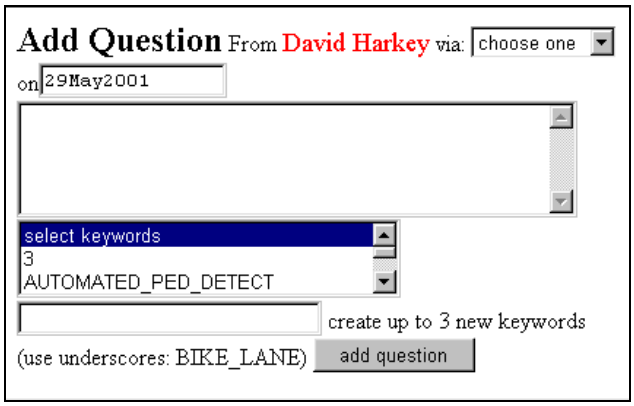


Figure 22

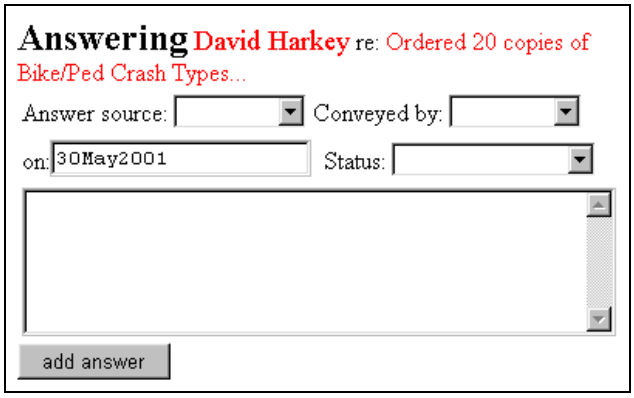


Figure 23

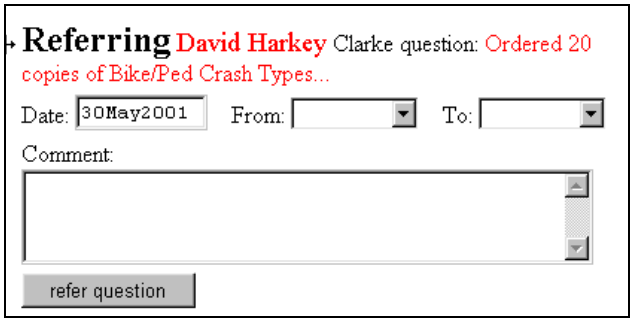


Figure 24

Dynamic select lists used throughout the forms provice choices for items such as the expert's name or the question source (Figure 25).

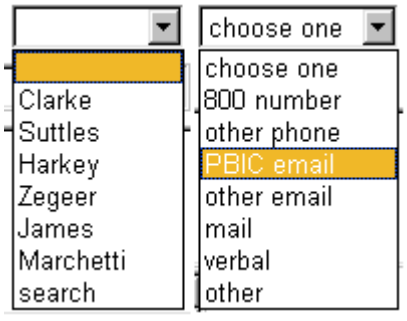


Figure 25

To explore another avenue, we return to the menu frame (Figure 18) and click 'todo'. A list of four followup categories appears in the bottom left frame (Figure 26).

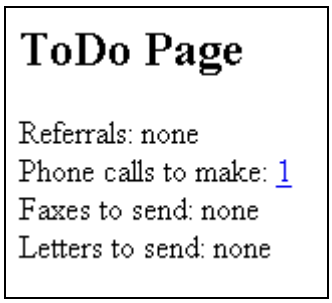


Figure 26

Beside each category is a numbered link for every item requiring attention. We click the link to see a not-yet-conveyed answer in the top right frame (Figure 27). The answer not yet conveyed is displayed as a form with conveyance marked in red. Once the answer is conveyed, the expert clicks the 'completed' button and the item will no longer show up in the 'todo' list.

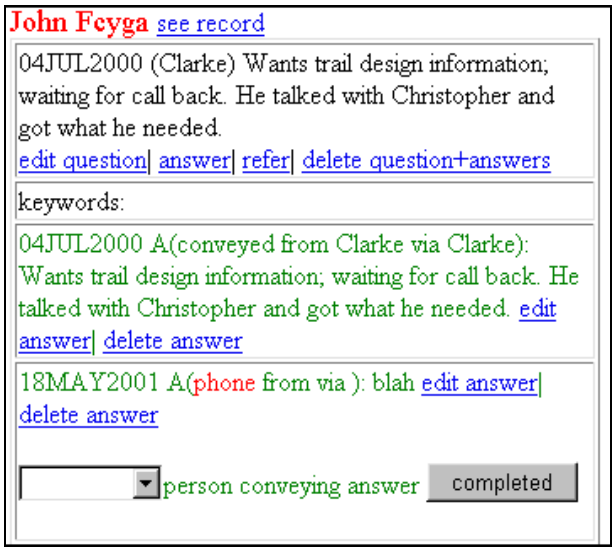


Figure 27

The menu (Figure 18) offers the ability to search questions or answers for a word or phrase. We enter the search term, choose the file to search, and click 'find' (Figure 28). Results are displayed in the lower right frame (Figure 29). Each matching item has a numbered link. Clicking the link would display the specific question and answer(s) in the top right frame.

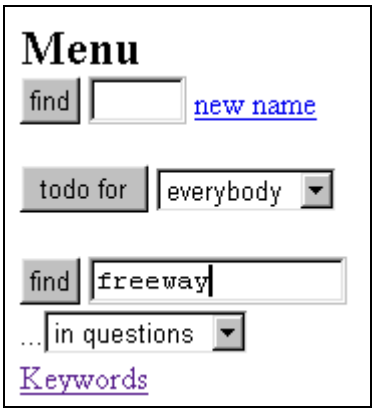


Figure 28

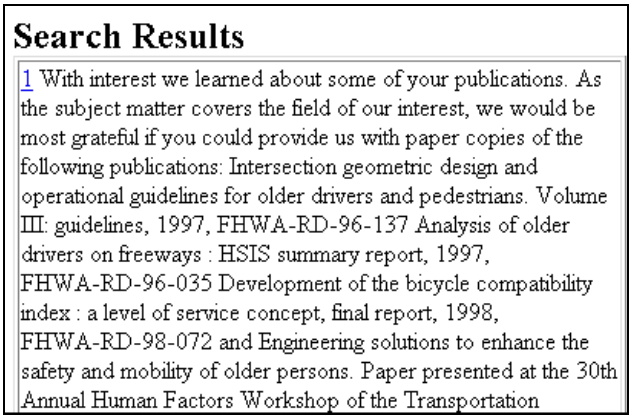


Figure 29

Now we choose the 'keyword' link from the menu frame (Figure 18). We see, in the lower right frame, three forms, two of which have select lists (Figure 30). These forms manipulate keywords. The first allows modification of an existing keyword.

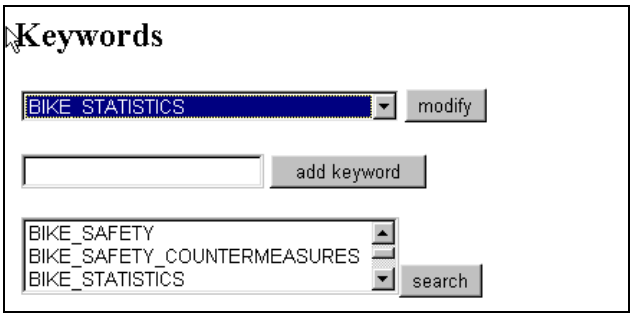


Figure 30

Clicking modify brings up three choices (Figure 31). One might find that two keywords should be collapsed into one, or that a keyword was misspelled and should be replaced with a new, correctly spelled keyword. Keyword deletion is also available.

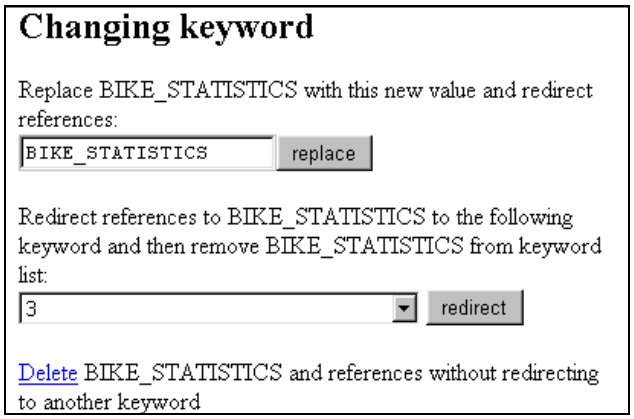


Figure 31

The keyword search form (Figure 30) allows multiple selection of keywords to find questions flagged with all the selected keywords. The search results (Figure 32) follow the familiar conventions: clicking the result number causes the question to display in the top right frame (Figure 33) with the usual accompanying information and links.

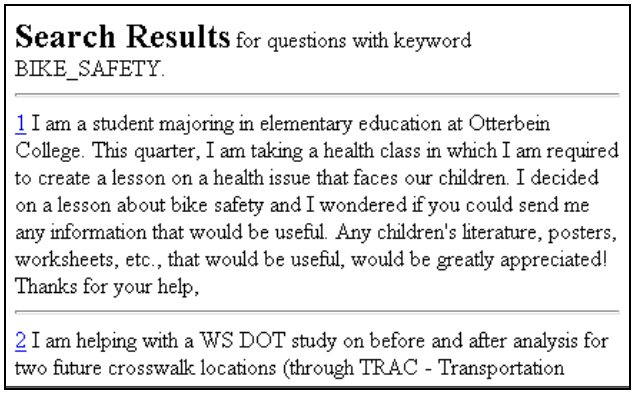


Figure 32

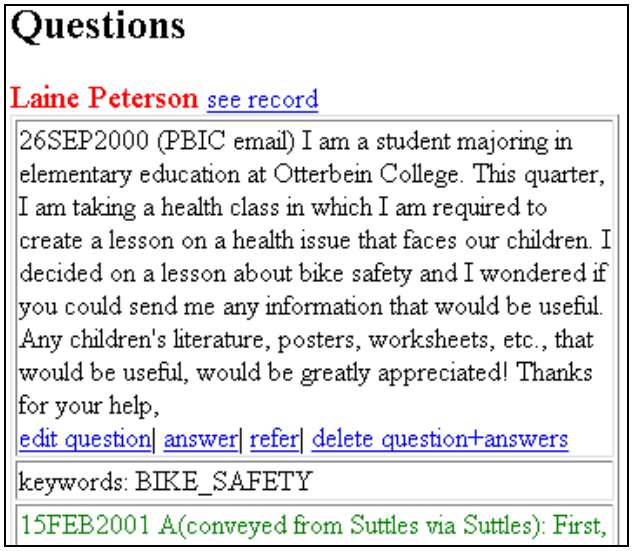


Figure 33

PARAMETERS

The system needed to track who is asking questions, what those questions are, who is answering the questions, and what the answers are, and needed to be searchable to allow reuse of previously supplied answers. An expert should be able to refer a question to another expert. All question and answer activity for a customer should be available to help provide the expert with a context. Determining whether someone is new versus already in the database should be extremely easy. A list of keywords to select and associate with questions must be included, along with the ability to add new keywords. Since the experts are scattered around the country, a web application was requested.

SOLUTION OVERVIEW

There are separate tables for the questions, answers, people, keywords, referrals, and experts. Each person, question, answer and referral is assigned a unique id. These are used to link the information for display. Where possible, answers are automatically sent by email or fax, using scheduled batch jobs that search the answer table for flagged records, send the answers, and change the flag values. Keyword management with retroactive action is included.

DATA MANAGEMENT

We will examine the htmSQL page invoked by clicking the 'activity' button from the person record (Figure 20). The person id has been passed in as 'p'.

```
<h2>Activity</h2>
{query server="myserver"}
{sql}
    select fname, lname from pbic.person
    where pid={&p}
{/sql}
{eachrow}{* display name, person rec link}
         <font size=+1 color="red">
         {&fname} {&lname}
         </font>
         <a href="person.hsql?p={&p}"
           target="tm">
           see record
         </a><br>
{/eachrow}
{sql} {* find questions for person}
    select qid as q, * from pbic.question
    where pid={&p} order by  date
{/sql}
{norows}
     <br>no activity
{/norows}
{eachrow}
   <table width=350 border=1>
   <tr><td>{&date}({&source}) {&text}<br>
    <a href="editques.hsql?p={&p}&q={&q}"
       target="br">edit question</a>|
    <a href="newansw.hsql?p={&p}&q={&q}"
       target="br">answer</a>|
    <a href="newref.hsql?p={&p}&q={&q}"
       target="br">refer</a>|
    <a href="deleteques.hsql?q={&q}"
       target="br">delete question+answers
       </a>
   </td></tr> {* show keywords}
   <tr><td>keywords: {&keyword}</td></tr>

 {query server="myserver"}{* find answers}
 {sql}
  select *,
  {* words/fonts vary by status}
  case sendthis
      when 'a' then 'conveyed'
      when 'p' then
        '<font color="red">phone</font>'
```

```
      when 'f' then
        '<font color="red">fax</font>'
      when 'x' then 'autofax'
      when 'e' then 'autoemail'
      when 'm' then
        '<font color="red">snailmail</font>'
      else '<font color="red">unknown</font>'
      end as status,
  {* HTML comment tags to fake out browser}
  case
     when sendthis in( 'a'  'x'  'e')
      then '<!--' else ''
      end as op,
  case when sendthis in( 'a'  'x'  'e')
      then '-->' else ''
      end as cl
  from pbic.answer
  where pid={&p} and qid={&q}
 {/sql}
{* formatting for each answer follows}
 {eachrow}
  <tr>
  <td>
    <font color="green">{&date} A({&status}
    from {&source} via {&provider}): {&text}
    <a href=
      "editansw.hsql?a={&aid}&p={&p}&q={&q}"
       target="br">edit answer
    </a>|
    <a href="deleteansw.hsql?a={&aid}"
       target="br">delete answer
    </a>
{* only need form for unconveyed answers}
{* surrounding HTML comment tags override}
    {&op}
      <form action="sentthis.hsql"
            target="br">
    {&cl}
    {&op}
      <select name="prv">
       <option value="">
    {&cl}
    {&op}
      <input type="hidden" name="p"
         value="{&p}">
    {&cl}
    {&op}
      <input type="hidden" name="a"
         value="{&aid}">
    {&cl}
    {&op}
      <input type="hidden" name="q"
         value="{&q}">
    {&cl}

    {sql}
       select distinct source as pr from
       pbic.sources
    {/sql}
    {eachrow}
     {&op}
      <option value="{&pr}">{&pr}
     {&cl}
    {/eachrow}
    {&op}
     </select>person conveying answer
     <input type="submit"
      value="completed">
     </form>
    {&cl}
   </td></tr>
  {/eachrow}
 {/query}{* ends answer query}
 {* search for referrals not yet addressed}
 {query server="myserver"}
  {sql}
   select * from pbic.referral
   where pid={&p}and qid={&q}
  {/sql}
  {eachrow}
   <tr><td>{&date}Referred to {&refto} by
    {&source}
    {&comment before=" with comment: "}<br>
   <a href=
    "newanswref.hsql?r={&rid}&q={&q}&p={&p}"
     target="br">respond to referral</a>
   </td></tr>
  {/eachrow}
  </table><br>
  {/query} {* ends referral query}
 <hr>
{/eachrow}
{/query}{* ends question query}
```

CONCLUSION

Web data management allows a distributed population to add, modify and query shared data. In this examination of case studies we have seen that a tailored solution can be designed to suit project requirements.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors at:

Carol Martell
UNC Highway Safety Research Center
730 Airport Rd, CB# 3430
Chapel Hill NC 27599-3430
Work Phone: 919-962-8713
Fax: 919-962-8710
Email: carol_martell@unc.edu
Web: www.hsrc.unc.edu

Using the SOCKET Access Method to Invoke SAS Programs

Rick Langston, SAS

Abstract: This paper discusses a prototype application that sends SAS code to 3 SAS sessions. The three SAS sessions are listening on sockets for SAS code, and execute the SAS code and set semaphores when the blocks of SAS code are completed. The Prototype application is also a SAS application, and it post-processes output that each SAS session produces. Used in the applications are the SOCKET access method, **%INCLUDE**, and the **SLEEP** function.

Obtaining and Using Euro Currency Rates in SAS® Programs

Rick Langston
SAS Institute, Inc., Cary, NC

BACKGROUND

The euro became the official currency of the European Economic Community on January 1, 1999. On that date, the currencies of eleven countries of Europe became the euro in lieu of their historical currencies. Those eleven currencies are the Belgian franc, the German mark, the Spanish peseta, the French franc, the Irish pound, the Luxembourg franc, the Dutch florin, the Austrian schilling, the Portuguese escudo, and the Finnish markka. Other countries will eventually join the list as their respective governments and voting public decide to do so. (Greece has since done this, on January 1, 2001). On September 28, 2000 Denmark voted down conversion to the euro, as another case. Other European countries are in various phases of decisions on the subject of conversion.

The SAS System provides a set of functions and formats to facilitate the conversion between the euro and historical currencies, and also conversion between the euro and the currencies of non-adherent countries. These formats and informats are called EURFRxxx and EURTOxxx (where xxx is the standard 3-character abbreviation for the country). The function is EUROCURR, which allows for conversion between any two currencies.

The informat/format/function set (herewith referred to as the euro IFF set) use a built-in table to obtain conversion rates. This works fine for the eleven initial adherent countries, because their conversion rates were made irrevocable with the adoption of the euro. However, for all other currencies, their rates fluctuate constantly in an open currency market. This aspect was recognized when the euro IFF set was first implemented, and the rates can be specified by an external table and/or macro variables, either of which will override the builtin rates.

This paper describes a method for obtaining the most current rates and incorporating them into a dynamic table so that the euro IFF set will give properly conversion results.

OFFICIAL CURRENCY RATES

The offical governing bank for the euro is the European Central Bank (ECB). Their web site, www.ecb.int, is the official web site to obtain rates pertaining to the euro. As of the time of this writing, the specific web page for obtaining currency rates is

http://www.ecb.int/home/eurofxref.htm

This web page contains rates displaying in the following fashion (using a snapshot from May 29, 2001):

Currency	Spot	Currency	Spot
USD US dollar	0.8552	LVL Latvian lat	0.5416
JPY Japanese yen	102.88	MTL Maltese lira	0.3955
DKK*Danish krone	7.4575	PLN*Polish zloty	3.4529
GBP*Pound sterling	0.60320	ROL*Romanian leu	24540
SEK*Swedish krona	9.0350	SIT*Slovenian tolar	217.5675
CHF*Swiss franc	1.5260	SKK Slovakian koruna	42.938
ISK Icelandic krona	88.24	TRL*Turkish lira	956700
NOK*Norwegian krone	7.8865	AUD Australian dollar	1.6608
BGN Bulgarian lev	1.9461	CAD Canadian dollar	1.3167
CYP Cyprus pound	0.57690	HKD Hong Kong dollar	6.6704
CZK*Czech koruna	34.219	NZD New Zealand dollar	2.0285
EEK Estonian kroon	15.6466	SGD Singaporean dollar	1.5449
HUF*Hungarian forint	254.00	KRW South Korean won	1102.78
LTL Lithuanian litas	3.4217	ZAR South African rand	6.7732

Because the web page consists of an HTML table and it is displayed based on your browser, the information seen above won't display the same way on your screen. I have made two other modifications to what is seen above: all decimals line up, and I've put a * beside the currencies actually recognized by the euro IFF set. Also recognized by the euro IFF set but not displayed above are the Russian ruble (RUR) and the Yugoslavian dinar (YUD).

Note the use of the 3-character abbreviation. This is the same abbreviation used by the euro IFF set. Not all currencies listed here are recognized by the euro IFF set.

Note also that these rates indicate the value of 1 euro in the specified currency. For example, 1 euro was worth .8552 US dollars on May 29, 2001. If you wanted to know how many euros were in one US dollar, you'd need the reciprocal of this rate (1/.8552, or 1.1693). This means it took 1.1693 euros to make one US dollar on May 29, 2001.

EXTERNAL TABLES WITH THE EURO IFF SET

The euro IFF set can use an external table for its rates. The table is stored in a file referenced by the EURFRTBL fileref. The entries in the table are as follows:

EURFRxxx=rate1
EURFRyyy=rate2
EURFRzzz=rate3

where xxx, yyy, and zzz are 3-character abbreviations for currencies. The values indicated by rate1, rate2, and rate3 are the number of units of the currency comprising one euro, just like the rates that appear in the ECB web site table. For example, the EURFRTBL entry for pounds sterling, using the rates seen in our May 29 table above, would be

EURFRGBP=0.60320

As a complete example:

```
filename eurfrtbl temp;
data _null_; file eurfrtbl;
     input; put _infile_; cards4;
EURFRGBP=0.60320
;;;;

data _null_;
     n_euros = eurocurr(1,'gbp','eur');
     n_pounds = eurocurr(1,'eur','gbp');
     put n_euros= n_pounds=;
     run;
```

The result is

```
n_euros=1.6578249337
n_pounds=0.6032
```

The EUROCURR function has the arguments

```
to_units =
eurocurr(from_units,from_curr,to_curr);
```

So in our example, the first use of EUROCURR is to convert 1 pound sterling into euros. The second use is to convert 1 euro to pounds sterling. As we expect, n_pounds is equal to the rate given in the EURFRGBP value.

MERGING THE ECB TABLES WITHIN THE SAS PROGRAM

We now know we can obtain current rates from the ECB web site, and we can dynamically specify a table for the euro IFF set to use. So here's how we can merge these abilities into a single SAS program.

In version 8 of the SAS System, the HTTP access method was made generally available. We can use this access method to read web pages directly. The syntax for this access method is as follows:

```
filename fileref HTTP 'web-page-address'
authentication-info;
```

where fileref is the filefef you want to use 'web-page-address' is the web site address, and authentification-info is whatever is necessary to access external web pages. It may be likely that you'll need a userid, password, and proxy address, depending on your security setup:

```
filename fileref HTTP 'web-page-address'
proxy='address' userid=userid
pass='password';
```

An approach I used in testing this access method on a Unix system was to read my userid and password information from the .netrc file, looking for a particular machine (in this example, called abc) to obtain the userid and password:

```
data _null_; infile '~/.netrc' length=l;
     input @; input @1 line $varying200. l;
     machine=scan(line,2);
     if machine='abc';
     userid=scan(line,4);
     pass=scan(line,6);
     call symput('myuserid',trim(userid));
     call symput('mypass',trim(pass));
     run;

filename xxx http
'http://www.ecb.int/home/eurofxref.htm'
    user=&myuserid. pass="&mypass."
    proxy='<our proxy machine name>';
```

```
data _null_;
     call symput('myuserid',' ');
     call symput('mypass',' ');
     run;
```

This SAS code allows me to use my real userid and password but without having to write it in a SAS program. Note that as soon as the FILENAME statement is processed, I reset those macro variables to blanks to avoid exposure of the values.

Note that if you are running a version of the SAS System prior to Version 8, and you don't have access to the HTTP access method, an alternative may be to use the lynx command. The lynx command is vailable on a variety of platforms. See the official lynx web site at http://lynx.browser.org. The FILENAME statement for lynx would incorporate the use of the PIPE method:

filename xxx pipe 'lynx -source http://www.ecb.int/home/eurofxref.htm';

The SAS code to read from this fileref is exactly the same as that from the fileref using the HTTP access method.

Here's the SAS code to read the HTML tables as they currently exist at the ECB web site. The SAS code also creates the EURFRTBL table that the euro IFF set will need, as well as %LET statements for the macro symbol version of these rates.

```
      /*-------------------------------------------------------------------*/
      /* Read in the currency abbreviations for the currencies that are    */
      /* recognized by the euro IFF set. The first 11 currencies are those */
      /* with the 01JAN1999 irrevocable rates                              */
      /*-------------------------------------------------------------------*/

     data curr;
          length abbr $3;
          input abbr $ @@;
          first11=_n_<=11;
          cards;
     ATS BEF FIM FRF DEM IEP ITL LUF NLG PTE ESP CHF DKK GBP GRD SEK CZK HUF NOK RUR TRL PLZ ROL
     YUD SIT
     ;
     run;
     proc sort data=curr; by abbr;

      /*-------------------------------------------------------------------*/
      /* Read the rates from the ECB web site.                             */
      /*-------------------------------------------------------------------*/

     data rates(keep=abbr desc rate);
          infile xxx length=l;
          retain part 0;
          length desc $40 abbr $3;
          retain abbr desc;

          /*-------------------------------------------------------------------*/
          /* Read the line of HTML. The HTML lines at the time of this         */
          /* implementation had the 0x0d 0x0a carriage-return / line-feed in   */
          /* MS-DOS style. On non-Windows systems, the 0x0d will remain as a   */
          /* data character, so we need to remove the character. Also we       */
          /* create an upcased version of the line so we can look at HTML tags */
          /* with case-insensitivity.                                          */
          /*-------------------------------------------------------------------*/
```

```
input @; input @1 line $varying200. l;
line=left(compress(line,'0d'x));
uline=upcase(line);


/*-------------------------------------------------------------------*/
/* We will effectively ignore all HTML text until the TABLE tag is   */
/* seen. our 'table' variable indicates we've seen this tag.         */
/*-------------------------------------------------------------------*/

if uline=:'<TABLE' then do;
   table=1;
   retain table;
   return;
   end;
if table;

/*-------------------------------------------------------------------*/
/* At this point we'll be reading from the HTML table of rates. We   */
/* will traverse through every token on the line. Tokens are         */
/* separated by < and >, so HTML tags and the data are separate. We  */
/* are only interested in the data between the <TD> </TD> tags.      */
/* We are interested in the 3 data items that appear in the TD       */
/* parts. Note an example of the section:                            */
/* <tr bgcolor="#fff7cc">                                            */
/* <td align="center">USD</td>                                       */
/* <td>US dollar</td>                                                */
/* <td align="right">0.8480</td>                                     */
/* <td bgcolor="#ffffff"></td>                                       */
/* <td align="center">LVL</td>                                       */
/* <td>Latvian lat</td>                                              */
/* <td align="right">0.5377</td>                                     */
/* </tr>                                                             */
/* We see both the USD and LVL currencies defined in this row.       */
/*-------------------------------------------------------------------*/

td=0;
do i=1 to 100;     /* do while(1); is more appropriate but dangerous */
   piece=left(scan(uline,i,'<>'));
   if piece='/TABLE'
      then stop;
   if piece=' '
      then leave;
   if piece=:'TD ' then do;
      td=1;
      end;
   else if piece='/TD' then td=0;
   else if td then do;
      part=mod(part,3);
      if part=0 then do;
         abbr=piece;
         end;
      if part=1 then do;
         desc=left(scan(line,i,'<>')); /* get original casing */
         end;
      else if part=2 then do;
```

```
            rate=input(piece,best12.);
            output;
            end;
         part+1;
         end;
      end;
   run;
proc sort data=rates; by abbr;

 /*-------------------------------------------------------------------------*/
 /* This macro performs conversions from all desired currencies to euro,  */
 /* using the builtin rates or the EURFRTBL rates if they are available.  */
 /*-------------------------------------------------------------------------*/

%macro doconv(iter);
data conv&iter.; set curr(where=(first11=0));
   value&iter.=eurocurr(1,abbr,'EUR');
   run;
%mend;


 /*-----iteration 1: using builtin rates-----*/

%doconv(1);

 /*-------------------------------------------------------------------------*/
 /* Create the EURFRTBL table using the rates we obtained from the ECB    */
 /* web site. We only emit the rates for the currencies that we are       */
 /* interested in. Be sure to add the greek drachma, now with an          */
 /* irrevocable rate. Other adoptive currencies would be added in this    */
 /* way, since they rates will not be in the ECB table.                   */
 /*-------------------------------------------------------------------------*/

filename eurfrtbl temp;

data curr;
   file eurfrtbl;
   length abbr $3;
   merge rates(in=have) curr(in=want where=(first11=0)); by abbr;
   if _n_=1 then put 'EURFRGRD=340.750'; /* Greek drachma */
   if want then do;
      newrate=have;
      output curr;
      end;
   if want and have;
   put @1 'EURFR' abbr $char3. '='  rate;
   run;

data _null_; infile eurfrtbl; input; put _infile_; run;

 /*-----iteration 2: using eurfrtbl rates-----*/
%doconv(2);

 /*-------------------------------------------------------------------------*/
 /* Now merge the different converted values (builtin vs. current rate)   */
 /* to see how they compare.                                              */
 /*-------------------------------------------------------------------------*/
```

```
data _null_; merge conv1 conv2; by abbr;
     put abbr= value1= value2=;
     run;
```

Here is what the EURFRTBL looks like, using May 30, 2001 rates:

EURFRGRD=340.750
EURFRCHF=1.5206
EURFRCZK=34.195
EURFRDKK=7.4556
EURFRGBP=0.5973
EURFRHUF=253.75
EURFRNOK=7.93
EURFRROL=24392
EURFRSEK=9.125
EURFRSIT=217.6353
EURFRTRL=995000

Here is the output from the final merged DATA step:

```
abbr=CHF value1=0.6233248146 value2=0.657635144
abbr=CZK value1=0.0286892183 value2=0.0292440415
abbr=DKK value1=0.1335097442 value2=0.1341273673
abbr=GBP value1=1.4283020916 value2=1.6742005692
abbr=GRD value1=0.0029335406 value2=0.0029347029
abbr=HUF value1=0.0038413522 value2=0.0039408867
abbr=NOK value1=0.1087228329 value2=0.1261034048
abbr=PLZ value1=0.2380952381 value2=0.2380952381
abbr=ROL value1=0.0729394602 value2=0.000040997
abbr=RUR value1=0.050586807  value2=0.050586807
abbr=SEK value1=0.106770191  value2=0.1095890411
abbr=SIT value1=0.0052356021 value2=0.0045948428
abbr=TRL value1=0.0029681341 value2=1.0050251E-6
abbr=YUD value1=0.0765438903 value2=0.0765438903
```

The values for PLZ, RUR, and YUD did not change since they don't appear in the ECB table. Some (TRL and ROL in particular) have changed drastically due to market fluctuations since the builtin table was created.

USING PICTURE AND MULTIPLIERS FOR THE RATES

Although the US dollar (USD) symbol is not part of the euro IFF set, it is set by our code above. If we want to convert between the USD and euro, the US dollar, we can incorporate it into a MULT= option of a PICTURE format. However, you have to be aware of what the MULT= does with respect to a decimal point. Consider the following SAS code:

```
 /* Create a macro variable for the US
    Dollar rate */

data _null_; set rates;
     if abbr='USD';
     call symput('EURFRUSD',
           trim(
            left(put(rate,best12.))));
     run;


 /* Create a picture format using this rate
as the multiplier */

proc format;
     picture usd other='000,000,009.99'
     (prefix='$' mult=&eurfrusd.);
     run;


 /* Try this format with $1 to see the
result as the number of euros.
    It will be wrong. */

data _null_; x=1; put x=usd.; run;
```

Indeed, the result is

```
x=$0.00
```

We don't get what we expect! This is because without a MULT= option, picture processing will multiply by 10**n, where n is the number of places to the right of the decimal point. But if a MULT= option is given, as it is here, no accommodation for the decimal point takes place, and it's assumed that the number has already been multiplied by the number of decimal places, or that the MULT= value factors this in.

```
/* Now try the following, which does work */

data _null_; x=1; x=x*100; put x=usd.; run;
```

This results in

```
x=$0.84
```

Meaning that 1 euro is worth .84 US dollars.

```
/* Instead create with the multiplier
included */

data _null_; set rates;
     if abbr='USD';
     call symput('EURFRUSD',
           trim(
            left(
              put(rate*100,best12.))));
     run;
proc format;
     picture usd other='000,000,009.99'
     (prefix='$' mult=&eurfrusd.);

/* This time the answer is as expected */

data _null_; x=1; put x=usd.; run;
```

And it is also generated as

```
x=$0.84
```

BEING AWARE OF OVERUSE OF WEB SITES

This SAS program will read the ECB site for rates. Please be aware that this site is accessed by users all over the world, and you should not access it more often than necessary. The values are changed each business day, around 1:30pm GMT. If you need to have access to these values, it is probably best to make a local copy of the values and then access that local copy throughout the day. Otherwise the CEB web site may be overwhelmed by accesses.

A SAS© Approach to WEB-Based Surveys

Bernard R. Poisson, Statprobe Technologies

Abstract

The United State Air Force had a problem. Its existing methods of collecting information from its people were time-consuming, error-prone, and costly. Not any more! The Survey Engine, based on SAS©/IntrNet solved those problems once and for all! The Survey Engine provides the ability to host any number of Internet/Intranet surveys concurrently without in-depth computer or Internet knowledge or experience. All questions and response-sets are stored in SAS© data sets, independent from any display characteristics, giving them greater flexibility and reusability. This application provides a powerful, efficient, and cost-effective approach to gathering data from large groups of individuals.

Introduction

The U.S. Air Force Surveys Branch performs more than 3 dozen large-scale surveys on an annual basis, polling members to obtain their impressions on a wide variety of topics such as adequacy of compensation (pay, living conditions, etc), job satisfaction (having proper equipment, having adequate spare parts, adequate and safe working conditions), and others. Two principal goals of the data gathering efforts focus on determining what AF members believe senior leadership must do to make the USAF desirable enough to retain existing members and to attract high-school and college graduates as new members. The global nature of the USAF makes sampling of the total force a very difficult task.

In the not so distant past, surveys were administered using question booklets and scan-sheets. With the advent of increased software capabilities over the World Wide Web (WWW), the survey analysts assigned to the AF Surveys Branch conducted periodic reviews over the WWW of available survey applications on a routine basis in search of a better way of fulfilling their mission. Each application was thoroughly reviewed to ascertain if it was capable of coping with the rigorous demands placed upon it by professional survey construction and administration requirements. Although some applications possessed certain capabilities that made them an interesting possibility, they were also were found to contain critical flaws that would render them practically unusable in the world of professional survey construction and administration. The primary capability lacking in most of these applications was inability to control program and question flow based on the respondent's responses. After several years of searching, the survey analysts and senior leadership concluded the most plausible solution was to obtain a military programmer/analyst with sufficient experience and knowledge to design and create the required application for them.

The resulting SAS©/IntrNet based survey application, the Survey Engine, written in SCL, has replaced all previous manual methods of data collection. Using this application, the operational efficiency of the AF Surveys Branch has increased by over 55 percent through decreased fielding and turn-around times. Operational costs of survey administration have been reduced by over 75%. More importantly, the survey experience leaves the respondent with a high degree of satisfaction through reduced frustration and time.

This paper will explain how this application came about and demonstrate how its use can help your organization, regardless of size, diversity, and geographic location.

Determining Program Requirements

In order to establish an organized idea of what the survey application was required to do, a list of program requirements was devised. These requirements came about through a series of many short informal meetings held between the two senior survey analysts and the programmer. The survey analysts brought 40+ years of survey experience and the programmer 18+ years of systems and programming knowledge to the table. Together, they established the requirements listed in Figure 1.

Questions must be reusable
Responses must be reusable
Responses must be independent of display characteristics
Additional Instructions capability needed
Response Format Options (Output)
Radio-Buttons (mutually exclusive)
Check-Boxes (mark all that apply)
Drop-Down List (single selection)
Drop-Down List (multiple selection)
Text-Box (single numeric value entry)
Free-Form Text field
Radio-Buttons with Large Response characteristics
Check-Boxes with Large Response characteristics
Check-Boxes with Large Response characteristics with branching
Multiple Drop-Down 'Rating' Lists
Multiple Drop-Down 'Rating' Lists with branching
Text-Box List (single numeric value entry for each)
Text-Box List (single numeric value entry for each) with branching
Responses must be validated to: disallow skipping of questions, ensure entry of values within allowable range, and to provide feedback to the respondent.
Respondent must be able to go back to previous questions, in case of errors or desire to change responses based on new questions
Respondent must be able to Stop & Resume a survey at a later time, continuing from the place where they left off.
Analysts must be able to 'track' respondents for demographic purposes and longitudinal/historical review
Must provide access control by restricting entry into surveys (limit entry to USAF Personnel, sample set, etc. only)
Prevent previous respondents from re-submitting multiple surveys
Follow-On Question and/or Skip-Logic required, based on respondent selection(s) to responses.
Must be able to administer multiple surveys concurrently
Must be coded so that future code maintenance is NOT required

Figure 1 – Program Requirements

Access Control & User Validation

An integral consideration of any survey, especially surveys administered over the WWW is that of access control. Sampling is used to derive a mathematically representative group of respondents, taking various strata into consideration. In order to maintain some sense of order, and to prevent the possible skewing of results, access to a survey must be restricted to only those individuals who fall within the sample group. This capability is provided through a small, independent module named CHKUSER.SCL, operating as a front-end to the Survey Engine. When a respondent attempts to enter a survey, they are greeted by the screen shown in **Figure 2**.

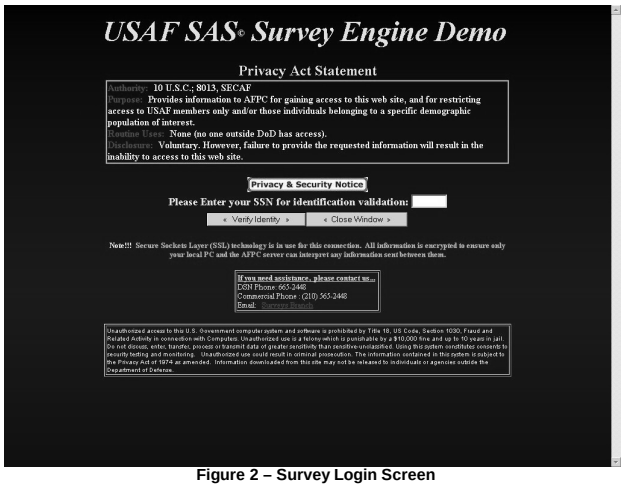


Figure 2 – Survey Login Screen

There are two distinct ways that a respondent may access a survey. These are:

- **A URL to a given survey such as: http://surveys.af.mil/srvyonl/login/demo.htm**
- **A URL + PID (Personal ID) value such as: http://surveys.af.mil/srvyonl/login/demo.htm?5349234234**

If the URL provided by the individual does NOT contain a PID value appended to the query-string, the individual **must** manually provide a valid Social Security Number (SSN) that is contained within the sample group of the survey. If the URL provided **does** contain a valid PID value appended to the query-string, JavaScript embedded within the login screen parses the value from the query-string and performs an 'auto submit' thereby effectively bypassing the individual's need to manually enter a valid SSN. In either instance, a look-up is performed to determine if a valid value was provided for authentication into the survey. If either of the values is not valid, the individual is redirected to a screen indicating an error condition. If the provided value **is** valid, the Survey Engine is invoked and survey administration begins.

Presenting Survey Information to the Respondent

The Survey Engine uses a standardized layout throughout the course of a survey to present the information to the respondent as shown in **Figure 3**.

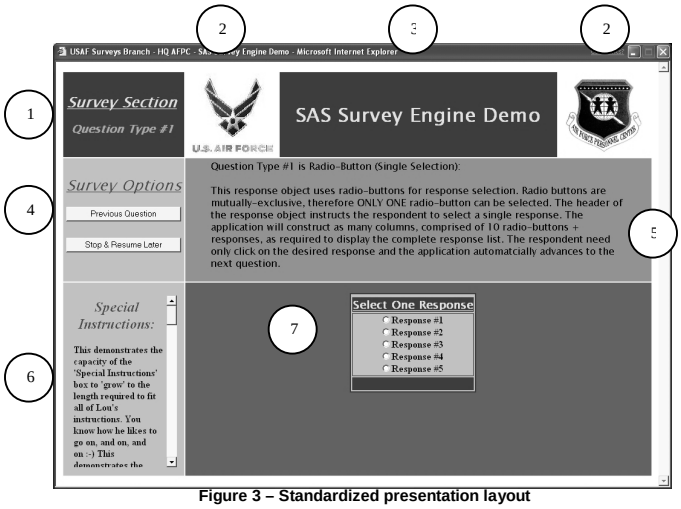


Figure 3 – Standardized presentation layout

Object 1: Section/sub-section designator. Allows surveys covering multiple topics to be broken up into multiple 'logical' sub-sections such as finances, job satisfaction, living conditions, career intent, etc.

Object 2: Graphic Banners: Allows customization of the output display.

Object 3: Survey Title: A data object containing the survey title.

Object 4: Survey Options: Control objects providing 'page back' and 'Stop and Resume' capabilities.

Object 5: Question Section: A data object containing the question to be asked.

Object 6: Special Instructions: A data object allowing the inclusion of additional information to be presented to the respondent to clarify a question, response, etc.

Object 7: Response Section: A data object containing the list of acceptable answers for the question presented.

All objects depicted above are static in that they do not change form or structure except for the Response Object (7). The response object can be dynamically represented in any one of 10 data-independent formats, depending on survey requirements.

The Response Object Formats:

The response object formats are described in detail below:

The Radio-Button Response Format (Single Selection)
This response object uses radio-buttons for response selection. Radio buttons are mutually exclusive; therefore **only one** radio-button can be selected. The header of the response object instructs the respondent to 'Select One Response'. The application constructs as many columns, comprised of 10 radio-buttons + responses, as required to display the complete response list. The respondent need only click on the desired response and the application automatically advances to the next question. (**Figure 4)**.

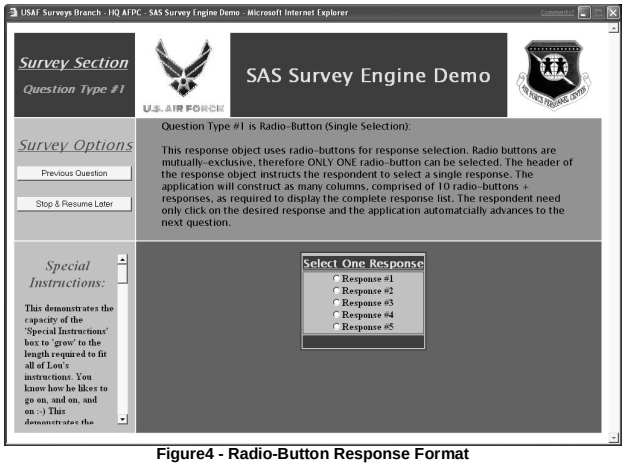


Figure4 - Radio-Button Response Format

The Check-Box Response Format (Multiple Selections)
This response object uses check-boxes for 'Mark All That Apply' response selection(s). This implementation allows the selection of **some** or **all** responses, **but not none**. The header of the response object instructs the respondent to 'Check All That Apply'. The application constructs as many columns, comprised of 10 check-boxes + responses, as required to display the complete response list. Since more than one response can be selected, the respondent must click the 'Submit' button for the application to advance to the next question. **(Figure 5)**.

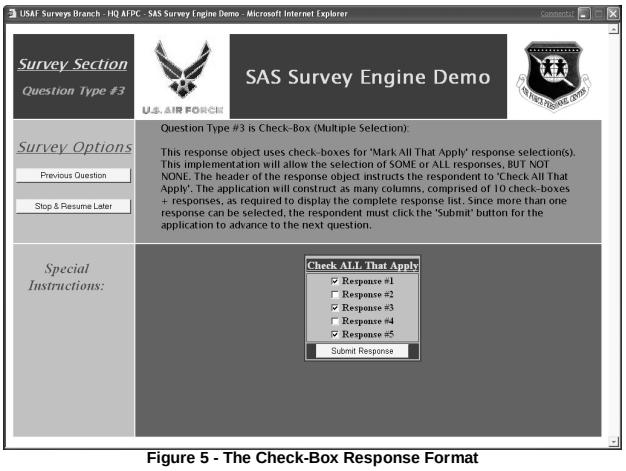


Figure 5 - The Check-Box Response Format

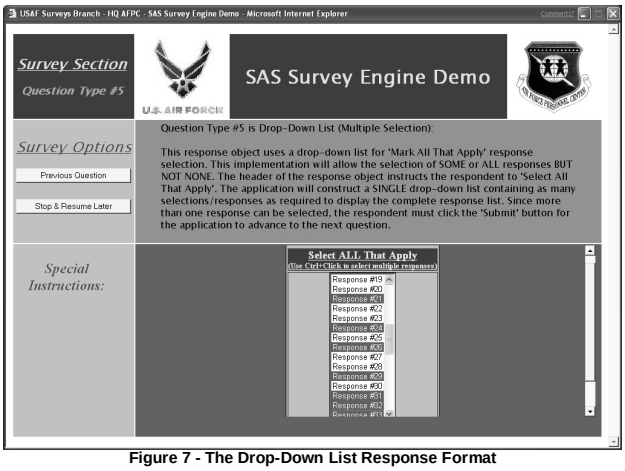


Figure 7 - The Drop-Down List Response Format

The Drop-Down List Response Format (Single Selection)
This response object uses a single Drop-Down List for response selection. This implementation of the Drop-Down List is mutually exclusive; therefore ONLY ONE item within the Drop-Down List can be selected. The header of the response object instructs the respondent to 'Select One Response'. The application constructs a SINGLE drop-down list containing as many selections/responses as required to display the complete response list. The respondent need only click on the desired response and the application automatically advances to the next question. **(Figure 6)**

The Text Value Entry Response Format (Variable Length)
This response object uses a variable-length text-box for the entry of a numeric value entry such as 'Number of days TDY'. The header of the response object displays the range of acceptable values. The entry field is dynamically sized based on the number of characters required to enter the maximum value allowed for the particular response. Dynamically embedded JavaScript validates the value entered and prompts the user with an error/correction dialog if the value entered is not within the allowable range. The respondent must click the 'Submit' button for the application to advance to the next question. **(Figure 8)**

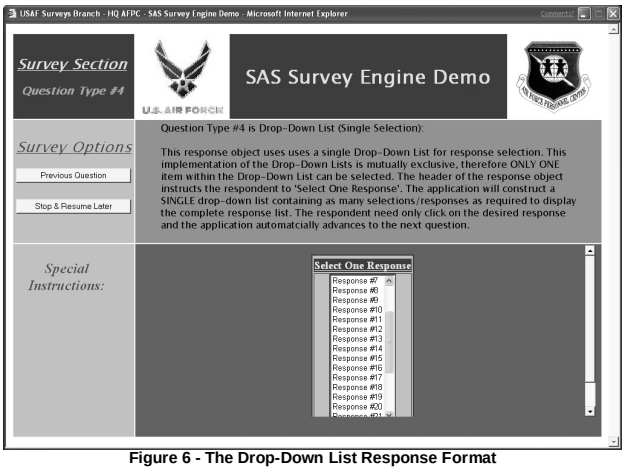


Figure 6 - The Drop-Down List Response Format

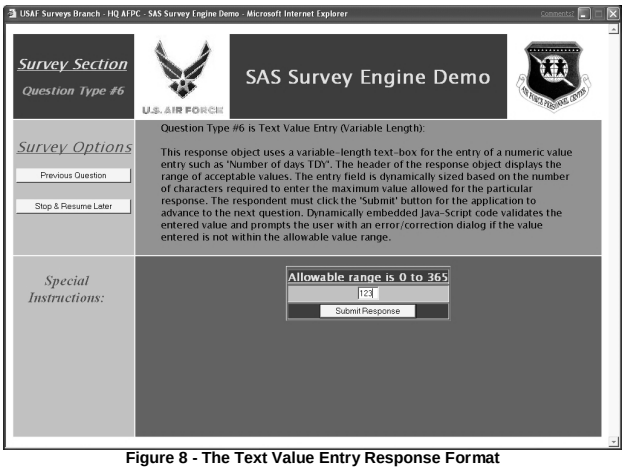


Figure 8 - The Text Value Entry Response Format

The Drop-Down List Response Format (Multiple Selections)
This response object uses a drop-down list for 'Mark All That Apply' response selection. This implementation of the Drop-Down List allows the selection of **some** or **all** responses, **but not none**. The header of the response object instructs the respondent to 'Select All That Apply'. The application constructs a SINGLE drop-down list containing as many selections/responses as required to display the complete response list. Since more than one response can be selected, the respondent must click the 'Submit' button for the application to advance to the next question. **(Figure 7)**

The Free-Form Text Entry Response This response object allows the entry of FREE-FORM text. Dynamically embedded JavaScript code monitors the respondent's progress and continuously displays '##### Characters Remain' in the response object header as the respondent types. The JavaScript code validates the number of characters typed and prompts the user with an error/correction dialog if the number of characters typed exceeds the allowable range. The respondent must click the 'Submit' button for the application to advance to the next question. **(Figure 9)**

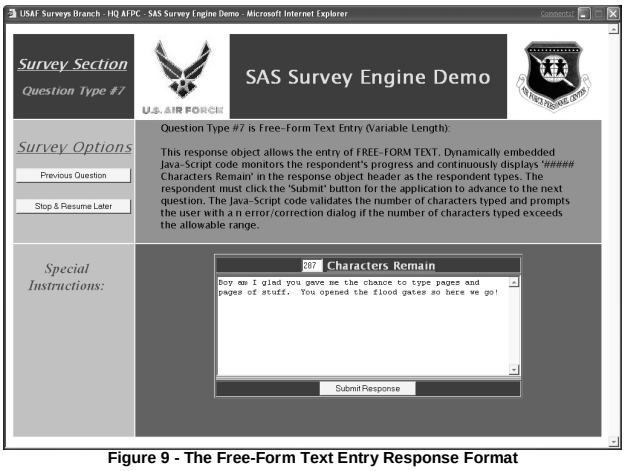


Figure 9 - The Free-Form Text Entry Response Format

The Radio-Button Response Format (Single Selection + Large Response) This response object uses radio-buttons for response selection. Radio buttons are mutually exclusive; therefore ONLY ONE radio-button can be selected. The header of the response object instructs the respondent to 'Select One Response'. The application constructs a single column containing as many entries as required (with scroll controls) to display the complete response list. The respondent need only click on the response and the application automatically advances to the next question. **(Figure 10)**

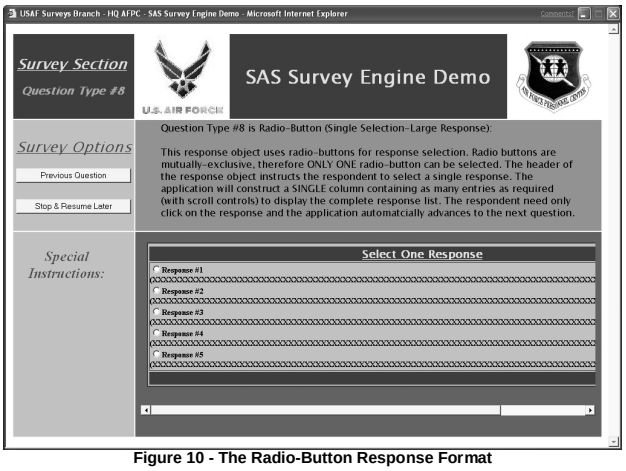


Figure 10 - The Radio-Button Response Format

The Check-Box Response Format (Multiple Selection + Large Response) This response object uses check-boxes for response selection. This implementation allows the selection of **some** or **all** responses **but not none**. The header of the response object instructs the respondent to 'Check All That Apply'. The application constructs a single column containing as many entries as required (with scroll controls) to display the complete response list. Since more than one response can be selected, the respondent must click the 'Submit' button for the application to advance to the next question. **(Figure 11)**

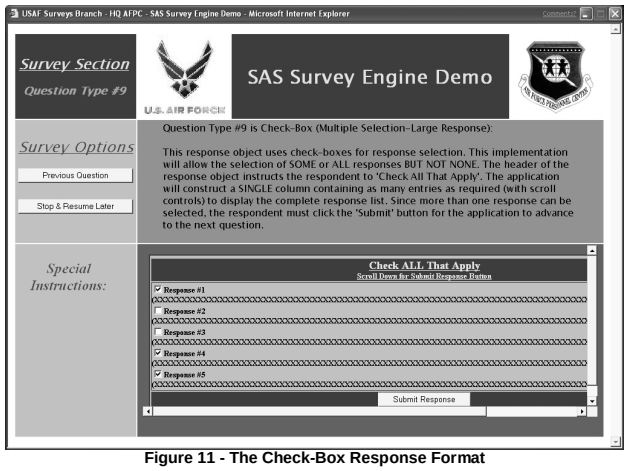


Figure 11 - The Check-Box Response Format

The Check-Box Response Format (Multiple Selection + Large Response + Branching) This response object uses check-boxes for response selection and uses the respondent's selection(s) to control program flow through the survey. This implementation allows the selection of SOME or ALL responses but **not** none. The header of the response object instructs the respondent to 'Check All That Apply'. **Each response option** in the list is capable of branching to a **different/separate** 'Follow-On' question. Sequential program flow continues once all follow-on options have been completed. Since more than one response can be selected, the respondent must click the 'Submit' button for the application to advance to the next question. **(Figure 12)**

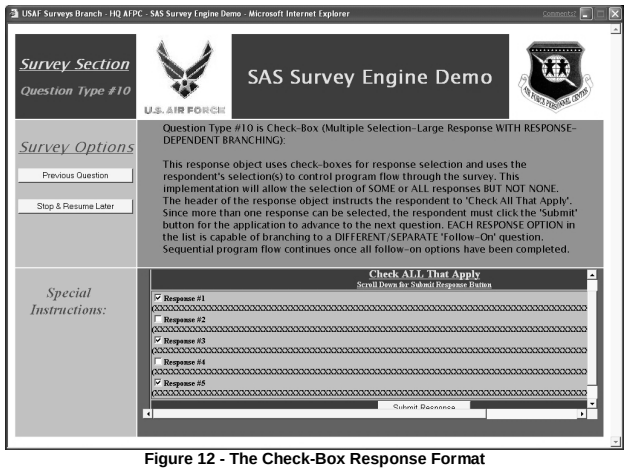


Figure 12 - The Check-Box Response Format

The Drop-Down 'Rating' list Response Format

This response object provides drop-down list containing pre-defined 'ratings/rankings' for the respondent to select from. Each response option 'Rated' by the respondent represents Selection+Rating. This implementation allows the selection of **some, all,** or **none** of the response objects. The header of the response object instructs the respondent to 'Rate Applicable Items'. The application constructs a single column of text items containing as many entries as required (with scroll controls) to display the complete response list. Since more than one response can be selected, the respondent must click the 'Submit' button for the application to advance to the next question. **(Figure 13)**

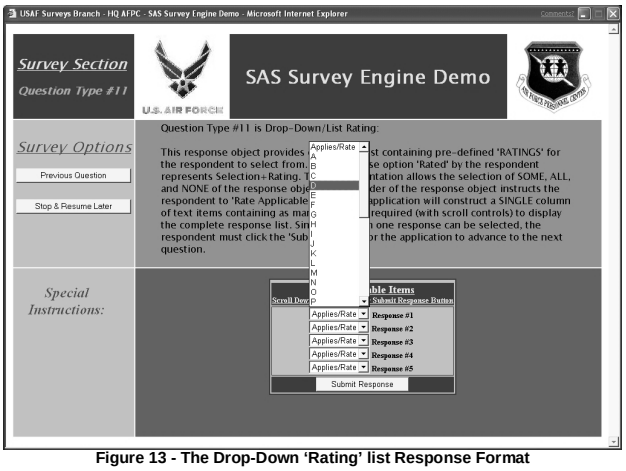


Figure 13 - The Drop-Down 'Rating' list Response Format

The Drop-Down 'Rating' list Response Format (+ Branching)
This response object provides drop-down list containing pre-defined 'ratings/rankings' for the respondent to select from. Each response option 'Rated' by the respondent represents Selection+Rating. This implementation allows the selection of **some, all,** or **none** of the response objects. The header of the response object instructs the respondent to 'Rate One or More response(s)'. The application constructs a SINGLE column of text items containing as many entries as required (with scroll controls) to display the complete response list. **Each response option** in the list is capable of branching to a **different/separate** 'Follow-On' question. Sequential program flow continues once all follow-on options have been completed. Since more than one response can be selected, the respondent must click the 'Submit' button for the application to advance to the next question. **(Figure 14)**

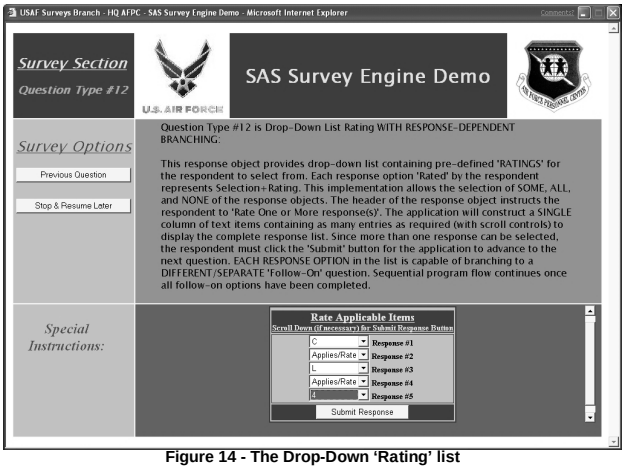


Figure 14 - The Drop-Down 'Rating' list

The Text Value Entry List Response Format (Variable Length)
This response object uses variable-length text-boxes for the entry of a list of numeric value entries such as 'Number of days TDY'. The header of the response object displays the range of acceptable values. The entry fields are dynamically sized based on the number of characters required to enter the maximum value allowed for each particular response. Dynamically embedded JavaScript code validates the entered values and prompts the user with an error/correction dialog if any of the values entered are not within the allowable value range. Since more than one response can be selected, the respondent must click the 'Submit' button for the application to advance to the next question. **(Figure 15)**

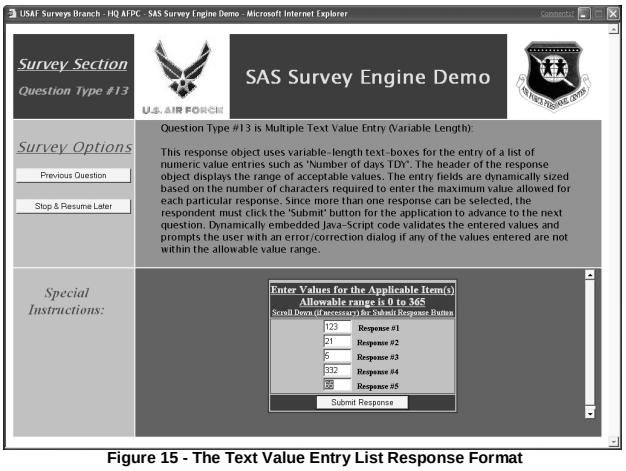


Figure 15 - The Text Value Entry List Response Format

The Text Value Entry List Response Format (+ Branching)
This response object uses variable-length text-boxes for the entry of a list of numeric value entries such as 'Number of days TDY'. The header of the response object displays the range of acceptable values. The entry fields are dynamically sized based on the number of characters required to enter the maximum value allowed for each particular response. Dynamically embedded JavaScript code validates the entered values and prompts the user with an error/correction dialog if any of the values entered are not within the allowable value range. EACH NUMERIC ENTRY in the list is capable of branching to a **different/separate** 'Follow-On' question. Sequential program flow continues once all follow-on options have been completed. Since more than one response can be selected, the respondent must click the 'Submit' button for the application to advance to the next question. **(Figure 16)**

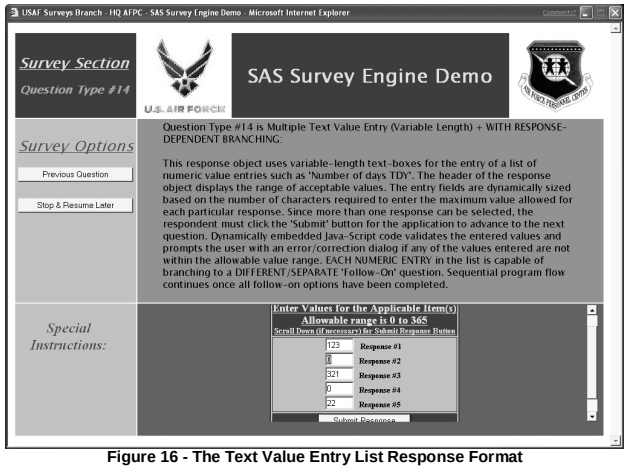


Figure 16 - The Text Value Entry List Response Format

The Survey Completion/Termination Screen:
Upon completion of a survey, the respondent is presented with a completion/termination screen explaining what will be done with the data collected from them during the course of the survey. At this point the survey has been terminated and all data recorded. The respondent is provided a 'Close Window' button to facilitate closing of the survey browser interface. An example screen is shown in **Figure 17**.

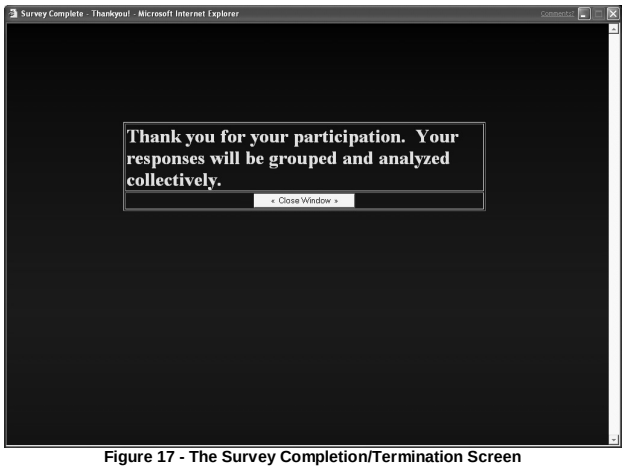


Figure 17 - The Survey Completion/Termination Screen

Behind The Screens – Making it all come together

Creating a survey engine capable of displaying any response using any one of several response formats came about from the fact that although the available topic matter of surveys may change and is unlimited, the ways in which responses can be successfully presented and formatted to create a workable survey is finite. Taking these factors into consideration, we created an adequate set of response object format styles to meet any future needs of the survey analyst. Taking advantage of the capabilities of today's web browsers, we can build powerful and truly dynamic surveys that can address any topic matter.

Data Sources

In order to have a truly dynamic application, one capable of literally changing with the needs of the organization, the application must obtain as much of its required operational information from the most dynamic of sources available to the computer. The most dynamic sources of data of course are data files. The survey engine acquires 99% of its operational data from four separate SAS© data sets. These are the QuestionsDB, ResponseDB, JumpTableDB, and AppCFG data sets. Their functions are as follows:

QuestionsDB: This local data set is where the survey engine obtains most the operational information required for a particular survey and is a subset of the Master Questions Database. The data obtained from this data set is comprised of the Question Text, Response Code, Response Format, Response Range, Skip-Logic Flag, Section Text, and Instruction Text. Using a combination of these elements, the application has the information it requires to dynamically generate a particular survey.

ResponseDB: This global data set is where the survey engine obtains ALL of the information required relating to ALL response-sets for ALL surveys. The data obtained from this data set is comprised of the Response Name and Individual Responses that comprise the response.

JumpTableDB: This local data set may or may not exist for a particular survey. If a survey does not require any response-dependent branching (skip-logic), then this file will not exist for the given survey.

AppCfg: This global data set is the 'master configuration file' used to prevent the need of maintenance on the survey engine's code. The data obtained from this data set is comprised various operational parameters such as libnames for the Master QuestionsDB, ResponseDB and other data sets, WEB server name, path to HTML information screens, foreground/background colors to be used when generating the surveys, etc. The libname 'APPCFG' is the ONLY EXTERNAL item of information that the Survey Engine requires and is dependent upon for its successful operation. Armed with this information, the Survey Engine can obtain all other required operational parameters from the AppCfg data set.

JumpTable.slist: This is a dynamically created SAS© SCL list. If/When a survey contains a response object utilizing response-dependent branching; this list will be created on the fly and saved to disk in order to perform correct question sequencing. Branching represents a deviation from the normal sequential flow from one question to another. The existence of this list indicates to the survey engine that it must anticipate this deviation for this particular respondent. Additionally, the engine uses the contents of this file to repopulate the SCL list, which contains the question numbers represented in the deviation from normal flow through the next 'logical' question.

ResponseHistory.slist: This is also a dynamically created SAS© SCL list. The intent of this list is to create an audit-trail of each respondent's path through a survey. As the respondent proceeds through a survey, the number of each question referenced is appended to the existing list and the list saved to disk for future reference thereby creating a complete transaction history. Its intended purpose is to provide an EXACT reverse-path through questions visited by the respondent so that if/when the respondent selects the 'Previous Question' button one or more times, the Survey Engine will be able to correctly display the desired question along with the respondent's response.

The Survey Engine Module and its Routines

The Survey Engine's principal module is comprised of 13 separate CONTROL sections, and 16 FORMATTING sections each constructed as subroutines. Due to the reentrant nature of this application, most variables used are global and are not unique to a particular section of code. Each section of code was given a 'meaningful' name which clearly describes the functionality of the code contained within it.

The CONTROL sections are:

INIT
FIND_NEXT_QUESTION_NUMBER
GET_PREVIOUS QUESTION
SAVE_RESPONSE
SAVE_HISTORY
SET_LOCKOUT
GET_PREVIOUS_RESPONSE
DISPLAY_NEXT_QUESTION
DISPLAY_MESSAGE
SET_LOCKOUT
WEBOPEN
WEBCLOSE
DEBUG
TERM

The formatting sections are:

FORMAT_RB
FORMAT_CM
FORMAT_DS
FORMAT_DM
FORMAT_TX
FORMAT_TF
FORMAT_RL
FORMAT_CL

FORMAT_DL
FORMAT_TL
ADD_CONTROL_PARAMETERS_FRAME
ADD_CONTROL_PARAMETERS_NOFRAME
IFRAME_RESPONSE_OPEN
IFRAME_RESPONSE_CLOSE
INNER_TABLE_OPEN
INNER_TABLE_CLOSE

Walking Through the Survey Process – The Operational Cycle

We will now step through the typical processing cycle of the Survey Engine and briefly describe the various processes involved within each of the CONTROL and FORMATTING sections encountered during the run. Please note that there are NO 'hard-coded' values incorporated into the Survey Engine code itself. Any required parameters and values are provided by the AppCfg data set and will be discussed in greater detail below. Pseudo-code will be used to simplify explanation and understanding of code functions and provide brevity.

Once validated into the system via the ChkUser module, a call is made to the Survey Engine's main processing module, POSTRESP.SCL. Respondent-dependent data values such as SURVEY_NAME, PID, PREVIOUS_QUESTION, NEXT_QUESTION and the like are passed to POSTRESP via an SCL list, PARAMS, as part of the program call.

INIT Section: This section is **unconditionally** executed and performs the following:

DEFINE required variable types and lengths
RETRIEVE data values from the PARAMS list and set like-named variables
INITIALIZE control variable values
RETRIEVE control variable values from the AppCfg data set and set like-named variables
EXECUTE 'LINK' calls to the other sub-routine sections contained within POSTRESP

The sub-routines are called by the INIT section in the following order:

GET_PREVIOUS_QUESTION
FIND_NEXT_QUESTION_NUMBER
GET_PREVIOUS_RESPONSE
SAVE_HISTORY
SAVE_RESPONSE
DISPLAY_NEXT_QUESTION
RETURN

The remaining sub-routines within the POSTRESP.SCL module are called by these sections, as required, to satisfy the balance of control and function of the sections listed above.

GET_PREVIOUS_QUESTION Section: The function of this sub-routine is to replace/bypass the value provided by the FIND_NEXT_QUESTION section and enables the Survey Engine to logically 'reverse' its normal program-flow and revisit questions previously displayed to the respondent. This section is **conditionally** executed and is called ONLY when the PAGE BACK flag is set. This section is 'triggered' by the respondent's selection of the 'Previous Question' button on the survey screen and performs the following:

Create SESSION_HISTORY_LIST (SCL list)
IF SESSION_HISTORY_LIST.SLIST exists
    LOAD values from .SLIST entry identified by user's PID and SURVEY NAME
IF SESSION_HISTORY_LIST length > 0 THEN
    SET NEXT_QUESTION to value of last item in SESSION_HISTORY_LIST
    DELETE last item from SESSION_HISTORY_LIST
    SAVE updated SESSION_HISTORY_LIST.SLIST
ELSE
    SET NEXT_QUESTION to 1
RETURN

FIND_NEXT_QUESTION Section: This section determines the next question to be displayed to the respondent. This section is **conditionally** executed and is executed only when the PAGE BACK flag is **not** set. The determination is made using dynamically provided values and is the single most complicated section of code within the POSTRESP module. This section performs the following:

JT is a flag indicating that skip-logic will be used.

MATA is a flag indicating that the question will use 'mark all that apply' functionality.

RF is a flag indicating the output response format to be used in displaying the response object to the respondent.

IF RESPONSE_INDEX > 0 and JT='Y' and MATA='Y' THEN
    Open JUMPTABLE file
    Set WHERE clause to value of PREVIOUS_QUESTION
    Create JUMP_TABLE_LIST (SCL List)
    Insert PREVIOUS_QUESTION value into JUMP_TABLE_LIST
    LOOP
    IF Nth response is selected
        Insert nth value from JUMPTABLE data set based on numeric value of the respondent's response

    END LOOP
    Insert PREVIOUS_QUESTION value +1 into JUMP_TABLE_LIST
    SAVE JUMP_TABLE_LIST.SLIST

IF RESPONSE_INDEX > 0 and JT='Y' and MATA='Y' AND RF='DL' OR RF=TL' THEN
    Open JUMPTABLE file
    Set WHERE clause to value of **Nth sub-item** of PREVIOUS QUESTION
    Create JUMP_TABLE_LIST (SCL List)
    Insert PREVIOUS_QUESTION value into JUMP_TABLE_LIST
    LOOP
    IF Nth response of **Nth sub-item** is selected
        Insert nth value from JUMPTABLE data set based on numeric value of the respondent's response

    END LOOP
    Insert PREVIOUS_QUESTION value +1 into JUMP_TABLE_LIST

IF JUMP_TABLE_LIST**.SLIST** EXISTS THEN
    Create JUMP_TABLE_LIST (SCL List)
    Load JUMP_TABLE_LIST with values contained in JUMP_TABLE_LIST**.SLIST**

    JUMP_TABLE_LIST_LENGTH = length of JUMP_TABLE_LIST
    FIRST=value of 1st entry in JUMP_TABLE_LIST
    LAST=value of last entry in JUMP_TABLE_LIST

Look for PREVIOUS_QUESTION in JUMP_TABLE_LIST

If position returned <= JUMP_TABLE_LIST_LENGTH - 1 then
NEXT_QUESTION=value of position+1 in JUMP_TABLE_LIST
If position returned = 0 then
NEXT_QUESTION=NEXT_QUESTION+1

If position returned <= JUMP_TABLE_LIST_LENGTH then
NEXT_QUESTION=value of position in JUMP_TABLE_LIST +1 (normal resume point)

IF RESPONSE_INDEX > 0 and JT='Y' and MATA='N' THEN
Open JUMPTABLE file and set WHERE clause to value of PREVIOUS QUESTION
Set NEXT_QUESTION to Nth value from JUMPTABLE data set based on numeric value of the respondent's response
ELSE
NEXT_QUESTION=PREVIOUS_QUESTION+1

RETURN

GET_PREVIOUS_RESPONSE Section: This section retrieves the respondent's response(s) to the **current** question from the RESPONSE data set. The purpose of this section is to enable the pre-select/re-selection of the respondent's responses (remember) AFTER they have selected the 'PREVIOUS QUESTION' button and revisited one or more questions. This section performs the following:

Open RESPONSE data set and set WHERE clause to respondent's PID value
Retrieve respondent's previous response from RESPONSE data set
IF response value = 0 then
PREVIOUS_RESPONSE=''
RETURN

SAVE_HISTORY Section: This section tracks a respondent's progress through a survey. The purpose of this section is to enable the respondent to navigate BACKWARDS through question(s) they have already responded to. This is accomplished by incrementally modifying an SCL list and saving the list to a .SCLIST entry for future reference. This section performs the following:

IF NEXT_QUESTION >1 THEN
Create SESSION_HISTORY_LIST (SCL List)
IF SESSION_HISTORY_LIST**.SLIST** exists
Load SESSION_HISTORY_LIST with values contained in SESSION_HISTORY_LIST**.SLIST**
IF SESSION_HISTORY_LIST length > 0 THEN
LAST_HISTORY_LIST_ITEM=value of last entry in SESSION_HISTORY_LIST
IF PAGE_BACK flag **not** set and PREVIOUS_QUESTION <> LAST_HISTORY_LIST_ITEM THEN
Insert value of PREVIOUS_QUESTION into last position of SESSION_HISTORY_LIST
Save SESSION_HISTORY_LIST to SESSION_HISTORY_LIST**.SLIST**
RETURN

SAVE_RESPONSE Section: This section records the respondent's response(s) to the current question to the RESPONSE data set. This section performs the following:

Open RESULTS data set
Set WHERE clause to respondent's observation
IF the question is a 'mark all that apply' type THEN
Initialize MATA_REPONSE variable to ''
LOOP
Set index to maximum number of possible responses
If Nth response is selected THEN
Concatenate current response value to MATA_RESPONSE
END LOOP
Save MATA_RESPONSE or RESPONSE to RESPONSE data set
Close RESULTS data set
RETURN

DISPLAY_NEXT_QUESTION Section: This section displays the 'next question' and dynamically generates and displays the response object to the respondent. This section performs the following:

QT contains the question-text obtained from the QUESTIONS data set
JT is a flag indicating that skip-logic will be used.

MATA is a flag indicating that the question will use 'mark all that apply' functionality.

RF contains the output response format to be used in displaying the response object to the respondent.

RC contains the NAME of the RESPONSE SET to be formatted and displayed to the user. Example: Agree007. Each response-set indicates the number of entries within it. In the example above, the response-set is an Agree/Disagree response-set and contains 7 entries/values to be displayed to the user.

Open QUESTIONS data set
Set SURVEY_TITLE to value obtained from 1st observation
Set WHERE clause to NEXT_QUESTION
Load Question-Text
Close QUESTIONS data set

IF QT='END' THEN
LOCKOUT='Y'
Delete JUMP_TABLE_LIST**.SLIST**
Link SET_LOCKOUT
Display Thank you/Termination Message

Open RESPONSE data set
Load response-object entry/entries
Close RESPONSE data set

If RF is 'mark all that apply' type
MATA = 'Y'
ELSE
MATA = 'N'

IF 'PAGE_BACK flag set THEN
Link GET_PREVIOUS_RESPONSE

CALL appropriate Response-Set formatting routine (an example will be provided later)

Display static HTML containing global output display formatting objects such as TABLES, FRAMES, and embedded JavaScript for input validation

Output Question-Text
Output Special-Instructions Text
Output Response-Object (SCL List containing dynamically generated HTML/JavaScript)

RETURN

FORMAT_XX Sections: These 10 sections of code are responsible for dynamically generating the output response object (HTML and JavaScript) using the response-text obtained from the RESPONSE data set. The appropriate section is triggered by the value contained in 'RF' during the current execution cycle. We will discuss one of the simpler response-objects (Radio-Button) in order to avoid confusion. The compound-object types, dynamic objects comprised of one or more dynamically generated objects are too complex to illustrate within the constraints of this paper. That is unless you want a really, really BIG paper ☺. These sections generally perform the following:

Create RS SCL List (Container to hold dynamically generated HTML/JavaScript)
Calculate # of columns in output display (N-times rows of 10 responses)
Link IFRAME_RESPONSE_OPEN (Create In-Line frame)
Generate dynamic header
Link INNER_TABLE_OPEN (provide structure for response objects)
Link ADD_CONTROL_PARAMETERS_FRAME (add required hidden variables to dynamic HTML to make selection re-select Survey Engine & provide required operational/informational parameters)
LOOP
Generate Nth response object
END LOOP
Generate 'Submit Response' or 'Re-Submit Response Button
Link INNER_TABLE_CLOSE
Link IFRAME_RESPONSE_CLOSE
RETURN

========== End Code ==========

Once the referenced code sections have been executed and have performed their operations, communication between the SAS©/IntrNet server and the respondent's browser is terminated. All control information required for progression to the next question has been embedded within the display's response-object region thereby negating the need for continued communication between the server and respondent via sessions or other methods, thereby conserving resources on the server. The resultant display within the respondent's browser contains all the information required by the respondent to successfully understand and provide their response(s) to the question.

Conclusion

Professional survey construction and administration places tremendous demands on an application. The advent of the WWW proffers one of the most efficient and cost-effective methods of administering surveys to individuals located in even the most remote locations. The SAS©/IntrNet-based Survey Engine is the application of choice for any organization requiring a professional commercially available product capable of fulfilling its survey administration needs.

Contact Information

Bernard R. Poisson
(801) 816-0137
BPoisson@Statprobe.com

Avoiding Entanglements: Migrating Applications to the Web

Eric Brinsfield, Meridian Software, Inc.®

ABSTRACT

After many years of developing SAS/AF applications, most of us find ourselves facing or pondering conversions to Web-based SAS® applications. Before beginning the process, we need to ask ourselves a series of important questions. For example, is the end result worth the cost of migration? Or, should we convert the entire application or just parts of it or should we just start over? And, what tools should we use in the new Web-based software?

In this paper, I will discuss many of the issues that we should take into consideration when preparing for and planning a migration from a fat-client application to a thin-client browser application. I will present specific case studies that illustrate successful migration paths.

INTRODUCTION

For many years, SAS/AF and SAS/EIS were the primary tools used for SAS user interface development. SAS/AF and SAS/EIS were, and still are, highly customizable and powerful. The functionality and "look and feel" of these custom applications varied widely based on the experience and training of the designers and programmers building them.

Now, as the use of Web browsers has become a household activity, most people are reasonably experienced and comfortable using a Web browser. Although every Web site application is different, they all have some common threads of behavior, based on the browser used to access the site. In other words, different Web browser applications seem to behave similarly just because they are using the same Web browser.

So, with widespread availability and familiarity, Web-based applications offer shorter learning curves and less resistance to change. In many cases, companies have set the Web browser as the standard interface for any new application installed on their networks.

But, how do we move existing applications to the Web browser interface without rebuilding the entire application. To address this issue, this paper will present a case study that illustrates one way to deal with this very common dilemma.

In this paper, I will use a case study to illustrate the issues involved in migrating an application to the Web.

For that case study, I will:

- Provide a general description of the case study
- Document the objectives of the project
- Identify the high-level set of options available for achieving the objectives
- Note the client requirements and constraints on our solution
- Discuss the option selection process
- Explain our rationale for the final solution

Case Study

In 1998, Meridian Software completed a pilot project and delivered a Quality Analysis system to a manufacturing client that provided the following features:

- Real-time data feeds from a production control system
- Master quality database stored in Microsoft SQL Server
- Data-entry subsystem built using SAS/AF Frame technology
- Real-time quality monitoring functions using SAS development tools including SAS/AF, SAS/Graph, SAS/Stat, and SAS/QC.
- Some historical analysis using the master quality database and SAS with the interface built in SAS/AF

In reality, our project was more than a pilot. In this case, pilot meant that we would develop a complete system, but only provide services for one department rather than all. We also deferred some large-scale decisions, such as data warehousing, until after we proved the value of the application. For clarification, note that department refers to discrete steps in the manufacturing process. The pilot focused on one piece of the puzzle with its own set of engineers and technicians.

Our projects with this client have been broken into phases. Phase 1 was the pilot project. Phase 2 involved adding more capability and statistical analysis and only expanded the scope to include one other department. The focus of this discussion is on our proposal process for Phase 3, which added significant capability and scalability and the potential for conversion into a Web application.

OBJECTIVES

Observations after 6 months of production usage

After using the pilot version of the Quality Analysis system for 6 months in a production environment, the client gained incredible insight into their manufacturing process as well as revealing facts about their suppliers' quality control. The system gave the quality analysts access to data that was never available before and reduced historical analysis time from six weeks to hours or even minutes.

After a second phase, that delivered even more statistical power, we found ourselves in the following situation:

- The client wanted to add more features and expand the system to cover additional departments (steps in the manufacturing process)
- Only a few users had access to the system, because they could not justify licensing SAS on the workstations of infrequent users.
- More technicians, engineers, and managers wanted to have access to the analysis and monitoring features
- SAS/Intrnet software had matured
- Our client had recently installed a set of Web servers and a corporate intranet. They set a new standard that favored Web browser interfaces for all new software. Any application that was not using a Web interface would eventually be phased out. Our application was the only SAS application at the site, which meant that SAS could have been phased out as well.
- Performance was unsatisfactory for users with low-powered workstations. With insufficient memory and CPU speeds, the SAS-based reporting tool seemed sluggish, although it worked fine on reasonable machines.

Phase 3 Objectives

Upon hearing the issues, detecting some of the future trends, and receiving their request for more features, we saw the need to move to a Web browser-based application. When we provided a proposal for Phase 3 enhancements and expansion, we included plans for a conversion to a Web application. With our plan, we hoped to achieve the following objectives:

- Add new analysis and graphics
- Expand availability by making monitor and analysis features available over the corporate intranet
- Get information and our system in front of engineers and upper management
- Insure future compliance with the corporate direction toward Web interfaces thereby increasing our chances of acceptance and showing how valuable SAS software can be
- Improve performance on low-powered workstations by using thin-client technology that shifts the workload to the server

But, we did not want to propose starting over and discarding all of the work from the past two years. So, we had to consider our options carefully.

OPTIONS

No Change

Doing nothing was not an option because they wanted expansion. We did not consider this option.

Upgrade SAS/AF Application and Spread to More Workstations

Because of cost considerations, we had to evaluate the option of keeping the application completely in SAS/AF and background processes. Although we included this option in our proposal, we pointed out that this option did not reflect the true capabilities of SAS with SAS/IntrNet. We provided a fair estimate of the upgrade price along with advantages and disadvantages.

If we had to add a few additional reports, staying with SAS/AF would have been a cheaper solution. But, after the pilot version had been in production for a year or so, we noted areas of frustration for the data-entry technicians that we needed to address.

In particular, data entry and reporting were all built into one application. The users did not like closing their data-entry windows in order to go look at a report. We solved this with multiple SAS sessions, but we did not consider this a robust solution for scalability. So, some redesign was necessary whether we converted to the Web or not.

The biggest negatives for a complete SAS/AF solution were cost of additional workstations with hardware upgrades and negative impression in view of the popularity of Web interfaces.

Convert to a Total Web Solution

We considered the possibility of converting all user interfaces to a Web application. This conversion would include all reporting tools, all analysis tools, all administrator tools, and data-entry subsystems. Given the power and complexity of the data-entry applications currently in use, a total conversion

seemed like a very expensive option.

In addition to deciding whether to migrate to the Web or not, we also had to consider which Web tools were appropriate. Specifically, what combination of the following tools should we use:

- SAS/IntrNet Application Dispatcher and Load Manager
- CGI
- JavaScript
- Active X
- JAVA
- AppDev Studio
- Or an additional Web development tool

Build a Hybrid Solution (partial conversion to Web)

As the final option, we also evaluated implementing a partial conversion, which would leave more complex functions that were used by fewer people, in SAS/AF, while converting queries and analytical functions to the Web. Very few users were entering new data, while many people wanted to access the informative graphs and reports available in the reporting and analysis subsystems. The hybrid solution looked appealing and offered some obvious payoffs.

REQUIREMENTS AND CONSTRAINTS

To evaluate the options fairly, we had to consider all of the requirements carefully and see which options withstood the customer's constraints. Specifically, we considered the following issues:

Usability

If more users were going to have access to the system, interface usability was critical for success. We could not ask high-level managers to take time to attend training classes on using our system. Because almost everyone is comfortable with a Web browser now, the Web conversion seemed like a very logical and important step for success.

Accessibility

Obviously, the Web application would be more accessible. The client was not willing to license SAS for every possible, casual user, who might want to view a graph. So, SAS/AF was definitely limiting accessibility throughout the company. The Web version would make the application available to everyone in the factory as well as employees at other plants on the same corporate intranet.

Maintainability

One of the reasons IT folks love the thin-client model is because thin-client applications are easier to maintain. All of the code resides in a central location on a server or servers rather than dispersed on individual user workstations. In addition, by using standard Web tools, the client would not be as dependent on contracted SAS expertise for all of their support, if that is an issue.

Performance

By forcing the processing back to the server with a thin-client Web application, we could reduce the demands on underpowered workstations. This benefit moved the scale heavily toward the Web conversion, but we also had to consider the increased workload on the server. By moving all processing to the same server along with increasing the number of users, we had to consider the possibility of overloading the existing server.

Security

Within the company, everyone was permitted to see the data and review the reports, but only selected employees were authorized to update the database. So, data entry posed the biggest security challenge.

Concurrent Access

Concurrency was built into the application from the original design and was still critical as we expanded the availability. With automated process updates every 5 minutes for two different data sources and manual data entry on a frequent but irregular basis, concurrent access to the data had to be studied again in view of the addition of more users coming in from the Web.

"Real Time" data with historical analysis capability

With potentially more people running historical analysis, concurrent access to the same database could become a performance problem. So, with the expansion to more users and conversion to the Web, we had to review the database design and server processing as well. The data-entryrequirements were in conflict with the analysis requirements, so how do you spell relief?

D - A - T - A M – A – R - T

We could not justify using data warehousing techniques in the pilot project, but before scaling up, we had to implement part of that strategy. Future expansion will involve even more formal data warehousing steps.

"Most Bang for the Buck"

As with any consulting project, our clients want to get the most "bang for the buck", in both the short term and long term.

We could achieve this only by:

- Reusing as much existing code as possible

- Evaluating cost of development versus value of end result (ROI)
- Delivering the new version in as short a time as possible so new capabilities were available sooner
- Keeping maintenance costs low (maintenance cost had been negligible over the past 2 years already)
- Decreasing demand on workstations so our application and SAS were not the cause for an increase in hardware requests
- Getting the most out of their SAS license, opting to add new products and capability rather than additional licenses, which would be underutilized
- Increasing the expected life span of the application thereby decreasing near-term costs of replacement and increasing long-term ROI

OPTION SELECTION PROCESS

In our proposal to the client, we provided cost figures for each major option along with the description of the pros and cons of each. As you may have guessed from my comments earlier, we recommended a hybrid solution, which left data-entry and administrative tools in SAS/AF and moved all monitoring and analysis solutions to the Web.

Basically, we could not recommend building new features with SAS/AF for all of the reasons listed above. Although SAS/AF is a powerful tool, the Web is too popular and widely accepted to ignore (even after the "dot com bust").

We also considered moving everything to a Web application, but to achieve the same level of flexibility in the data-entry features, we would need more time and more JAVA development.

At the time of the proposal, SAS Software's AppDev Studio did not support JAVA Server Pages (JSP) and we (and many of our clients) were not satisfied with the performance of JAVA applets. So, although a JAVA approach might make sense today, we felt that increased price and decreased performance were significant risks with a total Web conversion at that time.

The hybrid solution could be completed quickly, providing the greatest functionality and accessibility at the least cost.

FINAL SOLUTION

In our final solution, which went into production in December of 2000, we implemented a hybrid solution that kept data-entry functions in SAS/AF and moved all reporting and analysis functions to the Web browser. To support our new design, we also recommended some hardware and database changes that improved the overall performance and function of the Web application.

Each of the major changes is described below, starting with the system architecture changes.

Split the server

In anticipation of increased demand on the server when more thin-client users started executing SAS on the server, we recommended splitting the single Windows NT Server into two servers, thereby creating a three-tier architecture. One server housed the Microsoft SQL Server database, while the other became the Web server and SAS Application Server. By splitting these functions, multiple SAS sessions on the application server were not competing with SQL Server for CPU and I/O time.

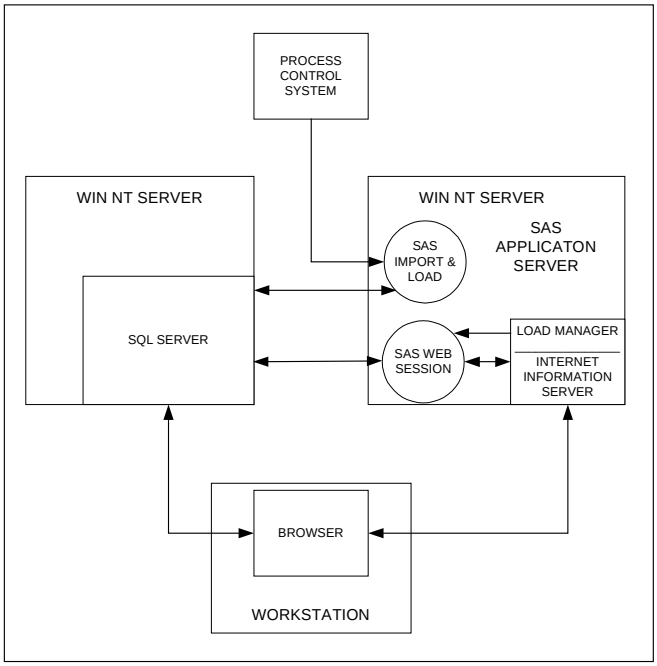


Figure 1. Three-Tier System Architecture

After implementing this split and upgrading the old machine at the same time, we saw significant performance improvements in the SAS/AF-based data-entry applications and the import processes even while supporting thin-client analytical users. To achieve this split, we did have to modify the design of our import process slightly to take advantage of the parallel processors. In addition, we optimized all queries to split the workload appropriately between the two servers. Large queries would reduce the result sets within SQL Server before passing them back to SAS for further processing.

Split the database

The original database was designed to support real-time updates and data entry. Consequently, historical analysis activity could sometimes be slow

or go into wait state during large imports. These conflicting objectives provide part of the motivation behind today's data warehouse technology. As part of Phase 3, we recommended implementing a more typical data warehouse approach before expanding to any other departments.

So, we took two steps to provide the fastest performance for both types of users (transactional and analytical). First, we partitioned the database into current data (less than three months old), historical data (between 3 and 12 months old), and archived data, which was moved offline.

All transactional processes, such as data entry and automatic import, accessed only the current partition. In addition, monitor programs that took a quick pulse of the system in real time, used only the current data.

Secondly, we designed and built a data mart that is refreshed twice a day automatically, but can be refreshed on demand if desired. The data mart was designed to optimize analytical processing and included data from the current and historical databases.

By implementing this strategy, we improved performance for data entry and imports, while providing impressive speeds in the Web application. At any given time, the data mart may not contain the most recent process data, but for historical analysis or defect trend analysis, new data points are not critical. In addition, this strategy increases the scalability of the application as we incorporate more departments into the application.

Upgraded Data Entry

Although we did not convert the data-entry application to the Web, the hybrid approach provided enough leeway in the budget to allow us to streamline the existing SAS/AF application. By removing functions from the SAS/AF application that had been shifted to the Web, we could give users an application that was more focused on their specific data-entry tasks. They could use the Web for monitoring, while using SAS/AF for the data entry on the same machine.

We also took advantage of the split server and optimized the data-entry inserts and queries in consideration of that design. Consequently, the technicians were much happier, because data entry was faster and easier and they could pop over to the Web application to see their results instantly without closing their data-entry application or running two SAS sessions.

As a standard client-server application, the data-entry subsystem was unaffected by any workload on the SAS application server, because SAS and the SAS/AF application ran on the data-entry workstations and communicated directly with SQL Server. The data-entry application did not need to request any services from the SAS application server.

Converting Monitoring and Analysis to the Web

As part of phase 3, we evaluated the current application and categorized the reports and graphs as either process monitors or quality analyzers. We used this dichotomy in the Web application. In addition, the new version also added a yield analysis feature that produced reports on yield rates and production levels rather than on quality and defect levels.

Each category has the following attributes:

Process Monitors:

- Provide snapshots of current processes in real time
- Utilize the "current" database, so historical data was not available
- Are customized for a specific function
- Execute with a default set of options and filters, so they could be run with a single click of the mouse
- Run extremely fast
- Offer very few customizable options, if any at all
- Include status reports, event viewers, control charts, trend charts, density plots, and Pareto charts

Quality Analysis and Yield Analysis:

- Provide a historical view
- Utilize the data mart and did not usually touch the real time database
- Are extremely flexible, permitting the user to define data subsets and specify graph or reporting options
- Include predefined and custom reports, control charts, graphs, trend charts, density plots, Pareto charts and advanced analysis
- Enable users to download the data that was used to create the report into SAS data sets or Excel

We utilized SAS/Intrnet and the Application Dispatcher with Load Manager for load balancing. The Load Manager controls how many SAS sessions are made available to the browser sessions and reuses open sessions that are not in use.

We used a combination of HTML, JavaScript, and VB Script with Active Server pages to make the Web site dynamic. SAS was used primarily for reporting and analysis after the user selects options. The Active Server pages communicate directly with SQL Server to create dynamic select lists and to perform non-analytical functions for the

browser sessions.

Because the Meridian Software standard for software development encourages SAS/AF developers to isolate SAS programs in a standard directory or macro library rather than embedding them within the SAS/AF application or catalog entry, reuse of existing report programs was very easy.

Instead of collecting and passing user parameters with an AF window, we collected the parameters within a browser and passed them to the same programs. Our only major modifications to the report programs were to add utilization of the Output Delivery System (ODS), because we also upgraded to Version 8 in Phase 3. ODS made the conversion to the Web even easier.

SUMMARY AND FINAL ANALYSIS

Faced with upgrading an existing SAS/AF application, you should seriously consider converting all or part of the application to the Web. SAS/IntrNet and the CGI interface provide a painless path for existing SAS programmers to take into the Web application development world. SAS programmers do not have to jump directly to JAVA.

With proper analysis, you can convert quickly and cost effectively to the Web browser without losing the power of SAS. We achieved a significant cost savings by settling on a hybrid solution rather than a total Web conversion. Our client is very happy with the new interface and the users seem more comfortable working in the Web environment.

If we were starting on the project today, we would probably consider using more JAVA now that Version 2.0 of AppDev Studio is available. The development tool offers more features with more communications capability and support for JAVA Server Pages. It is quite likely that we will be using AppDev Studio in Phase 4, when that happens.

CONTACT INFORMATION

Eric Brinsfield
Meridian Software, Inc.
12204 Old Creedmoor Road
Raleigh, NC 27613
(919) 847-6750

merecb@meridiansoftware.com

www.meridiansoftware.com

Delivering OLAP Solutions to the Web

Tammy Gagliano, SAS Institute Inc., Carmel, IN
Tony Prier, SAS Institute Inc., Kansas City, MO

ABSTRACT

What is OLAP (Online Analytical Processing) and how can it benefit your organization? Can you or should you be delivering your OLAP applications via the web? If so, what tools are available to you?

These questions and more will be answered during the presentation, which will include live demonstrations of the Java technologies available to you for web-enabling your OLAP solutions. Using tools that are part of AppDev Studio™, you can develop applet-based or servlet-based (including JavaServer Pages™) OLAP applications. Comparisons will be made regarding the pros and cons of each approach as well as how the different technologies can be used together to provide a complete solution.

INTRODUCTION

The ability to look at data from multiple dimensions, or areas of interest, and to access that data quickly and in a consistent manner is key to the success of any business intelligence application. Today, OLAP (On-Line Analytical Processing) is acknowledged as a key technology for a successful implementation of any business intelligence system and is vital for creating strategic competitive advantages for any organization.

OLAP provides analysts, managers and executives the freedom to interrogate their enterprise data. Using multidimensionality, the data is organized according to the categories that reflect the way the user thinks about the enterprise.

The SAS OLAP solution uses SAS Institute's multidimensional database (MDDB) server to provide high-performance OLAP capabilities within an integrated data warehouse environment. MDDB's package warehouse data into multidimensional data structures, which deliver data to OLAP client software upon request. The SAS OLAP Server bundles all the required server-side functionality you need for defining your OLAP data. Using the SAS OLAP Server you can:

- create MDDB's using PROC MDDB,
- register OLAP Metadata,
- create Access Control definitions,
- use Model Coordination.

The SAS OLAP server also supports Hybrid OLAP (HOLAP). HOLAP provides the ability to partition your multidimensional databases so that data can be split across multiple MDDB's that could reside on different servers. Data can also be accessed from Relational databases, SAS data sets, summary data sets, flat files, etc. This is all meta data driven and often provides a much more open and scalable solution.

With SAS OLAP Server on the back-end, you can access data from any source – in any format. Now, you are ready to choose the front-end or the client interface to the data. For that, many organizations today are turning towards thin-client solutions – interfaces that can be easily deployed through the Internet or corporate-owned intranets.

The growing use of the Web as a way to deliver client services also results in the reduction of the costs associated with upgrading applications. The role of the client interface becomes that of a viewer of information delivered by the server. Adding a new OLAP user can be as simple as e-mailing them the URL of the OLAP application.

The SAS OLAP solution is web enabled and supports a full range of client/server configurations including totally thin-client. By relying on MDDBs on the server side for storage and processing, SAS OLAP clients can be used soley as viewers of MDDB data. Users can run queries and generate reports from their browser without the need to run a SAS session on the client. The SAS OLAP solution provides:

- web publishing tools such as the SAS Output Delivery System (ODS) available as part of base SAS software in Version 7 as well as the HTML Formatting Tools available with Release 6.12 of the SAS System.
- the Application Dispatcher and htmSQL which are both part of SAS/IntrNet™ software and are based on CGI technology.
- Java-based technology such as servlets, JavaServer Pages (JSP), applets or applications.

For the remainder of this paper, we're going to focus on the Java-based technologies available in AppDev Studio and how they fit as part of the SAS OLAP solution. AppDev Studio (ADS) is a complete, stand-alone development environment. It is an integrated suite of development tools that provides the power you need to build web-enabled applications that use HTML, CGI, Java servlets and JavaServer Pages (JSP) as well as sophisticated Java applications and applets. ADS contains many OLAP specific components and interfaces to help build sophisticated yet easy-to-use OLAP documents and deploy them to the web.

APPDEV STUDIO: WEBAF™ AND WEBEIS™ SOFTWARE

webAF software

Within the AppDev Studio suite of products, webAF software is the primary development tool for Java-based applications. It helps you build applications that are lightweight, easy to manage, and instantly connect to SAS software. Support for the creation and debugging of applets, servlets and JavaServer Pages is provided. webAF software's component-based visual development environment enables easy access to SAS software from Java classes, transparent access to SAS/AF objects, access to JDBC data sources, access to remote tables and MDDBs stored on one or more SAS servers, and access to SAS compute power through remote procedure submissions.

If you are a webAF user, you will have to have some knowledge of Java as it is a Java development tool. Code is automatically generated for you as you drag-and-drop your components into your project and it also provides a simple interface for linking properties and setting event handlers to get components talking to each other.

However, you still need to have a good understanding of the underlying language in order to build a sophisticated application.

webEIS software

For those of you who are not Java experts, but still need to build OLAP applications, you should take a look at webEIS software. webEIS covers the entire spectrum of user needs – from business executives who want an easy-to-use document for viewing multidimensional data to business analysts who expect sophisticated reach through, data visualization, analysis and reporting capabilities. It is an application written in Java that makes it easy to create interactive, EIS-style documents containing charts and multidimensional tables. A webEIS document is published on the Web as a Java applet or JSP. You do not write any Java code to create these applications. They are created purely using the intuitive and powerful point-and-click, drag-and-drop interface provided by webEIS software. It is also important to note that any webEIS documents you create can easily be incorporated into a webAF project for further customization using Java.

Using webEIS, you can point to existing MDDB or HOLAP data on a remote SAS server. You can also point to an existing SAS/EIS object, which enables you to reuse functionality and behavior defined in the remote server object. Once these documents are deployed, your users can:

- navigate intuitively through volumes of data using drill down, expand/collapse or subsetting operations.
- perform analysis, and row and column calculations at run time.
- apply traffic lighting or exception highlighting to record important trends or outliers in the data.
- dynamically reach through to detail data anywhere on the network without having to know where the data resides.
- and more.

All of this functionality is built into the various MDDB components that are used in the background. It allows users to work interactively on the client from their Web browser while they are unknowingly communicating back to a SAS server where the data is stored. webEIS is implemented in a multi-user client/server mode and offers consistent rapid response to queries, regardless of database size and complexity.

During this presentation, we will walk you through how to build an OLAP document using webEIS and then show how easy it is to deploy the document either as an applet or JSP – without you having to write a single line of code!

JAVA TECHNOLOGY

We've talked about the tools that will enable you to deploy your OLAP applications using Java technology, but we haven't discussed why you should consider choosing Java as your development platform to begin with. Just why is it that Java is considered by some to be the premier language of choice for providing highly interactive user interfaces to the Web browser?

Java technology is continuously being enhanced to provide components and features that elegantly handle problems that are difficult in traditional programming languages such as multithreading, database access, network programming and distributed processing. It is ideally suited for the Web because it is:

- portable across platforms by virtue of it being an interpreted language. A Java Virtual Machine (JVM) must be available on the user's machine. Most browsers (e.g. Netscape and Internet Explorer) contain a JVM as part of their standard installation.
- secure through its ability to maintain the integrity of the client machine. The JVM has the opportunity to enforce the rules specified by the security manager to ensure that the integrity of the user's machine is maintained and that the applet does not have access to resources other than those the user has specifically granted. In addition, it allows vendors to digitally sign the archive file to identify the vendor that created the JAR file. This allows the user to decide whether they "trust" the software provided by this vendor.
- considered to be a true thin client solution because of its ability to be dynamically downloaded on demand versus permanently installed on the user's machine. This eliminates the user or IS staff at your site from having to install and maintain current versions of software on each client machine.

JavaBeans™

Another reason that Java is so popular is because of its JavaBeans and Enterprise JavaBeans architectures. These object-oriented frameworks allow for the building of some very powerful components that make it easy for developers to create, deploy and manage cross-platform applications. webAF offers its own set of JavaBeans compliant components referred to as InformationBeans™. These beans allow you to tap into the enterprise data access, data warehousing, and decision-support capabilities of SAS software.
You can build sophisticated web applications that can:

- access SAS data libraries on a remote server allowing access to any data source that SAS can access through its extensive list of database engines.
- display SAS multidimensional databases in a ready-to-use OLAP viewer that has built-in functionality for drilling down through the data, subsetting, exporting the data to a spreadsheet, applying exception highlighting, adding computed columns and more.
- perform compute services by submitting SAS code on the server to perform tasks such as statistical analysis, reporting, summarization, and quality control -- just to name a few.

There are many components available with webAF that are specifically designed for generating custom OLAP applications – accessing the power of SAS on the server for both data (MDDBs) and compute resources. For example, the com.sas.sasserver.mdtable.MultidimensionalTableV3Interface is an InformationBean that is designed specifically to read and manipulate MDDB data.

The data supplied by this model can be displayed on the web using one of the following viewer components:

- in a Java applet using the MultidimensionalTableView component (found in com.sas.mdtable)
- or in JavaServer Pages/Servlets using the MDTable TransformationBean™ (found in com.sas.servlet.beans.mddtable.html). TransformationBeans are discussed in more detail later in this paper.

Both of these viewers were designed to display MDDB data in a table format and communicate with the above model through model/view communication. Model/view communication enables a viewer (typically a visual control) to communicate with a model (typically a non-visual component) based on a set of common methods that are defined in an interface. The viewers seemlessly communicate with its attached model without you having to write additional Java code to perform tasks such as retrieving rows to display, handling updates that might be made through client-side table interaction and more.

InformationBeans virtually open the door to SAS, which enables

your web applications to take advantage of any and all functionality that SAS software provides. And using AppDev Studio, the power of having SAS on the server can be exploited without having SAS software installed on the client machine.

Java Applets

Applets lend themselves nicely for creating highly interactive user interfaces for thin-client applications. With applets, you avoid having to install an application locally on a user's machine. Instead, when an applet is executed (usually by being called from within an HTML page), the necessary files are automatically downloaded from the Web server. The applet is then loaded into memory and displayed in the browser. Typically, applets are subject to security restrictions on the client, the server, or both. Make sure that you understand any limitations that your production web environment may impose.

webAF's Project Wizard quickly steps you through creating an applet. Then you can begin building the pieces of your application using webAF's drag and drop interface to add visual and non-visual components to a window.

Even though the power of Java makes applets a popular choice among many Web application developers, applets present some deployment hurdles that you should be aware of. By "deployment" we mean the mechanism by which the applet is made available to the Web browser user.

Some of the most common complaints you hear about applets are:

- the need for the Java Plug-in which is an ActiveX control. This raises some security concerns for sites, which do not allow downloading of any ActiveX technology.
- the start-up time required to download an applet's implementation files and all required class libraries from the web server host to the client machine.
- the amount of time it takes the applet to set-up the server side environment – for example, to instantiate a SAS session and load large tables in that session that are then accessed from within the applet.

AppDev Studio developed applets require a Java VM that is JDK 1.3 or later. Popular web browsers like IE 5.0 and Netscape 4.5 do not natively support such levels of the Java VM. In response, Sun has produced the Java Plug-in to allow applets to run with the JDK level that they require in these browser environments. As a result ADS developed applets need to use the Java Plug-in on order to run in these browsers. Some sites do not allow downloading of any ActiveX controls because these controls have access to the local machine and can create havoc if coming from an unreliable source. One way to address the security issue that the Plug-in raises is to install it on a local *intranet* where the browser can be configured to allow it to be downloaded by applets as needed. This allows use of the Plug-in without allowing other ActiveX controls to be downloaded from the Internet. Another option, which avoids the ActiveX/Java Plug-in issue altogether is to consider using servlets/JSPs to deploy your application which is discussed later in this paper.

The latter two concerns from the above list fall under the 'applets are slow' category. Both tasks are performed every time the applet is invoked. When dealing with a slow network or dial-up connection, slow applet performance can severely limit their effectiveness.

To help overcome these problems, AppDev Studio provides a number of techniques that you can use. Where applet download times are a problem, SASNetCopy or JSASNetCopy along with the Java Plug-in's applet caching feature can be used to install the applet classes and any required extension classes onto the client machine. Both of these technologies use a zero-administration, auto-install approach. Using either technology, the applet's required class libraries can be downloaded automatically the first time that the applet HTML is referenced and then cached on the client machine for subsequent re-use. Updated class libraries are made available by simply installing them on the web server and modifying a server-side configuration file. Then, the next time a user accesses the applet HTML, the updated class libraries will automatically be downloaded and cached on the client machine.

Where SAS startup time is an issue, the AppDev Studio Middleware Server's SAS session preloading and MDDB or class preloading feature can be used such that the required SAS support is available to a client virtually instantaneously. This feature can be used for servlet/JSP environments as well.

For more details on these techniques, refer to the following papers available from the AppDev Studio Developer's Site:

- Improving Applet Performance: SASNetCopy and JSASNetCopy: http://www.sas.com/rnd/appdev/Tools/applet-performance.htm

- AppDev Studio Middleware Working with User Profiles: http://www.sas.com/rnd/appdev/doc/MWSprofiles.htm.

Java Servlets and JavaServer Pages™ (JSP)

While the techniques described above will help speed up applet startup times, applet technology may still present some problems that lend themselves towards a look at servlet or JavaServer Page technology instead. For example, suppose you have an applet that needs a connection to a SAS session on a remote server in order to take advantage of data stored there or the compute power of a SAS server. If you are deploying your solution in a more restrictive network environment (for example through firewalls, proxy servers, or using secure HTTP), you may have to undertake considerable effort to get applet solutions to work. In such a network environment, you should also consider using a server-side Java solution such as JavaServer Pages or Java servlets. These solutions simply return HTML back to the client device while the code runs on the application server machine. Both the server-side program and the SAS server reside behind the firewall.

At a conceptual level, servlets are just like applets except that they run in the server environment instead of the browser environment.
In the firewall scenario mentioned above, hosting the processing on the server eliminates the problems with communications from the client machine to the SAS server. Additionally HTTPS is fully supported since the only requests being passed between the client and the web server are HTTP requests.

JavaServer Pages are actually an extension of the Java Servlet API. However, developing an application based on JSP technology does not require in-depth knowledge of how servlets work. JSP technology makes it easier to build web pages with dynamically generated content through the use of Java's component-based technology. It separates the user interface from the application logic, which enables:

- the page designer to focus on writing the HTML that controls the overall page design.
- the application developer, using JSP tags (or scriptlets), to generate the dynamic content portion of the page.

Java is the native scripting language for JSP, which means you can develop platform-independent applications due to Java's "Write Once, Run Anywhere" characteristic. In the simplest

terms, a JSP page is simply an HTML page with embedded Java code. If you are comfortable writing Java code, you can embed Java programs directly in the JSP page using scriptlet tags.

If you're not a programmer, you can take advantage of reusable, cross-platform components (JavaBeans or Enterprise JavaBeans) with JSP-specific XML tags (e.g. USEBEAN tag) that make it simple to instantiate JavaBeans components and manipulate properties on a component from within your web page.

Through the use of this component-based logic, page developers are able to develop sophisticated, interactive web-based applications with very little Java programming knowledge. JavaServer Pages provide other benefits as well. Execution of JSP pages is simple and fast because they can be executed on any Java-enabled Web server, application server, or operating system. This differs from other technologies that have specific server requirements. For example, Microsoft's Active Server Pages™ technology is dependent on other Microsoft technology (such as COM).

JSP technology holds advantages over traditional CGI-based solutions, which have shown limitations with respect to scalability. With each CGI request, a new process on the server is launched. When multiple users access the program concurrently, these processes can quickly consume all of the web server's available resources and can bring the application to a halt. When a JSP page is first called, if it does not yet exist, it is compiled into a Java servlet class and stored in the server memory. A Java servlet is a Java-based program that runs on the server as opposed to an applet, which runs on the browser. This enables very fast responses for subsequent calls to that page (and avoids the CGI-bin problem of spawning a new process for each HTTP request, or the runtime parsing required by server-side includes).

Finally, JSP differs from other technologies because it utilizes reusable components and tags, instead of relying heavily upon scripting within the page itself. Through its use of servlet technology and Java server-side processing, it offers:

- scalability for complex, dynamic web pages
- a true thin-client deployment strategy (with an even smaller footprint than applets which require the Java classes to be downloaded to the client)
- persistence due to Java's true session management capabilities.

However, like CGI, the graphical user-interface (GUI) portion of the application is somewhat limited to what the HTML form elements can provide. For more detail on comparing this technology to CGI or Applets, refer to *Getting Started with AppDev Studio, First Edition***.**

In addition to what the servlet/JSP platform provides, AppDev Studio offers additional functionality, which makes the development of web pages in the servlet/JSP environment even easier. Those features are:

- TransformationBeans (available for both servlets and JSPs)
- Custom Tag Library for the TransformationBeans and key InformationBeans (custom tag technology available for use within JSPs only)

TransformationBeans

TransformationBeans are a set of specialized JavaBeans included with webAF. These beans are designed to consume data from an existing webAF model (e.g., using the DataSetInterface model that retrieves data from a SAS data set) and transform it into HTML to display on the Web page (e.g., using the Table TransformationBean, which displays the data in an HTML table).

When using webAF's InformationBeans and TransformationBeans together, not only does the page author have access to the power of SAS on a remote server but they also spend less time writing HTML. The TransformationBeans do all the work!

The table below, lists the TransformationBeans you will find useful when building an OLAP application:

Package: com.sas.servlet.beans.mddbtable.html	
TransformationBean	**Description**
MDBar	creates an HTML Bar chart image that represents data stored in a MDDB table
MDCombination	creates an HTML Combination chart image that represents data stored in a MDDB table
MDDrillPath	displays text that corresponds to the current drill path within an MDDB table
MDExportToExcel	exports MDDB data to MS Excel (using CSV file format)
MDFinder	generates HTML to represent a selector that allows a user to specify a text string to search for in a selected Column of an MDDB table.
MDNavigationBar	generates HTML visuals which allow a user to scroll within an MDTable Transformation Bean
MDPie	creates an HTML Pie chart image that represents data stored in a Multidimensional Database.
MDQuerySelector	generates HTML and JavaScript to represent a selector that allows a user to dynamically change Rows, Columns, Measures and Statistics displayed in an MDTable TransformationBean
MDScatter	creates an HTML Scatter chart image that represents data stored in a MDDB
MDSegmentedBar	creates an HTML SegmentedBar chart image that represents data stored in a MDDB
MDSelectorMenuItem	used to populate a Menu item in the MenuBar TransformationBean.
MDSortSelector	generates HTML and JavaScript to represent a selector that allows a user to change the sorting criteria for MDDB data displayed in a MDTable TransformationBean
MDSubsetSelector	generates HTML and JavaScript to represent a selector that allows a user to change the subset criteria for

	MDDB data displayed in a MDTable TransformationBean
MDTable	generates HTML tables to view data stored in a MDDB (attached to the MultidimensionalTableV3Interface InformationBean)
MDTopBottomSelector	generates HTML and JavaScript to represent a selector that allows a user to subset according to top N/bottom N criteria for MDDB data displayed in a MDTable TransformationBean
MDTotalsSelector	generates HTML and JavaScript to represent a selector that allows a user to specify totals to be displayed in a MDTable TransformationBean

Package: com.sas.servlet.beans.html	
TransformationBean	**Description**
MenuBar	generates HTML that builds a menubar which can be populated with Menu items and MDSelectorMenuItem items which launch the various TransformationBean selectors described in the above table
Menu	used in conjunction with the MenuBar and MDSelectorMenuItem components to populate a MenuBar

Note: All HTML beans adhere to functionality available in HTML 3.2.

There are many more TransformationBeans available in the com.sas.sservlet.beans.html package than what is listed in the above table. To see a complete list along with examples, use the following link to go to the *Server Side Examples* page on the AppDev Studio Developer's Site:
http://www.sas.com/rnd/appdev/webAF/server/examples.htm

At first glance, it might seem that TransformationBeans are simply a set of convenience objects that help a Java developer implement an HTML page. Compare the HTML code necessary to place a form input element such as a check box on a page

```
<input type="checkbox" name="box1"
       value="checked" checked>Label
```

with the Java code that implements a CheckBox TransformationBean

```
<%
com.sas.servlet.beans.html.Checkbox
  checkbox = new
  com.sas.servlet.beans.html.Checkbox(
  "box1", "Label", true, "checked");
checkbox.write(out);
%>
```

The simple HTML code is straightforward and easy to read. However, the TransformationBean makes it easy to implement dynamic content. If you wanted to simply check or uncheck the form control based on the value of a boolean variable named **status** defined in your JSP, the code without the TransformationBean looks like

```
<% if (status) { %>
<input type="checkbox" name="checkbox1"
   value="checked" checked>Label
<% } else { %>
<input type="checkbox" name="checkbox1"
   value="checked">Label
<% }%>
```

Using the TransformationBean, the JSP code contains

```
<%
com.sas.servlet.beans.html.Checkbox
   checkbox = new
com.sas.servlet.beans.html.Checkbox(
  "checkbox1", "Label", status,
  "checked");
checkbox.write(out);
%>
```

The use of the bean not only aids in dynamic content generation, but it is also easy to read and debug.

The above TransformationBean example was taken from a paper found on the AppDev Studio Developer's Site called *Why use TransformationBeans?* To read the complete article use the following link:
http://www.sas.com/rnd/appdev/webAF/server/whytbeans.htm

SAS Custom Tag Library

Along with the standard scriptlet and JSP specific tags, the JavaServer Pages specification also supports something called *custom tags*. A custom tag is a user-defined JSP language element.

Custom tags are usually distributed in the form of a *tag library*, which defines a set of related custom tags and contains the objects that implement the tags.

webAF offers an extensive tag library that corresponds to the TransformationBeans and key InformationBeans that you would want to use in your JSP applications. You do not have to be a Java expert to use these custom tags. The tags are HTML- or XML-like. You simply specify values for the attributes on a given tag statement in your JSP.

Continuing with the Checkbox TransformationBean example shown above, the equivalent custom tag code would look as follows:

```
<sasads:Checkbox id="checkbox1"
   text="Label" selected="<%= status %>"
   value="checked" />
```

In the section labeled *USING WEBAF TO BUILD A JSP-BASED OLAP APPLICATION* found later in this paper, you will see an OLAP example of the custom tag code that gets generated as you drag and drop components from the webAF toolbar onto your JSP project. The custom tag source code that gets generated is much easier to read and maintain versus scriptlet or standard Java code. As a result, you'll find that using custom tags increases your productivity because of the encapsulation they provide for more than often very complex tasks.

For more details on the SAS Custom Tag Library and what it has to offer, go to the following papers found on the AppDev Studio

Developer's Site:

- WebAF and the SAS Custom Tag Library http://www.sas.com/rnd/appdev/webAF/server/customtags.htm
- SAS Custom Tags: Overview http://www.sas.com/rnd/appdev/webAF/taglib.htm.

LETTING WEBEIS GENERATE YOUR OLAP APPLICATION FOR YOU

One of the easiest and quickest ways for you to design an OLAP document and deploy it to the web is to use webEIS software. Using webEIS, you do not need to write a single line of code. It provides you with a friendly user-interface to perform all the tasks you need such as

- creating OLAP documents that have one or more sections to them.
- choosing a remote data source.
- adding components to the sections within your document.
- rearranging the components and change their properties to give them the exact behavior and look and feel that you desire.
- deploying the finished document as either an applet or JSP.

Saving the document as a JSP application is new with Version 2 of AppDev Studio. Instead of choosing **File->Save as Applet…** from the webEIS menu,

1. Select **File -> Save As JSP…**
2. In the Save as JSP window, note the name of the JSP and the path where all dependent files will be saved. By default, JSPs are named using the name of the current document. They are saved to the default Web server location if one has been specified; otherwise, they are saved to the current document directory.

   When you save a document as a JSP, several JSP files are created:
   - a main JSP file
   - one JSP file for each section
   - any associated resource dependencies (for example, images that were specified with absolute file paths) are also saved.
3. (Optional) Click the **Package files as…** check box if a Web server has not been specified. All files are then saved as a .zip file. You can easily move the .zip file to another location, such as a Web server.

The EIS document itself is still available and is saved in a file with a .eis extension. This document can be edited later within webEIS. Additional applets and/or JSPs can be generated from it at any time.

Some developers like to have more control over their application or add additional behavior than maybe what a user-interface like webEIS provides. For example, an OLAP document may be only a small part of a much bigger application. The application may need to surface additional reports against relational data or submit other tasks such as more sophisticated data analysis or forecasting. Or, some developers simply like to write their own code instead of letting an application generate it for them. If you're one of these people, you may prefer to use webAF software to develop your OLAP applications.

The underlying OLAP components are the same between the two products. webEIS uses the MultidimensionalTableV3Interface to access MDDB data from a remote SAS server. You would use the same model in webAF regardless of whether you're building applets or servlets. The difference between the two tools obviously being that in webAF, you will be writing the code to make the various components communicate with each other and to control the overall behavior of the application. The remaining two sections of this paper focus on using webAF to build your OLAP solutions.

USING WEBAF TO BUILD A JSP-BASED OLAP APPLICATION

With webAF, you can choose to create either a JSP or servlet project. It's not unlike the steps you go through when building an applet. webAF offers various components that are available from its component palette. You drag and drop those components onto the project you are creating. Some code gets generated automatically for you. You can use various wizards, dialogs and properties windows to specify options and control some of the behavior. But more often than not, you will end up going directly to the code and writing some yourself to complete the application!

First, let's take a look at building a JSP-based OLAP application using webAF. We'll take a look at the custom tag code that gets generated and how to manipulate that code. In the next section, we'll show the same example from a servlet perspective. Again, the underlying components used are the same – only the implementation differs. One uses custom tags and scriptlets mixed with HTML elements… the other uses pure Java code.

To begin creating a JSP project, select **File→New** from the webAF menu. When the New window appears,

1. select JavaServer Pages Project from the Projects tab
2. type `sugi26` in the Project Name field
3. select OK
4. select Finish in the Project Wizard – JSP options window.

Change the component palette from JSP/Servlet to MDDB JSP/Servlet. Now we need to drag the six components needed for this OLAP application onto the project. Select the Visuals tab in your webAF project frame area, then drag a

- MultidimensionalTableV3Interface from the SAS tab (a Connection object will automatically be created along with this component since it needs to make a remote connection to SAS to get the data),
- MDCommandProcessor from the Data Viewers tab,
- MDExportToExcel from the MDDB Tasks palette,
- MenuBar from the Selectors tab,
- MDTable from the Data Viewers tab,
- and MDBar from the Graphics tab.

The Visuals tab within webAF should look like the following after dropping the components.

Open the customizer for MDModel1. On the Data Source tab,

1. type `sugidemo` in the Metabase field

2. type SUGIDEMO.CBMDDB in the Database field.

On the Query tab,
1. type Product Hierarchy for Add New Item and select the Add button for Rows
2. type Time Hierarchy for Add New Item and select the Add button for Columns.

On the Measures tab type,
1. Actual for Measure and SUM for statistic
2. Forecast for Measure and SUM for statistic
3. Difference for Measure and SUM for statistic.

On the Totals tab type,
1. Product Group for Level
2. Total for Label
3. true for state.

Close the MDModel1 customizer.

Open the property sheet for MDExportToExcel1, type
1. index.jsp in the formAction field
2. exportForm in the formName field
3. false in the render field.

Setting render to false will prevent the tag from writing itself out at the location of the custom tag. The tag will be rendered later when it is referenced as a menu item in the Menu bar. Close the Property Sheet Window.

Open the customizer for menuBar1. On the Menu Bar tab,
1. select MenuBar in the Tree
2. select SELECTOR_EXPAND for Menu Type
3. type | in the separator field
4. select Menu in the Tree
5. type Subset in the Label field
6. type /assets/subset.gif in the Image field
7. type Subset in the Alternative Text field
8. expand the Subset Node in the Tree
9. select MDSelectorMenuItem in the tree
10. type MDModel1 in the Model field
11. select SUBSET_SELECTOR from Selector Type choice box
12. select MenuBar in the Tree
13. select New→Child
14. type Export to Excel in the Label field
15. type /assets/export.gif in the Image field
16. type Export To Excel in the Alternative Text field
17. type document.exportForm.submit(); in the Custom Action field
18. select New→Child
19. select MDSelectorMenuItem from the Menu Item Type choice box
20. type MDExportToExcel1 in the Selector field

Close the customizer for menuBar1.

Open the customizer for MDTable1. On the MDTable tab,
1. type MDModel1 in the Model field
2. set the border width to 1
3. type MDCommandProcessor1 in the CommandProcessor field.

On the NavBar tab type,
1. /assets/double_left_03b.gif in the First field
2. /assets/left_03b.gif in the Previous field
3. /assets/right_03b.gif in the Next field
4. /assets/double_right_03b.gif in the Last field.

Close the customizer for MDTable1.

All the properties of the MDTable are not displayed in the customizer. To view all the attributes available in the custom tag open the help window for the MDTable custom tag. To open the help window,
1. select MDTable1 in the Project Navigator,
2. use the right mouse button to display a pop-menu
3. select help from the pop-menu.

Switch to the Source tab in webAF project. Edit the MDTable1 custom Tag directly to specify additional attributes. New attributes are in **bold**.

```
<sasads:MDTable id="MDTable1" maxRows="25"
  maxColumns="10" scope="session"
  borderWidth="1"
  commandProcessor="MDCommandProcessor1"
  model="MDModel1"
  detailDataStyleSheet=
    "/assets/sasads.css"
  upArrowImage="/assets/up_03b.gif"
  rightArrowImage="/assets/right_03b.gif"
  leftArrowImage="/assets/left_03b.gif" >

  <sasads:MDNavigationBar
    doubleLeftArrowImage=
     "/assets/double_left_03b.gif"
    doubleRightArrowImage=
     "/assets/double_right_03b.gif"
    leftArrowImage="/assets/left_03b.gif"
    rightArrowImage=
     "/assets/right_03b.gif"
    disabledDoubleLeftArrowImage=
     "/assets/double_left_03g.gif"
    disabledDoubleRightArrowImage=
     "/assets/double_right_03g.gif"
    disabledLeftArrowImage=
     "/assets/left_03g.gif"
    disabledRightArrowImage=
     "/assets/right_03g.gif" />
</sasads:MDTable>
```

Open the property sheet for MDBar1, specify
1. MDCommandProcessor1 for the commandProcessor property
2. MDModel1 for the Model property.

Close the property sheet.

To complete the JSP page we need to add a few HTML tags to complete the HTML. The complete index.jsp file is shown on the following page. The added HTML tags are shown in bold.

```
<%-- Copyright (c) 2001 by SAS Institute Inc., Cary, NC 27513 --%>
<%@taglib uri="http://www.sas.com/taglib/sasads" prefix="sasads"%>

<sasads:Connection id="D2159_s_PC" serverArchitecture="PC" persistedName="D2159's PC"
  command="sas.exe -dmr -comamid tcp -noterminal -cleanup" host="D2159"
  scope="session"  />

<sasads:MDModel id="MDModel1" connection="D2159_s_PC" scope="session" metabase="sugidemo"
    database="SUGIDEMO.CBMDDB" >
  <sasads:MDRowAxis >Product Hierarchy</sasads:MDRowAxis>
  <sasads:MDColumnAxis >Time Hierarchy</sasads:MDColumnAxis>
  <sasads:MDMeasure measure="Actual" selectedStatistics="SUM" />
  <sasads:MDMeasure measure="Forecast" selectedStatistics="SUM" />
  <sasads:MDMeasure measure="Difference" selectedStatistics="SUM" />
  <sasads:MDTotal level="Product Group" label="Total" state="true" />
</sasads:MDModel>

<sasads:MDCommandProcessor id="MDCommandProcessor1" scope="session" />

<sasads:MDExportToExcel id="MDExportToExcel1" formName="exportForm" formAction="index.jsp"
  render="false"/>
<html>
<head>
<link rel="stylesheet" type="text/css" href="/assets/sasads.css">
</head>
<body>
<sasads:MenuBar id="menuBar1" menuType="SELECTOR_EXPAND" separator="|" >
  <sasads:Menu label="Subset" image="/assets/subset.gif" alternateText="Subset" >
    <sasads:MDSelectorMenuItem model="MDModel1" selectorType="SUBSET_SELECTOR" />
  </sasads:Menu>
  <sasads:Menu label="Export To Excel" image="/assets/export.gif"       alternateText="Export
to Excel" customAction="document.exportForm.submit();" >
    <sasads:MDSelectorMenuItem selector="MDExportToExcel1" />
  </sasads:Menu>
</sasads:MenuBar>

<sasads:MDTable id="MDTable1" model="MDModel1" commandProcessor="MDCommandProcessor1"
  maxRows="25" maxColumns="10" scope="session" borderWidth="1"
  detailDataStyleSheet="/assets/sasads.css" upArrowImage="/assets/up_03b.gif"
  rightArrowImage="/assets/right_03b.gif" leftArrowImage="/assets/left_03b.gif">
  <sasads:MDNavigationBar doubleLeftArrowImage="/assets/double_left_03b.gif"
    doubleRightArrowImage="/assets/double_right_03b.gif"
    leftArrowImage="/assets/left_03b.gif"
    rightArrowImage="/assets/right_03b.gif"
    disabledDoubleLeftArrowImage="/assets/double_left_03g.gif"
    disabledDoubleRightArrowImage="/assets/double_right_03g.gif"
    disabledLeftArrowImage="/assets/left_03g.gif"
    disabledRightArrowImage="/assets/right_03g.gif" />
</sasads:MDTable>

<sasads:MDBar id="MDBar1" model="MDModel1" commandProcessor="MDCommandProcessor1"
  imageLocation="/assets/" scope="session" />
</body>
</html>
```

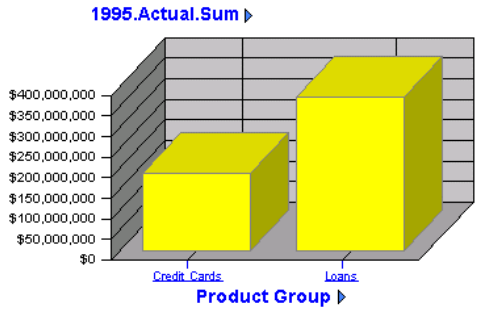
To test the JSP page from within webAF:

1. Make sure your web server is running by selecting Tools->Services->Start Java Web Server.
2. Select Build→Execute in Browser.

The resulting page should appear in the browser as shown.

Subset | Export To Excel

Year	1995			
	Actual	Forecast	Difference	Actual
Product Group	Sum	Sum	Sum	Sum
Credit Cards	$189,841,079	$210,916,460	$-21,075,381	$193,941,557
Loans	$375,606,799	$278,075,052	$97,531,747	$383,780,698
Total	$565,447,878	$488,991,512	$76,456,366	$577,722,255



DEPLOYING YOUR OLAP APPLICATION USING SERVLETS AND JAVASERVER PAGES

In most Web project development scenarios, multiple roles and responsibilities will exist. For example, an individual who designs HTML pages fills the role of a Web designer, while someone who writes Java code might fill a software development role. With the Web technologies available in J2EE -- namely JSP and servlets -- not only can you appropriately separate the development roles, you can also maintain a separation between your business logic and presentation code.

We have already demonstrated how you can use JSP technology for presentation purposes. Let's examine how one might separate some elements of the business logic from that presentation. To accomplish this, you can identify the components you need, then decide which technology to use to implement those components based on the role they play:

- Java servlets are well-suited to manage application flow and business logic evaluation. You can use servlets to provide a single point of entry into the Web application by performing such actions as intercepting HTTP requests that arrive from the client and simplifying security management.

- Presentation JavaServer Pages generate HTML (or other mark-up), and have as their main purpose the presentation of dynamic content.

The application can flow from the servlet to the presentation JSPs because the servlet can simply delegate or "forward" the incoming request -- plus any additional information returned by the business logic processing -- to the JSP. This separation, then, is analogous to the Model-View design pattern, where the front-end servlet functions as the model and the presentation JSP functions as the view. This approach is often presented as a "best practice" in implementing Web applications using J2EE.

Consider the OLAP application example, beginning with its Connection component. If you were to deploy the application within an organization that had many users, simple tasks such as connecting to SAS become much more costly because so many users could be accessing the application at the same time, which increases the load on the SAS server. To remedy this, your team would likely employ a server load balancing technology such as AppDev Studio Middleware. Or, you might direct users with specific roles or from specific departments to other SAS servers. The logic to implement the Connection component under such constraints is no longer appropriate within the JSP page. It would require embedding conditional Java scriptlets within the JSP or adding a significant number of attributes to the Connection custom tag. Instead, you can instantiate the Connection component from within a servlet. You've now begun to separate your business logic from the presentation code.

When determining how to separate the components of your application, you can apply a simple test: Does this component contribute to the rendering of the response? If it does, then you can leave it in the JSP. Otherwise, it is appropriate to move it into the servlet. The MDModel in our OLAP application fits this criteria. For example, by coding the metabase and multidimensional database information directly on the JSP as attributes of a custom tag (as we did in the previous demonstration), you place a reference to key information about your organization within the presentation. In addition, this metabase information may be data-driven itself -- or, based on your business logic, you provide different metabases for different user roles or scenarios -- and the presentation simply displays it.

In general, it is best to leave a Web designer with the responsibility of coding attributes on tags that simply render information. Any information related to organizational data or an organizational process is more appropriately included in the non-visual servlet.

Some other things to consider:

- webAF supports the development of both servlets and JSPs from within a single project. You can use File > New and specify a HTTP Servlet if you want to add a servlet file to a project.
- To forward a request from a servlet to a JSP, you use the code shown at the top of the following page.

Example of forwarding request from a servlet to a JSP:

```
RequestDispatcher dispatcher=getServletContext().getRequestDispatcher("jsp-file.jsp");
dispatcher.forward(request, response);
```

where *jsp-file*.jsp is the name of the presentation JSP.

The remainder of this section illustrates how the example we've been building can more appropriately be separated into servlets and JSPs. We've separated the business logic and application flow components (Connection, MultidimensionalTableV3Interface, CommandProcessor) and placed them in a servlet. The presentation component (MDTable) is placed in the JSP. To complete the example, you would also need to add the MDNavigationBar, MDBar, and Menubar custom tags to the JSP since they also represent rendering of visuals that are part of the application.

Servlet Example:

```
/* Copyright (c) 2001 by SAS Institute Inc., Cary, NC 27513 */
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import com.sas.rmi.Connection;
import com.sas.rmi.Rocf;
import com.sas.servlet.util.BoundConnection;
import com.sas.sasserver.mdtable.MultidimensionalTableV3Interface;
import com.sas.servlet.beans.mddbtable.html.*;
import com.sas.servlet.beans.mddbtable.MDCommandProcessor;
import com.sas.servlet.util.Util;
import com.sas.servlet.beans.mddbtable.commands.*;

public class servletSugi26
    extends javax.servlet.http.HttpServlet
{
    /**
     * Respond to the Post message.
     */
    public void doPost(javax.servlet.http.HttpServletRequest request,
            javax.servlet.http.HttpServletResponse response)
        throws javax.servlet.ServletException, java.io.IOException
    {
        //Get the session object for this user
        HttpSession session = request.getSession();
        if (session != null)
        {
            //Declare the Objects that will have Session Scope
            MultidimensionalTableV3Interface MDModel=null;
            MDCommandProcessor MDCommandProcessor1;

            //Get the connection object
            Connection olapConnection = (Connection)session.getValue("olapConnection");

            //If olapConnection is null then the user has not connected to the server.
            //Establish a connection and create the objects needed for application and put
            //the objects in the users session so they can be retrieved by future requests.
            if (olapConnection==null)
            {
                //Create the Connection Object, set the host to localhost, and
                //store it as a session object.
                olapConnection = new Connection();
                olapConnection.setHost("localhost");
                session.putValue("olapConnection",olapConnection);

                //Make this a bound connection so the SAS session will be
                //terminated when the http session times out.
                BoundConnection bc = new BoundConnection(olapConnection);
                session.putValue("bc",bc);

                //Create the Rocf Object and store it as a session Object
                Rocf rocf = new Rocf();
                session.putValue("rocf",rocf);

                //Create the MDCommandProcessor Object and store it in the session
```

```
                MDCommandProcessor1 =  new MDCommandProcessor();
                session.putValue("MDCommandProcessor1",MDCommandProcessor1);

                //Create the MDModel Object.
                MDModel =(MultidimensionalTableV3Interface) Util.newInstance(rocf,
                               olapConnection, MultidimensionalTableV3Interface.class);

                if (MDModel != null)
                {
                    //Store the MDModel as a session object
                    session.putValue("MDModel",MDModel);

                    //Create String arrays to initialize the MDModel.
                    String rows[]={"Product Hierarchy"};
                    String cols[]={"Time Hierarchy"};
                    String measures[]={"Actual","Forecast","Difference"};
                    String stats[]={"SUM"};
                    try
                    {
                        //Set up the MDModel
                        MDModel.setMetabase("sugidemo");
                        MDModel.setDatabase("SUGIDEMO.CBMDDB");
                        MDModel.setRowAxis(rows);
                        MDModel.setColumnAxis(cols);
                        MDModel.setSelectedMeasures(measures);
                        MDModel.setSelectedStatistics("Actual",stats);
                        MDModel.setSelectedStatistics("Forecast",stats);
                        MDModel.setSelectedStatistics("Difference",stats);
                    }
                    catch(com.sas.table.TableException te)
                    {
                       //If there is a problem, forward to an error page.
                       RequestDispatcher errPage =
                       getServletContext().getRequestDispatcher("/error.jsp");
                       errPage.forward(request, response);
                    }
                }
            }
            else
            {
                //If the Connection has already been made retrieve the session objects.
                MDModel = (MultidimensionalTableV3Interface)session.getValue("MDModel");
                MDCommandProcessor1 =
                  (MDCommandProcessor)session.getValue("MDCommandProcessor1");
            }
                RequestDispatcher dispatcher =
                  getServletContext().getRequestDispatcher("/{ProjName}/presentation.jsp");
                dispatcher.forward(request, response);
            }
    }

    /**
     * Respond to the Get message.
     */
    public void doGet(javax.servlet.http.HttpServletRequest request,
            javax.servlet.http.HttpServletResponse response)
        throws javax.servlet.ServletException, java.io.IOException
    {
        // Note: Add User DO_GET code here
        doPost(request,response);
    }
}
```

The request gets forwarded to the JSP.

JSP EXAMPLE:

```
<%-- Copyright (c) 2001 by SAS Institute Inc., Cary, NC 27513 --%>
<%@taglib uri="http://www.sas.com/taglib/sasads" prefix="sasads"%>

<html>
```

```
<head>
<title>OLAP Presentation JSP</title>
<!-- Include sasads style sheet for default style classes -->
<link rel="stylesheet" type="text/css" href="/assets/sasads.css">
</head>
<body>

<h2>OLAP Presentation Example</h2>

<!--Create the MDCommandProcessor, Connection and MDModel -->
<!--Locate the MDCommandProcessor tag at the top of the page-->
<!--because the execution of other tags depend on the executed command-->
<sasads:MDCommandProcessor scope="session" id="MDCommandProcessor1" />
<sasads:Connection ref="olapConnection" scope="session" />
<sasads:MDModel ref="MDModel" scope="session" />


<!--Create and write out the MDTable -->
<sasads:MDTable id="MDTable1" borderWidth="1" cellSpacing="0"
    maxColumns="8" maxRows="20" cellPadding="1"
       detailDataStyleSheet="/assets/sasads.css" commandProcessor="MDCommandProcessor1"
    scope="session" model="MDModel" />

</body>
</html>
```

CONCLUSION

AppDev Studio gives application developers a simple way to build OLAP applications that leverage the broad power of SAS on the server. It delivers powerful, easy-to-use OLAP reporting capabilities through

- WebEIS software, a rich 100% Java-based application, which enables you to build your OLAP applications without having to write any Java code. Documents built with webEIS can be easily deployed as applets or JSPs by simply making a choice off of a menu!
- webAF software which is a stand-alone, integrated Java development environment. Through its drag-and-drop interface you can quickly build applications, applets, servlets or JSPs. webAF also provides a rich class library that includes the foundation components for building OLAP solutions. These components are dedicated to delivering OLAP data on the web and utilizing SAS on the back end for both computation power and data storage.

By offering this all of this technology under one umbrella, AppDev Studio gives you the tools to build the right solution for your enterprise.

ADDITIONAL RESOURCES AVAILABLE

AppDev Studio Developer's Web Site

The AppDev Studio Developer's Web site is designed to help you develop and implement enterprise applications that use the power of SAS software to support information delivery and decision making.

The AppDev Studio Developer's Web site is continuously updated with new information, including comprehensive tutorials, how-to topics, and technical papers.

A snapshot of the AppDev Studio Developer's Web site is installed to your local Web server when you install AppDev Studio. You can always access the most current version of this site at www.sas.com/rnd/appdev/.

Training

SAS Institute offers a broad curriculum of instructor-based courses to help you use SAS software to meet your development goals with AppDev Studio. Courses cover a wide range of Web applications development, including:

- *SAS Web Tools: Accessing MDDB Data Using webEIS Software*
- *SAS Web Tools: Developing JSP Applications Using webAF Software*
- *SAS Web Tools: Understanding Java in webAF Applications*
- *SAS Web Tools: Overview of SAS Web Technology*
- *SAS Web Tools: Running SAS Applications on the Web*

Instructor-based training allows you the flexibility to attend courses in training facilities across the United States and in other countries. In addition, SAS staff can conduct on-site training. For more information on these and other courses, visit the SAS Training site at www.sas.com/training.

REFERENCES

SAS Institute Inc. (2000), *Getting Started with AppDev Studio, First Edition*, Cary, NC: SAS Institute Inc.

OLAP Tools and Techniques within the SAS System, A SAS White Paper written by John McIntyre, Mark Moorman, and Johnny Williams, SAS Institute Inc.

ACKNOWLEDGMENTS

Much of the content of this paper was pulled from various papers that are available on the AppDev Studio Developer's Site. We tried to give credit in the appropriate sections for those contributions, but we would like to thank all of the authors for the valuable information and useful examples that are available from that web site.

We'd like to personally thank several SAS developers for either review and/or writing of material for this paper: Angela Allen, Corey Benson, Scott Leslie, Rich Main and Marty Tomasi.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. You can contact the authors at:

Tammy Gagliano
SAS Institute Inc.
Work Phone: (317) 569-9598
Email: Tammy.Gagliano@sas.com

Tony Prier
SAS Institute Inc.
Work Phone: (913) 491-1166, x1364
Email: Tony.Prier@sas.com

WebHound: Your Best Friend for Web Traffic Analysis

Dean Duncan, School of Social Work, University of North Carolina -- Chapel Hill, NC
Frank Lieble, SAS, Orlando, FL
Carol Martell, Highway Safety Research Center, University of North Carolina – Chapel Hill, NC
Sally Muller School of Social Work, University of North Carolina – Chapel Hill, NC

The University of North Carolina at Chapel Hill has two state-chartered organizations engaged in e-campus and e-business initiatives. The Work First Group (WFG), in the School of Social Work, provides a web site that presents dynamic statistics and analyses about the welfare reform program, "Work First," in North Carolina. The Highway Safety Research Center (HSRC) develops project-specific sites for contracting agencies. These sites are as diverse as the projects themselves, ranging from a site designed to deliver specific information for college students, to sites created to increase public awareness. This paper presents two case studies on lessons learned from the experiences of implementing SAS WebHound™ at WFG and HSRC. We describe how the study of web traffic logs allows both organizations to understand how their users are using their web sites.

Introduction

The Internet has revolutionized how we access and retrieve information. It has touched all industries including commercial, government, and academia. The need to easily provide accurate information in a timely manner is more important today than it ever has been. The need to find solutions to meet this level of service is crucial in the success of these organizations.

SAS WebHound™ was used to analyze web logs at two University of North Carolina – Chapel Hill sites, the School of Social Work (SSW) and the Highway Safety Research Center (HSRC). We will discuss the need for web traffic analysis, the SAS WebHound solution, what issues it is addressing, and finally present two case studies documenting discoveries found from analyzing web logs using the SAS WebHound solution.

Why Web Traffic Analysis?

Web traffic analysis is crucial for any organization that has a web site. For example, commercial corporations or the private sector want to increase revenue and profitability, while government agencies, universities and colleges or the public sector want to provide services and information. Even though theses two sectors have different goals they do have one goal in common. They must be able provide the highest level of service to their users. This can be achieved by:

- Providing the most relevant information to the people visiting the web site.
- Improving communication via the intranet and extranet.
- Optimizing the buying process to maximize revenue.
- Promoting the information that is the most popular.
- Creating a convenient experience by organizing information more effectively on the site.
- Improving user satisfaction by reducing the number of clicks it takes to access information.

Web traffic analysis is the study of web user usage patterns. This will allow an organization to determine:

- Where are my web users coming from?
- What path do they take before finding the information they want?
- What path do they take before leaving?
- Which pages are the most popular?

- Which pages are the least popular?
- How many clicks did it take to find the information?

But before these questions can be answered a process and/or application is needed to capture, store, manage, analyze, and report web usage data.

SAS WebHound

SAS WebHound is a solution that incorporates SAS technologies to allow web analysts to understand web traffic usage within their organization's web site. With WebHound they can identify who is browsing their web site, what are they looking at, and how often they visit. WebHound also provides ease of use and flexibility by:

- Processing large volumes of web log data (scalability).
- Integrating web data with other data sources on-line and off-line.
- Providing dynamic reporting through any browser by providing accurate information about how users interact with the web site.
- Tracking visitor usage trends across time of day, weeks, months, and even years.

WebHound will allow an organization to measure and improve the effectiveness of their web site, which will provide the highest of service to their users.

WebHound Case Studies

The following are two case studies on lessons learned from the experiences of implementing SAS WebHound. The first case study, the School of Social Work, have never performed web traffic analysis at their site. The second, the Highway Safety Research Center, have been using other web analysis tools for web traffic analysis for the past year. Each case study will introduce their organization, describe their web environment, discuss their e-problem and solution implementation, and close with a conclusion.

Case Study 1:

School of Social Work – University of North Carolina at Chapel Hill

The North Carolina University - Chapel Hill (UNC-CH) School of Social Work (SSW) has 45 full-time faculty and more than 300 graduate level students. The School ranked 4th in the nation, according to the 2000 U.S. News and World Report survey of social work at public universities and 7th overall of 140 graduate schools of social work. At UNC-CH, the School is ranked third in externally funded projects. The School's mission is to provide an interface between the physically and economically disenfranchised people of North Carolina and the government, non-profit, and private foundations of North Carolina. The School's site (http://ssw.unc.edu) hosts 12 individual programs: such as the Work First project. The Work First's Web site (http://ssw.unc.edu/workfirst) provides longitudinal (i.e. historic) statistics on more than 335,000 families and 795,000 individuals who have received assistance through welfare (i.e. Work First) since January 1995. This SAS/IntrNet web site was developed by Dr. Dean Duncan and his staff in 1998 to provide performance measures to county Departments of Social Services (DSS) to administer the Work First program. Also it enables DSS staff, social workers, researchers, policymakers, and the public to access analyses regarding welfare reform.

Web Environment

SSW's web site (Figure 1) consists of three web servers: two of which are HP Netserver 5 servers running Windows NT 4.0 Service Pack 4. Respectively, installed with 180 MHz and 200 MHz processors, attached to 12 and 32 GB RAID drives. Both servers have 128 MB of RAM. The third server is a HP Netserver LX-Pro running Windows 2000. Installed with a 600MHz processor, 640 MB of RAM, 2 20 GB RAID drives. Two servers collect IIS web logs and one collect Netscape web logs.

The SSW web sites are distributed across all three machines, but each individual site is

only on one machine. SSW decided that, rather than include all of their web logs in one WebHound report, for each web server they would essentially run a separate WebHound application. Thus, each web server has it's own URL for the WebHound reports. This case study will examine web usage logs collected for six weeks from January 2001 to February 2001.

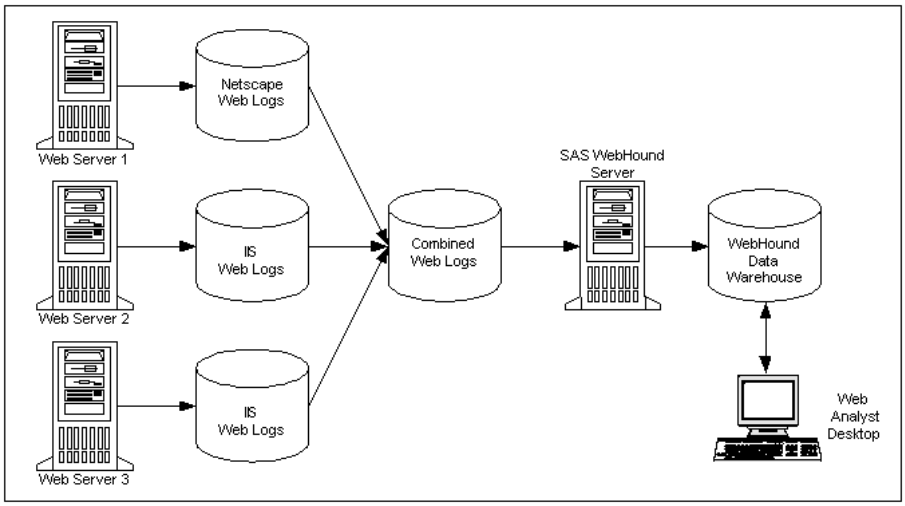


Figure 1

e-Problem

In 1999, the SSW faculty realized that their School's web site was the first place people went for information about the School, and so the dean formed a committee to look at the site. Their consensus was that the web site could make a significant contribution toward achieving the School's mission, provided:

- There are clear goals and valid measures for tracking progress.
- Faculty and students have easy access to computers, software, and the Internet.
- Faculty know how to use the technology effectively.

A member of the committee and director of the Computing Information and Technology Unit, Laura Zimmerman, Ph.D. said, "We want to implement a look and content that appeals to our students and faculty as well as potential students, researchers, faculty from other campuses, and the media." The School encourages faculty to use the web site to promote their research and courses. And because the site is often the first entry point for alumni and constituents, SSW administrators, such as the Director of Alumni Relations, increasingly are using the web site.

Just when the School's faculty was voicing concerns that students and researchers who visited the SSW site could not easily find what they needed, Duncan was considering ways his group could determine if their Work First web site was effective.

Duncan and his staff decided that in order to improve their Internet presence it would be necessary to identify the purpose of their site and then identify valid measures for tracking their progress. They identified the mission of the Work First web site as a vehicle for easy access to statistics regarding Work First recipients for DSS staff, social workers, researchers, policy makers, program leaders, and the public.

The measures Duncan and his staff identified for tracking progress:

- Can visitors, who know what they are searching for, find it?
- If visitors know where to go to find the information, is it still too difficult to navigate?

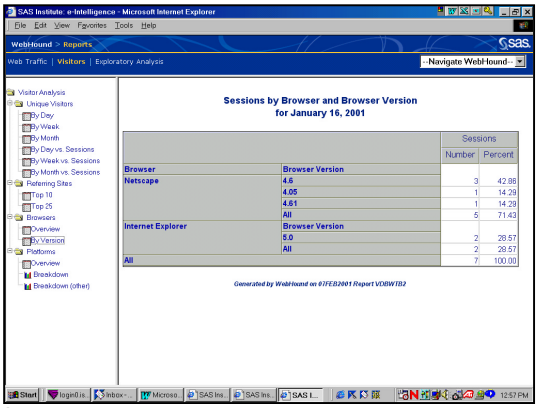
- Where are visitors coming from and going to and which links to the site are the most popular?
- Are pages that we promote actually the pages that are visited the most?
- Are the visitors who come to the site, the visitors that we expect?
- Are our web pages being developed in a cost-effective manner?
- Is the impact of our web pages on the School's computer minimal?

Solution Implementation

In January 2001, SAS consulting staff installed and configured the WebHound solution at the School of Social Work. A Windows NT AT process was created to archive the web log files collected by Expanded Log Format (ELF) each night. They were then copied from the three NT web servers to a fourth NT server on which SAS WebHound was installed (Figure 1). After WebHound processes the logs, reports are generated back to one of the three web servers where they can then be accessed via any web browser. The reports produced by WebHound at this point are all static. Dynamic reports are also produced using SAS/IntrNet features of WebHound. These applications are submitted to a broker, which resides on one of the three web servers. During the implementation, decisions had to be made as to where to put various pieces of the process and also how to surface the resulting reports.

As soon as the reports became available, we began investigating which of the reports provided measures for tracking progress toward achieving web site goals.

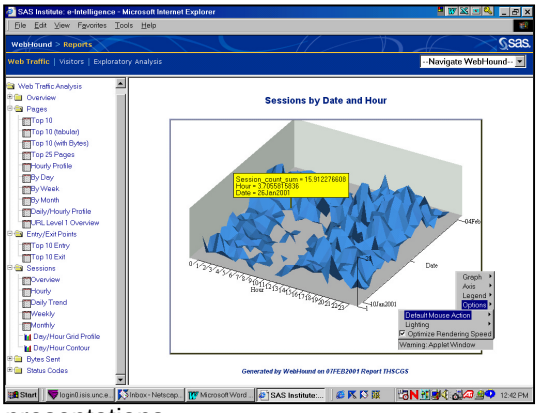
Duncan discovered that the WebHound report, "Browsers By Version," (Figure 2), could provide him with another source of information regarding his client's use of IT. Duncan can use this information in several ways, including testing new releases of the web site by using the same browsers and browser versions that visitors use. This was salient because last year Duncan got a call from the NC Division of Social Services that a staff member was having trouble accessing the Work First site. After investigating, Duncan discovered the problem was the version of the staff's



browser.

Figure 2

As the Principle Investigator of the Work First project, Duncan is also interested in the information provided by the "Sessions -- Day/Hour Contour" report (Figure 3). From this report, Duncan can determine the times of the day when usage peaks. He can examine those days when his staff have issued announcements or given presentations about the site and determine the impact of these announcements and



presentations.

Figure 3

In justifying resource allocation to funding agencies, it is helpful to be able to substantiate projections with statistics such as the actual number of web visitors to the site (Figure 4).

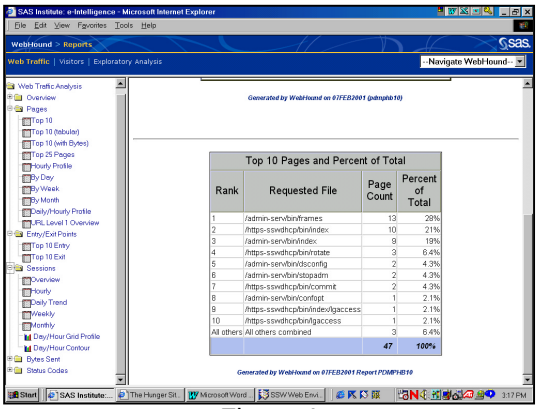


Figure 4

Another measure Duncan selected to track was the impact of the Work First site on the School's web server. Two of the School's system managers, Andy Broughton, Ph.D. and Manuel Garcia, were consulted. The system managers agreed that the reports they needed most were those that would allow them to identify and track "trends" in system usage. Only by identifying trends can they make informed decisions about the need for system expansion. They found that the "Sessions By Month," report (Figure 5), provides the kind and level of information that they are looking for: a summary of total number of web sessions over months. At the time of this writing, only six weeks of data were available. However, as new months are added, the system managers will be able to detect patterns of usage and use this information for capacity planning.

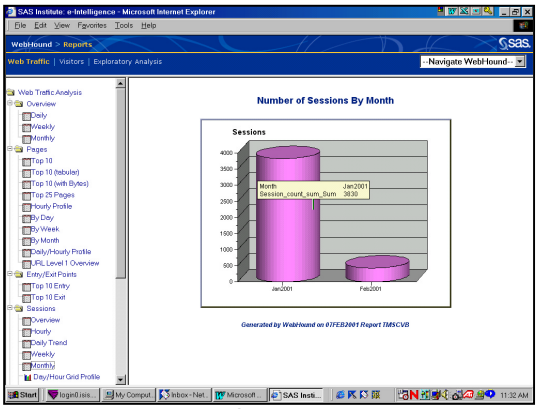


Figure 5

The system managers also found the report, "Sessions Day/Hour Grid Profile," helpful for identifying patterns of usage (figure 6). The report provides information regarding the total number of sessions for any particular hour on any particular day, for as many weeks as were specified in the WebHound configuration. With this information, the system managers can correlate NT performance measures on memory, disk, processor, and network usage with these measures of total number of sessions for specific date/time combinations. Note that the report allows for "rubber banding" a piece of the report may be selected for further analysis.

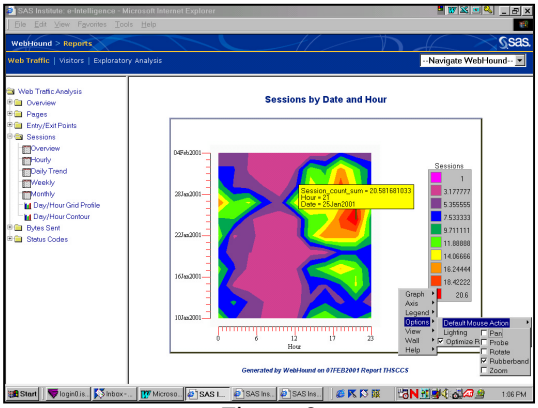


Figure 6

Conclusion

Using results "straight out of the box", the web server system managers, project director, and IT director were each able to immediately find reports appropriate to their areas of interest. The system managers have a finger on the pulse of system parameters. The project director can see what users respond to his announcements, he has documentation to justify funding, and he knows the range of browser versions for future development parameters. The IT director can learn which areas of the sites are most popular and plan future development to use the most appealing approaches. This is just the tip of the iceberg. They will next explore: 1) removing in-house traffic, 2) using the TreeView Applet to examine pathing, and 3) examining the times spent on individual pages. WebHound has already provided leads that have allowed SSW and the Work First Group to address a multitude of problems, issues, and opportunities. Now they need only follow these leads to obtain the information that they are seeking.

Case Study 2:

Highway Safety Research Center – University of North Carolina at Chapel Hill

The UNC Highway Safety Research Center (HSRC) conducts interdisciplinary research aimed at reducing deaths, injuries and related societal costs of roadway crashes in North Carolina and the nation. Our research addresses crashes involving motor vehicles, bicyclists and pedestrians, and takes into account the various human, vehicular, roadway and environmental components of these risks. HSRC strives to translate developed knowledge into practical interventions that can be applied at local, state, national and international levels. While public service announcements, posters and printed documents are still important ways of sharing life-saving transportation safety messages, the web offers an explosion in capabilities for marketing our research and outreach projects.

Web Environment

HSRC is, at the time of this writing, bringing online a Sun Enterprise 250 with 1GB of RAM, one 400MHz UltraSparc-II processor and three 18 GB drives to consolidate web and application server needs, which had been distributed across in-house and UNC campus servers (Figure 7). Other in-house Sun servers are an Ultra 5, an Ultra 1 and a SparcStation 5. This case study will examine logs collected in December of 2000 from a Sun Ultra 5, serving at the time as the Center's Apache web server. Virtual sites are not configured to log separately, so a single log file contains all entries. The log is archived out twice a day. DNS lookup has already been performed for these logs.

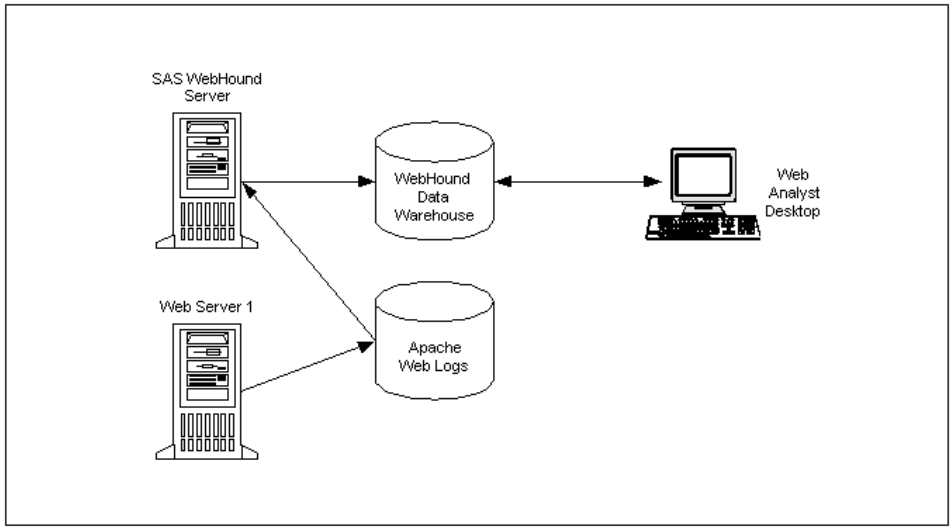


Figure7

e-Problem

Web traffic statistics have been implemented at HSRC using other software for more than a year. These numbers have allowed us to examine traffic volumes, but these static reports cannot answer all our questions. Each new web project has design strategies influenced by lessons learned in earlier projects. The web development team is extremely talented but small. Efficiency, not a new concept, must be applied to this relatively new environment.

Solution Implementation

Archived log files were transferred via FTP from the Ultra 5 to the E 250, which will serve both to process the logs and to serve the results to the web. After the initial installation and test run of WebHound, we immediately wanted to incorporate the virtual site name as a variable, and to

eliminate all in-house traffic. Since the logs were collected using the default Common Log Format (CLF), the virtual site name was not part of the logging information. Also, since virtual hosts were not configured to log separately, the virtual host could not be determined from the source file.

Our site name workaround was to employ operating system utilities to parse out the log files. We "grep'd" the logs by writing each virtual site to a separate file. The first level of each resulting filename was assigned the value we wished to see in the reports to represent the virtual site. For example, this command:

```
cat logfiles | grep www.hsrc.unc.edu > hsrc.log
```

creates a file containing every log entry involving the main HSRC virtual site. In order to use the filename to assign a value to the variable SITENAME, we added code to a catalog in the USERMODS library. USERMODS is employed to house override code for a WebHound data store. The variable SITENAME already exists in WebHound, so we were populating an existing variable with values rather than adding a new variable.

USERMODS was also the appropriate place to delete log entries from in-house users. WebHound already has a facility for ignoring traffic from of a range of IP addresses. We could not use the facility, however. These logs no longer had IP addresses since DNS lookup had already been performed. The following code, placed in entry USER_ASSIGNMENTS_AFTER_INPUT in the USERMODS.WBETL catalog performs both customizations:

```
if index(client_id,'.hsrc.unc.edu')>0 then delete;
 length _sitetmp $200;
 _sitetmp = scan(File_Name, -1, "/");
 _sitetmp = scan(_sitetmp, 1, ".");
 sitename = _sitetmp;
```

Our next task was to make SITENAME available in all the Exploratory Analysis groups. The SAS table WBCROSS contains all the class variables that will be used to create the MDDBs for the Exploratory Analyses. We found SITENAME in two groups, and added entries for the remaining groups (Figure 9)

	table_prefix	VariableName
62	referrers	Date
63	referrers	Week
64	referrers	Month
65	browsers	Browser_Version
66	browsers	Browser
67	browsers	Platform
68	browsers	Date
69	browsers	Week
70	browsers	Month
85	pagesviewed	sitename
86	urldirectory	sitename
87	visitors	sitename
88	client	sitename
89	referrers	sitename
90	statuscodes	sitename
91	browsers	sitename

Figure 9

We could have stopped at this point and had SITENAME available in all the interactive table groups. We wanted, however, to change some of the hierarchical groupings available in those tables. Again, we used override code in the USERMODS library.

Figure 10 shows the original code creating the Top Pages Exploratory Analysis.

```
%let mddbrc=;
%Create_MDDB_from_Summary_DS (
   summary_ds  = summary.pagesviewed_1_2,
   mddb        = mddbs.pagesviewed,
   mddb_label  = Webhound Pages Viewed MDDB,
   hierarchy1  = ,
   drillhier1  = Requested File->Status Code: Re
   drillhier2  = Requested File->Referrer Domair
   timehier1   = Week->Date: week date ,
   filtervars  = Status_Code File_Type ,
   repository  = &repository,
   nuniques    = ,
   nunique_labels = ,
   detail_dataset = detail.weblog_detail_1,
   _rc         = mddbrc
   );
%put NOTE: Return code from Create_MDDB_from_sum
```

Figure 10

In our situation, file structures and names are often duplicated across sites, so without SITENAME, the requested file is ambiguous. Consequently, we modified the hierarchical definitions as seen in Figure 11. A new hierarchy replaces drillhier2 and drillhier3 is the same as the original drillhier2 with SITENAME inserted as the first level.

```
%let mddbrc=;
%Create_MDDB_from_Summary_DS (
   summary_ds  = summary.pagesviewed_1_2,
   mddb        = mddbs.pagesviewed,
   mddb_label  = Webhound Pages Viewed MDDB,
   hierarchy1  = ,
   drillhier1  = Requested File->Status Code: Re
   drillhier2  = Site Name->Referrer Domain: Sit
   drillhier3  = Site Name->Requested File->Refe
   timehier1   = Week->Date: week date ,
   filtervars  = Status_Code File_Type ,
   repository  = &repository,
   nuniques    = ,
   nunique_labels = ,
   detail_dataset = detail.weblog_detail_1,
   _rc          = mddbrc
   );
%put NOTE: Return code from Create_MDDB_from_sum
```

Figure 11

To first examine the enhancement introduced by adding SITENAME to the WBCROSS table, we examine Top File Types. Suppose we'd like to know how the heavy use of graphics affects our traffic. The initial report is seen in Figure 12.

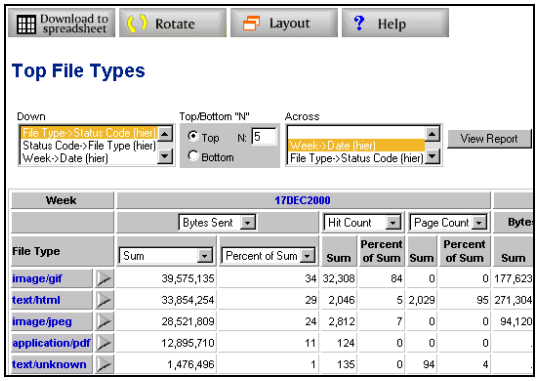


Figure 12

What we see is file type by week (graphic is truncated here). The boxes between the title and the table indicate that the 'down' variable is hierarchical: 'File Type->Status Code (hier)'. The other boxes indicate that the top 5 file types are displayed and that the 'across' variable is also hierarchical: 'Week->Date'. Clicking on the date causes the table to refresh with each day of that week appearing separately across the table. The hierarchy of the 'down' variable may be explored in two ways. Clicking on the arrow to the right of a file type causes the second level of the hierarchy to appear (Figure 13), whereas clicking on the file type itself drills down to a table of only that file type (Figure 14)

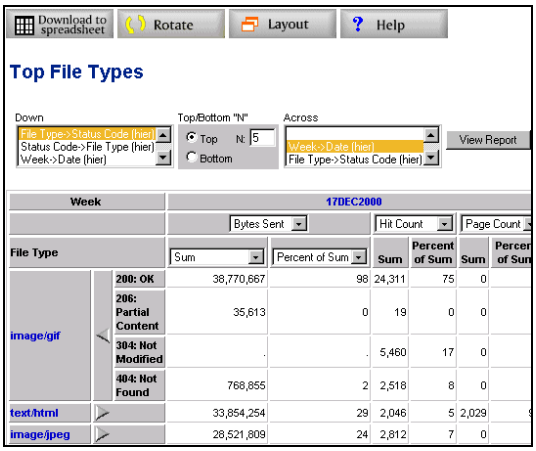


Figure 13

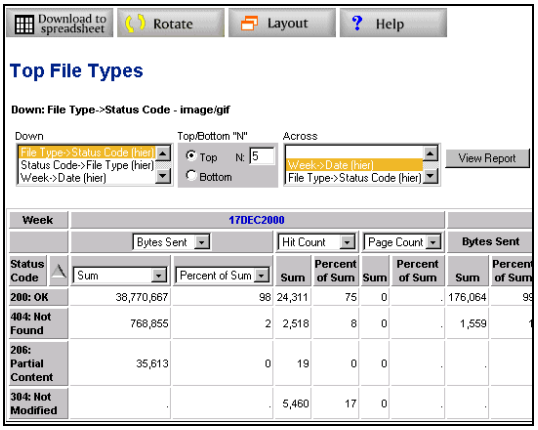


Figure 14

The Layout button provides complete control over the report. Every variable in WBCROSS for the group as well as the hierarchies defined in MDDB creation are available. Filter variables allow the end user browsing to subset based on filter variable values (Figure 15).

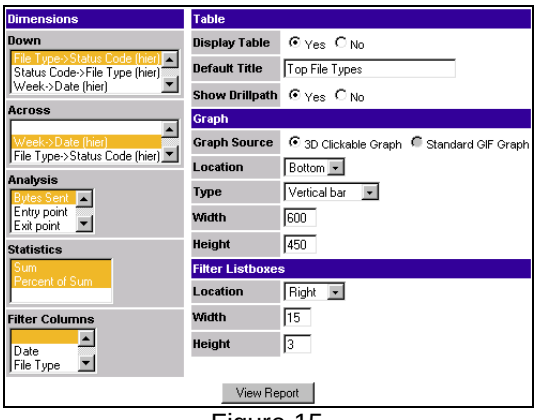
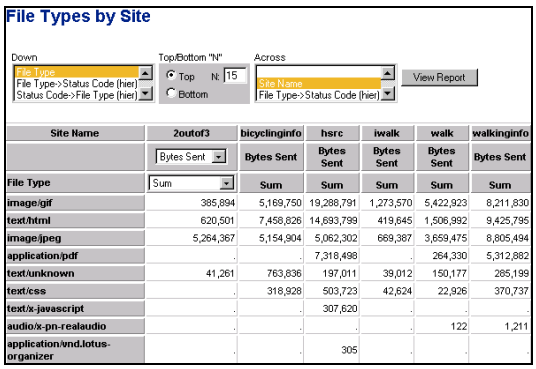


Figure 15

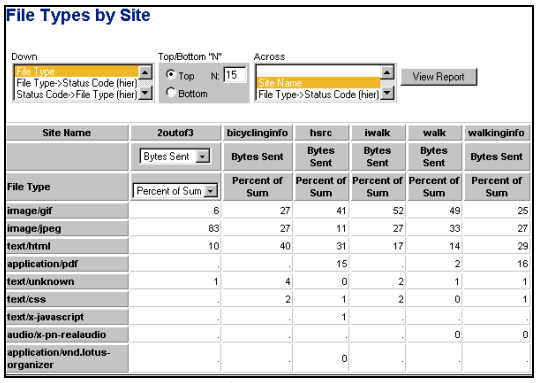
Changing the configuration to reveal bytes sent for file type by site generates Figure 16.



Site Name	2outof3	bicyclinginfo	hsrc	iwalk	walk	walkinginfo
	Bytes Sent	Bytes Sent	Bytes Sent	Bytes Sent	Bytes Sent	Bytes Sent
File Type	Sum	Sum	Sum	Sum	Sum	Sum
image/gif	385,894	5,169,750	19,288,791	1,273,570	5,422,923	8,211,830
text/html	620,501	7,458,826	14,693,799	419,645	1,506,992	9,425,795
image/jpeg	5,264,367	5,154,904	5,062,302	669,387	3,659,475	8,805,494
application/pdf	.	.	7,318,498	.	264,330	5,312,882
text/unknown	41,261	763,836	197,011	39,012	150,177	285,199
text/css	.	318,928	503,723	42,624	22,926	370,737
text/x-javascript	.	.	307,620	.	.	.
audio/x-pn-realaudio	.	.	.	.	122	1,211
application/vnd.lotus-organizer	.	.	305	.	.	.

Figure 16

Changing 'Sum' in the select box in the table to 'Percent of Sum' yields Figure 17. Seeing that 89% of the bytes sent for the 2outof3 site are gif or jpg files, which represent 58% of our traffic (Figure 18) would make it seem a byte hog were it not revealed in Figure 19 that 2outof3 represents only 5% of the traffic.



Site Name	2outof3	bicyclinginfo	hsrc	iwalk	walk	walkinginfo
	Bytes Sent	Bytes Sent	Bytes Sent	Bytes Sent	Bytes Sent	Bytes Sent
File Type	Percent of Sum	Percent of Sum	Percent of Sum	Percent of Sum	Percent of Sum	Percent of Sum
image/gif	6	27	41	52	49	25
image/jpeg	83	27	11	27	33	27
text/html	10	40	31	17	14	29
application/pdf	.	.	15	.	2	16
text/unknown	1	4	0	2	1	1
text/css	.	2	1	2	0	1
text/x-javascript	.	.	1	.	.	.
audio/x-pn-realaudio	.	.	.	.	0	0
application/vnd.lotus-organizer	.	.	0	.	.	.

Figure 17

	Bytes Sent
File Type	Percent of Sum
image/gif	34
text/html	29
image/jpeg	24
application/pdf	11
text/unknown	1
text/css	1
text/x-javascript	0
audio/x-pn-realaudio	0
application/vnd.lotus-organizer	0

Figure 18

	Bytes Sent
Site Name	Percent of Sum
hsrc	40
walkinginfo	27
bicyclinginfo	16
walk	9
2outof3	5
iwalk	2

Figure 19

To illustrate the success of the hierarchy modification, we see in Figure 20 that SITENAME appears as a first level in two of the available hierarchies for the Top Pages reports.

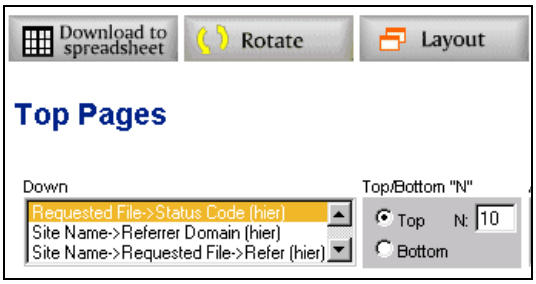


Figure 20

Conclusion

SAS WebHound Exploratory Analysis tools give us access to a frontier of information that was not provided in our other web analysis tools. Our job is to ask questions. When the answers raise more questions, it is easy to dig around for those answers as well. We have only begun to explore the information available using this e-tool. There are two areas we will pursue but have not addressed in this paper: customization of static reports to automate virtual site-specific information and implementation of the TreeView Applet, which visually presents click stream data.

Summary

The Internet has transformed how corporations, government, and academia conduct day-to-day business. The common goal of these organizations is to provide the web user with the highest level of service

possible. To achieve this web traffic analysis must be performed. SAS WebHound provides the ability to capture, store, manage, analyze, and report web usage data.

The School of School of Social Work (SSW) and the Highway Safety Research Center (HSRC) had the need to analyze web logs to improve the level of service for users visiting their web sites. They both installed SAS WebHound on their web site servers and started to analyze six weeks of web log data. SSW, who never has analyzed their web site, is now able to access web usage information they were not able to access before. This information will allow them to predict user impact on hardware resources and justify resource allocation to funding agencies. HSRC, who has analyzed their web site with other web analysis tools, discovered SAS WebHound exploratory analysis tools could provide additional information about their web site and user patterns. They now have the flexibility to determine how user traffic will affect their web site. These discoveries have provided new information for both SSW and HSRC to improve the level of service for their web users.

Acknowledgements

The authors thank the following people for their contribution to this paper:

Andy Broughton, Ph.D., Manual Garcia, Harvey Hou, Michael Ingraham, Andy Parks, Stephen Schultz, Christian Valiulis, Jia Xu, and Laura Zimmerman, Ph.D.

Contact Information

Your comments and questions are encouraged. Contact the authors at:

Dr. Dean Duncan
UNC School of Social Work
301 Pittsboro Street, CB# 3550
Chapel Hill, NC 27599-3550
919-962-7897
Dean_Duncan@unc.edu

Frank Lieble
SAS Institute Inc.
Orlando Regional Office
1035 Greenwood Blvd., Suite 465
Lake Mary, FL 32746
407-804-1995 x237
Frank.Lieble@sas.com

Carol Martell
UNC Highway Safety Research Center
730 Airport Road, CB# 3430
Chapel Hill, NC 27599-3430
919-962-8713
Carol_Martell@unc.edu

Sally Muller
UNC School of Social Work
301 Pittsboro Street, CB# 3550
Chapel Hill, NC 27599-3550
919-843-7798
sally@email.unc.com

Energizing End Users with a Slice of SAS® and a Cup of Java™

Randy Curnutt, Solutions Plus, Inc., Indianapolis, IN
Michael Pell, Solutions Plus, Inc., Indianapolis, IN
John LaBore, Eli Lilly And Company, Indianapolis, IN

ABSTRACT

Many corporate information systems have evolved to integrate a diverse mixture of hardware platforms. Although commonplace, the mainframe has been joined by a plethora of UNIX and NT servers as well as an army of personal computers often connected by Local Area Networks (LAN) or Wide Area Networks (WAN). Further, software tools in this environment are diverse. Users now wish to analyze their mainframe data with PC and UNIX tools in addition to using mainframe tools. It is often necessary to migrate data to new platforms because some tools require local data access. Users may become intimidated and stymied by the prospect of finding an easy and reliable method for moving mainframe data to remote platforms. We explore how the flexibility of SAS system architecture, combined with Java programming capabilities, provides instant access in an easy-to-use tool for point-and-click data migration and conversion (OS/390 SAS, DB2 views, Oracle views) to a PC or UNIX/NT servers. We also discuss methods to automatically convert the resulting dataset to the desired format (SAS dataset, SAS transport file, Excel, CSV, tab delimited, space delimited, or dBASE) on the target platform. Our tool implements a verification process that ensures integrity of transferred data. Additionally, meta-data is available before and after the transfer.

INTRODUCTION

BUSINESS PROBLEM

The DATAccess Tool facilitates clinical data analysis by providing an easy way to transport and convert SAS data on the OS/390 mainframe to the platforms and formats desired for analysis. Instead of learning mainframe or communications programming commands, or waiting for a systems analyst to move the data for them, the user is free to focus on analyzing the data.

CONSTRAINTS

The budget for the project was limited to purchasing a Java integrated development environment and leveraging other existing tools already in house, such as PC SAS. The application had to adhere to all existing security policies across all platforms.

GOALS

The goal of the DATAccess Tool was to provide an intuitive graphical user interface that gave non-technical users a simple and repeatable process for copying (and converting) data from any platform to any supported target platform. Data integrity was of the utmost importance, especially since this tool may be used to create SAS Transport files that are submitted to the regulatory agency for review. The DATAccess Tool facilitates the verification process to ensure the transferred data is accurate. Since the tool brought together so many disparate technologies and platforms, it was imperative that it go through a rigorous validation process to ensure that it functioned as desired. It was also important for the tool to provide instantaneous transfers since the users had previously experienced lengthy delays waiting on technical analysts to move and convert the data for them.

The DATAccess tool allows the user to select test or production data, provides a list of research project codes for the selected environment. It then queries OS/390 to build a list of SAS libraries that the user is permitted to access for the selected research project code. Then a user can select a library and request that the tool identify the datasets available within that library (this generates another query to the OS/390). To transfer a dataset, the user selects a target host, a directory location, and a file format, then clicks a "Transfer" button.

ENVIRONMENT

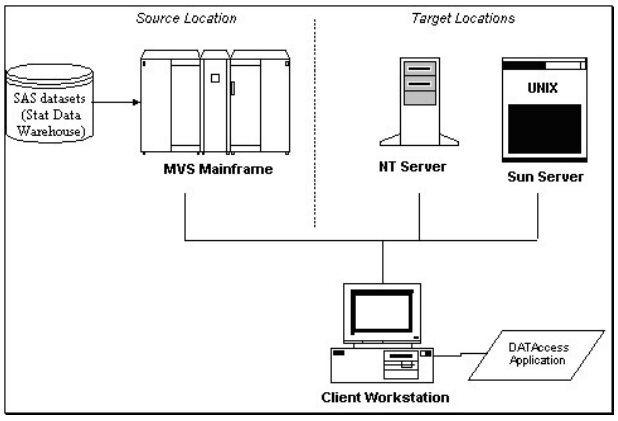
Our environment is very diverse, both from a user perspective as well as from a technological standpoint. The end-users span a very broad base; from non-technical to very technical. This user base includes statisticians, systems analysts, data analysts, research scientists, and physicians; these staff are located at both US and overseas locations.

The hardware encompasses an OS/390 mainframe, Sun Solaris systems, Windows NT servers and Windows based client machines.

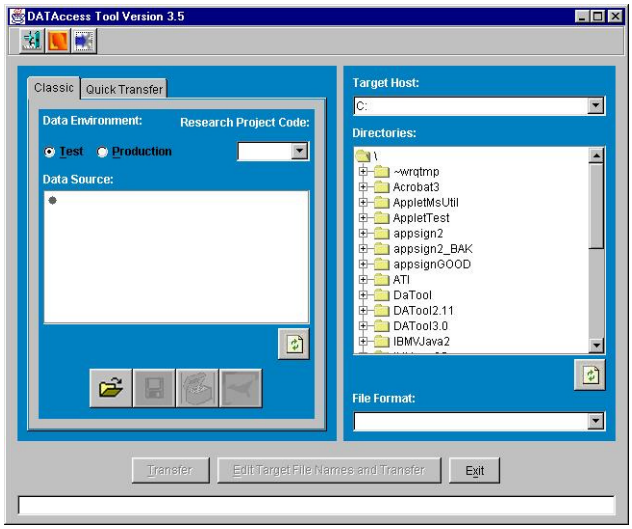
The software environment is comprised of a Java applet, Windows NT server, Windows 95/98/2000/NT clients, PC SAS 6.12, UNIX SAS 6.12 and OS/390 SAS 6.09, JCL, FTP. The data is a mixture of SAS, SAS Transport, DB2, Oracle, CSV, dBase, tab delimited text files, and Excel. Communications techniques utilize both SAS/Connect® and FTP. The DATAccess Tool also generates and submits OS/390 JCL jobs, as well as dynamically generating SAS programs that run on the most appropriate platform via SAS/Connect (i.e., the Windows client, OS/390, UNIX, or NT Server).

SOLUTION

SAS and Java provide the glue necessary to tie all of the various technologies together. Architecturally, the DATAccess Tool is viewed as a pipe that connects a source to a target, and manages the transfer and conversion process. Realistically, the initial design was for the source to always be SAS data residing on an OS/390 platform, but we did not wish to architect to such a rigid standard. The basic idea is that it shouldn't matter what platform or data structure resides at either end of the pipe. The following diagram depicts the physical architecture for the tool:



The user interface is developed with Java. The code that was developed can be deployed as either a stand-alone application or a Web based applet.



Based on the user's selections, Java is used to dynamically generate PC SAS, OS/390 SAS, UNIX SAS programs or OS/390 JCL jobs to accomplish the data transfer and conversion. The tool submits the generated program to the appropriate SAS engine depending on the client and server configuration. For example, if the user has PC SAS, then most programs generated by the application would leverage the local SAS software. If the user does not have PC SAS, then all generated programs are submitted to OS/390 SAS, which would make use of SAS/Connect as needed. If the user desires a transfer to the UNIX platform, this would always be accomplished by generating an OS/390 SAS program that runs SAS/Connect on the OS/390 to spawn a job on the desired UNIX host.

Generating SAS code for multiple platforms is relatively easy due to the fact that SAS code is highly portable across platforms.

We have included 3 code samples in this paper. The first is an example of a Java method to generate a SAS program. The output of the Java method is a String of text that is the SAS program. The second is an example of a SAS program that was created by Java, and the third is an example of a Java method that accepts the String of text, writes the String to a temporary file, then invokes PC SAS using the new file as a parameter to the sas.exe.

The following Java code sample is an example of using Java to generate a SAS program:

```
/** Builds a PC-SAS program to transfer
* the requested files.
* The PC-SAS program will connect to the
* MVS host using the
* user's username and password (as
* gathered from the Password dialog).
*/

public String buildPcSasProgram(String
sourceLib, String sourceDataset,
String targetHost, String targetDirectory,
String destinationFileFormat,
String targetFileName, String username,
String password,
String rowConstraints)
{

     StringBuffer strBuf = new StringBuffer();

     strBuf.append("%macro scon2Win(userId”);
     strBuf.append("=, pswrd=);\r\n");
     strBuf.append("options comamid=tcp “);
     strBuf.append("remote=mvs");
     strBuf.append("set=tcptn3270 1;\r\n");

     strBuf.append("signon "C:\\SAS\\”);
     strBuf.append("connect\\saslink\\”);
     strBuf.append("datcptso.scr\";\r\n")
     strBuf.append("%let userId=&userId; ");
     strBuf.append("\r\n”);
     strBuf.append("%let pswrd=&pswrd;\r\n");

     strBuf.append("rsubmit;\r\n");

     strBuf.append("libname srcLib '");
     strBuf.append(sourceLib);
     strBuf.append("' DISP=SHR;\r\n\r\n");

     strBuf.append("proc sql;\r\n");
     strBuf.append("create table WORK.");
     strBuf.append(sourceDataset);
     strBuf.append(" as\r\n");
     strBuf.append(" select * from srcLib.");
     strBuf.append(sourceDataset);
     strBuf.append("\r\n");

     //  Do the actual download.
     strBuf.append("\r\n\r\n");
     strBuf.append("proc download “);
     strBuf.append("data=WORK.");
     strBuf.append(aDataSetName);
     strBuf.append("\r\n");
     strBuf.append("    out=WORK.");
     strBuf.append(aDataSetName);
     strBuf.append(";\r\n run;\r\n");
(the rest of this method was not included)
```

Below is a simple example of what the output of the Java code above might look like follows (note: this is not showing the exact output of the method shown above). Also, to fit in the columns, the formatting had to be altered in a few places.

```
%macro scon2Win(userId=, pswrd=);
options comamid=tcp remote=mvs
     set=tcptn3270 1;

signon "C:\SAS\connect\saslink\datcptso.scr";
%let userId=&userId;
%let pswrd=&pswrd;
libname targDir 'C:\temp\';
```

```
/* Beginning of LIB_ABC.sd2 transfer  */

rsubmit;
libname srcLib 'LIB_ABC' DISP=SHR;

proc sql;
create table WORK.LIB_ABC as
    select *
    from srcLib. LIB_ABC;

proc download data=WORK.LIB_ABC
    out=targDir.LIB_ABC;
run;
/* validate the transferred data */
PROC upload data=targDir.LIB_ABC
out=WORK.xfer;
run;
proc compare  base=WORK.LIB_ABC
 compare=WORK.xfer
        outstats=WORK.vallog;
run;
proc download data=WORK.vallog
    out=WORK.vallog;
run;
endrsubmit;
/*  Now put the output of the Contents to the
Validation log file. */
proc contents data=targDir.LIB_ABC;
run;

PROC PRINTTO;
run;
RSUBMIT;
/* The following bit of code is used to
record this transfer on the OS/390 host
for reporting purposes. */
FILENAME SUB 'MYDIR.PDS.CNTL' DISP=OLD ;
DATA _NULL_;FILE SUB(DATMETR);
 PUT @1 "//MYJOB  JOB (,8305,S,MYUSERID),   "
/
     @1 "//
'anytexthere',CLASS=A,PRTY=10,MSGCLASS=T " /
     @1 "//STEP2   EXEC PGM=IEFBR14   " ;
RUN;
FILENAME SUB CLEAR;
X SUBMIT 'MYDIR.PDS.CNTL(DATMETR)'  ;
RUN;
endrsubmit;

/* End of LIB_ABC.sd2 transfer  */

signoff
"C:\SAS\connect\saslink\datcptso.scr";
%mend scon2Win;
%scon2Win(userId="MYID",pswrd="xxx");
```

The appendix contains an example of how Java can actually submit the generated SAS program to PC SAS.

A variety of SAS Procedures are used within the generated programs. For example, PROC SQL is used because it allows the tool to dynamically limit which rows and columns to include in the target dataset (per the user's specification). PROC DOWNLOAD is used to copy the dataset from a source to a target host, and PROC COMPARE is used when possible to assist with the verification that the transferred data is exactly the same as the source data. There are also cases where a SAS PROC was not available, such as for creating text files from the SAS datasets. Buchecker (1996) describes SAS macros that effectively automate the creation of flat files from SAS datasets.

SAS/Connect facilitates the communication with many different hardware platforms. This topic is dealt with in greater detail in our other paper titled, “Integrating SAS/Connect With Java”.

When it is necessary to execute an OS/390 based program, the tool will generate a JCL job. It then uses the FTP protocol to connect to the OS/390 mainframe, send a command to alert the host that the next command should be auto-submitted to the JES2 job stream.

CONCLUSION

Some of the benefits of the DATAccess Tool include:

- 15,000+ files transferred with no data corruption
- 35 teams have used the DATAccess Tool
- Ease of use – minimal or no training required
- Improves the clinical analysis process by allowing the users more flexibility to choose the software tools that provide the analytical capabilities they need
- Provides an easy to use graphical user interface for transferring SAS datasets from OS/390 to the NT or Unix servers, or the local machine.
- Shields the user from the complexities of OS/390, JCL, SAS/Connect and FTP
- Provides data conversion to a format that is usable at the remote host (SAS, SAS Transport, Microsoft Excel 97, CSV text, Tab delimited text, space delimited text, and dBase)
- Displays which SAS data libraries exist on OS/390 for a given research project code
- Displays which SAS datasets are available within a selected SAS library
- Source data can be moved immediately, any time of the day, any day of the week
- Common tool used across the organization (and around the globe)
- Source data can be OS/390 SAS, DB2 views, and Oracle views
- Leverages capabilities of OS/390 and UNIX
- Allows user to specify subsetting criteria by columns and rows
- Provides a verification report that addresses the integrity of the transferred data and that can be submitted to a regulatory authority.
- Provides transfer progress monitoring (color-coded)
- Provides users instant access to SAS datasets on the platform they prefer and in the format they choose
- Adheres to all corporate security standards
- Provides metrics reports on the number of transfers, what data was transferred and where, who transferred the data, what formats were used, etc.
- Uses batch mode for multiple transfers

REFERENCES

Curnutt, R., Pell, M., LaBore, J., “Integrating SAS/Connect® with Java™”, These Proceedings.

Buchecker, M. M. 1996. "%FLATFILE, and Make Your Life Easier," Proceedings of the Twenty-First Annual SAS Users Group International Conference, 21, 178-180.

SAS Institute Inc. (1996), *SAS Companion for the Microsoft Windows Environment*, Cary, NC: SAS Institute Inc.

SAS Institute, Inc. (1996*), SAS Companion for the MVS Environment, Version 6, Second Edition*, Cary, NC: SAS Institute Inc.

SAS Institute Inc. (1993), *SAS Companion for Unix Environments: Language, Version 6, First Edition*, Cary, NC: SAS Institute Inc.

SAS Institute Inc. (1994*), SAS/Connect Software: Usage and Reference, Version 6, Second Edition*, Cary, NC: SAS Institute Inc.

SAS Institute Inc. (1990), *SAS Guide to Macro Processing, Version 6, Second Edition*, Cary, NC: SAS Institute Inc.

SAS Institute Inc. (1989*), SAS Guide to the SQL Procedure: Usage and Reference, Version 6, First Edition*, Cary, NC: SAS Institute Inc.

TRADEMARK NOTICE

ABOUT THE AUTHORS

Randy Curnutt, Solutions Plus, Inc. (http://www.sol-plus.com) Randy Curnutt is the president of Solutions Plus, Inc., a software consulting company that specializes in applying leading edge technologies in order to provide comprehensive solutions to its clients. He focuses on client/server solutions, especially object oriented technology, and relational database management systems. He has experience with Java, Smalltalk, Visual Basic, C, C++, Oracle, MS SQLServer, and numerous other development languages. Randy may be reached via email at rcurnutt@sol-plus.com.

Michael Pell, Solutions Plus, Inc. (http://www.sol-plus.com) Michael Pell is a consultant at Solutions Plus, Inc., a software consulting company that specializes in applying leading edge technologies in order to provide comprehensive solutions to its clients. He focuses on the analysis, design, and implementation of object oriented technology client/server solutions. Michael has 2-3 years of Java development experience, and spent 6 years as an IBM consultant prior to joining Solutions Plus, Inc. Michael may be reached via email at mjpell@sol-plus.com.

John LaBore, Eli Lilly And Company (http://www.lilly.com) John LaBore is the SAS and JMP Coordinator for Eli Lilly and Company, a leading innovation-driven pharmaceutical corporation. He is responsible for supporting SAS and JMP use by Lilly staff worldwide. John has been a SAS software user for more than 20 years, and has authored numerous SAS technical papers for SUGI, PharmaSUG, SEUGI, and other SAS user group conferences.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors at:

Randy Curnutt
Solutions Plus, Inc.
10401 North Meridian, Suite 300
Indianapolis, IN 46217
(317) 848-3081
rcurnutt@sol-plus.com
http://www.sol-plus.com

Michael Pell
Solutions Plus, Inc.
10401 North Meridian, Suite 300
Indianapolis, IN 46217
(317) 848-3081
mjpell@sol-plus.com
http://www.sol-plus.com

John LaBore
Eli Lilly And Company
Lilly Corporate Center
Drop Code 6334
Indianapolis, IN 46285
(317) 277-6387
jml@lilly.com
http://www.lilly.com

APPENDIX

The generated PC-SAS program can be written to a temporary file, then PC-SAS can be run using the generated PC-SAS program (all done in the Java method as shown below).

```
/**
  * Writes the PC-SAS program to disk, then runs it.
  * @param sasProgramString String of the SAS program to be executed
  * @return int the return code returned by the executed SAS program
  */
public int runSasProgram(String sasProgramString) throws TransportFailedException
{
    int returnCode = 0;
    try
    {
           //Write the file to disk
           File sasProgramFile = new File("C:\\SAS\\GenProg.sas");
           BufferedReader buffReader = new BufferedReader(new StringReader(sasProgramString));
           PrintWriter sasProgOut = new PrintWriter(new FileOutputStream(sasProgramFile));
           printDIStreamOnPrintWriter(buffReader, sasProgOut);
           Runtime rt = Runtime.getRuntime();
           sasProgOut.close();

           //Now run the program using PC-SAS.  This is equivelant to starting PC-SAS from a
           //DOS command line
           StringBuffer programSb = new StringBuffer("C:\\SAS");
           programSb.append("\\SAS.exe -NOSPLASH -ICON -SYSIN \"");
           programSb.append(sasProgramFile.getPath());
           programSb.append("\"");
           Process myProc = rt.exec(programSb.toString());
           returnCode = myProc.waitFor(); //wait here until SAS is done running
           SasProgramFile.delete();  //be a good neighbor and clean up
    }
    catch (Exception e)
    {
           throw new TransportFailedException("Error communicating with PC-SAS software.");
    }
    return returnCode;
}
```

The Role of SAS/IntrNet* in a Web-Enabled Database System

John R. Copeland, David W. King, Paul C. Gangarosa; CDC, Atlanta, GA

ABSTRACT

The Centers for Disease Control and Prevention (CDC) has implemented an application, known as the New Vaccine Surveillance Network (NVSN) data system, to study the impact of new vaccines, specifically influenza and pneumococcal. The system allows investigators in Rochester, NY and Nashville, TN to enter patient data, generate reports and graphs, and download data, using only a web browser.

The NVSN data system allows users to store and retrieve data from a SQL Server 7 database. SAS/IntrNet®, along with HTML, JavaScript, Java applets, and Java Database Connectivity, are used for data extraction and analysis. The SAS Output Delivery System is used to send SAS output to users' web browsers. The system employs a Compaq 7000 for data storage and a Sun E5500 for SAS/Intrnet processing.

The NVSN data system demonstrates the ability of the SAS/IntrNet* product, combined with other Web and networking technologies, to provide remote users with a myriad of reports, graphs, and statistical analyses - any task that the SAS System can perform - as well as to extract their own data to perform independent analyses.

INTRODUCTION

In November 1999, the National Immunization Program (NIP) of the CDC implemented the NVSN. The initial purpose of NVSN was to measure incidence of acute respiratory infections (ARI) in children less than five years of age, with plans to study how the introduction of new vaccines (such as influenza and pneumococcal) affect ARI incidence. It was decided that first a system would be put in place to record ARI's from hospital admissions, and later a study involving outpatient ARI's would be added.

NIP enlisted Vanderbilt University in Nashville, Tennessee and The University of Rochester in Rochester, New York to recruit hospital patients to consent to having certain information submitted anonymously to this study. Vanderbilt University has access to three Nashville area hospitals and The University of Rochester has access to two Rochester area hospitals. Medical personnel and epidemiologists from NIP, Vanderbilt, and Rochester devised several forms, to be completed by Vanderbilt and Rochester personnel, to provide the information needed for this study. The forms were designed to collect demographic information, as well as information from hospital charts and laboratory tests.

Once the content of the forms was determined, the method by which the data would be delivered to NIP and stored had to be decided. Three methods were suggested. One suggestion was that Vanderbilt and Rochester would each enter their form data into Epi Info screens. (Epi Info is a thick-client application for data entry and some data analysis. It is widely used in the Epidemiology field.) The resulting data tables would then have to be sent to NIP by FTP, or by storing the data on compact disk and sending the disk by mail. Another suggestion was that NIP would develop software for entry and storage of the form data. This method would also require a means of transporting the data to Atlanta. The other suggestion was that NIP would develop a database and enable Vanderbilt and Rochester

personnel to, using only a Web browser, enter the data over a secure data network.

This presentation is part of the "Internet, Intranet, and Web" section of the conference, so obviously the secure data network is the method that was chosen. Several advantages were noted for using a secure data network over the other two methods:

- Ensures data consistency between the two sites
- Provides most rapid delivery of data to NIP
- Requires no on-site installation of software
- All upgrades can be made at NIP without having to ship subsequent versions of software

NVSN DATABASE

A relational database was developed using Microsoft® SQL Server. The NVSN database contains eight tables (one for each form), and the tables are linked by a unique ID variable. Staff at Vanderbilt and Rochester enter the data from the forms using Active Server Pages, also designed by NIP. SAS/IntrNet is used to allow NIP, Vanderbilt, and Rochester personnel to view reports and graphs; and to allow Vanderbilt and Rochester to retrieve their own data in comma-separated-values format, so that they can generate their own analyses.

SAS/ACCESS

SAS® Version 8.1, which is the version currently being used by NIP, allows tables of relational databases to be read as SAS datasets. Reading MS SQL Server tables requires first creating an ODBC connection from the machine running the SAS session to the database. Once this is accomplished, the SQL Server tables may be accessed as if they were SAS datasets by simply submitting a `libname` statement designating ODBC as the engine. The statement is of the form:

```
libname sqldata odbc
      noprompt="uid=userid;pwd=password
;dsn=odbcsourcename;";
```

where *`userid`* is an account with read access to the SQL Server database, *`password`* is the corresponding password for *`userid`*, and *`odbcsourcename`* is the name of the ODBC data source chosen when the ODBC connection was created. The process of accessing the SQL Server tables with SAS is slightly more complicated in the NVSN data system because the SAS/IntrNet product runs on a Sun® E5500 server with the Solaris 2.6 operating system, and ODBC is a Microsoft feature. This problem was solved by scheduling a SAS batch job to run every night on the MS NT Compaq® 7000 server which houses the SQL Server database, utilizing SAS/CONNECT* to copy all of the NVSN tables to the SAS/IntrNet server as SAS datasets.

Another attempted method of solving this access problem was to write a Java application that, when invoked by a SAS program, would read the appropriate SQL Server table and write its contents to the SAS/IntrNet server as a delimited text file. This text file could then be read by SAS with an `infile` statement. This method had the advantage of allowing SAS/IntrNet to capture the most recently recorded data, whereas the SAS/CONNECT method only allows SAS/IntrNet to report data that was entered as recently as the previous day. However, for simplicity's sake, the SAS/CONNECT method is the one that is being used.

SETTING UP SAS/INTRNET

To take advantage of the SAS/IntrNet product, we first had to install SAS version 8 on the SAS/IntrNet server. We followed

the recommendation of the installation program and installed SAS in the OPT directory on our Sun server. Then the IntrNet product itself was installed. A Pearl script file (/opt/sas8/utilities/bin/inetcfg.pl) comes with SAS/IntrNet and performs initial setup. When executed, this program creates subdirectories, sets up a port for default service, and builds the start.pl file. The start.pl file starts the SAS/IntrNet session and names the appropriate Appstart file. We customized the Appstart file to create the libraries to be later used in SAS programs written for SAS/IntrNet.

After the SAS/IntrNet server was configured, our Web server had to be set up to work properly with the SAS/IntrNet server. The same Sun E5500 is used for both the SAS/IntrNet server and the Web server, but this is not a requirement. We use Apache Web server which, after a separate setup program shipped with SAS/IntrNet is run, houses the Broker executable and the broker.cfg file in its cgi-bin directory. The broker.cfg file contains information about our Web server's directory structure and had to be customized for SAS/IntrNet to function properly.

SAS/INTRNET TASKS

The NVSN data system employs a Sun E5500, which serves as a Web server, a SAS/IntrNet server, and houses SAS datasets that contain the same data as the SQL Server tables. What do we want to do with it? What can we do? It was soon discovered that there is little that anyone wants that we cannot provide using SAS/IntrNet, especially when we combine SAS/IntrNet with Java and JavaScript. One of the earliest concerns about the NVSN data was, "is the data being entered logically?". For instance, is the patient's admission date before the discharge date? Or, is the difference between the patient's admission date and birth date within a month of what was entered as the patient's age? The SAS/IntrNet product provides a convenient way for project scientists to check questions such as these with the click of a mouse. A long list of such logic checks was devised for each NVSN form (some checks involve items from more than one form) and SAS programs were written to subset the NVSN datasets to include records where errors were found, and display the erroneous records in the requestor's Web browser.

Other items of interest on a daily basis are questions about enrollment and parental consent, and the age of patients participating in the study. These are issues that can be effectively addressed with charts and plots. The second NVSN SAS/Intrnet task is to graphically display simple statistics such as frequencies, percents, means, and quartiles.

The third task of SAS/IntrNet within the NVSN data system is to allow the remote sites (Vanderbilt and Rochester) to retrieve their own data, so that data analysts at each site can perform more complex and site-specific analyses of interest to them.

LOGIC CHECKS

As was stated before, a long list of logic checks was compiled for the seven main NVSN forms. To perform the checks, SAS data steps were used to subset the data, retaining records that do not pass the logic checks. Then `proc print`, along with the Output Delivery System (ODS) directing HTML output to the `_webout` location is used to print the results to the web browser. For instance, to check for records showing a discharge date before the admission date, code similar to the following is used:

```
ods listing close;
ods html body=_webout (dynamic nobot)
         rs=none;
data screening;
  set nvsnd.screening;
```

```
  keep caseid admitdate whoisit
timestamp;
proc sort;
  by caseid;
data chart;
  set nvsnd.chartrev;
  keep caseid dischargedate;
proc sort;
  by caseid;
data screening;
  merge screening (in=a) chartrev
(in=b);
  by caseid;
  if a and b;
data screening2;
  set screening;
  if input(admitdate,mmddyy10.) gt
input(dischargedate,mmddyy10.);
proc print;
  var caseid admitdate dischargedate
whoisit timestamp;
  title 'Admit Date After Discharge
Date';
run;
```

The `nvsnd` data library is assigned in the Appstart program. A `label` statement is also included in the last data step, but was omitted here for brevity. The variable, `whoisit` indicates the person who entered the record and `timestamp` is the date and time that the record was entered. The `caseid` variable is a unique identifier by which datasets may be merged. All other logic checks, such as making sure fields are the correct length and ensuring that age is consistent with birth and admission dates, are performed using this simple method of subsetting and printing with ODS.

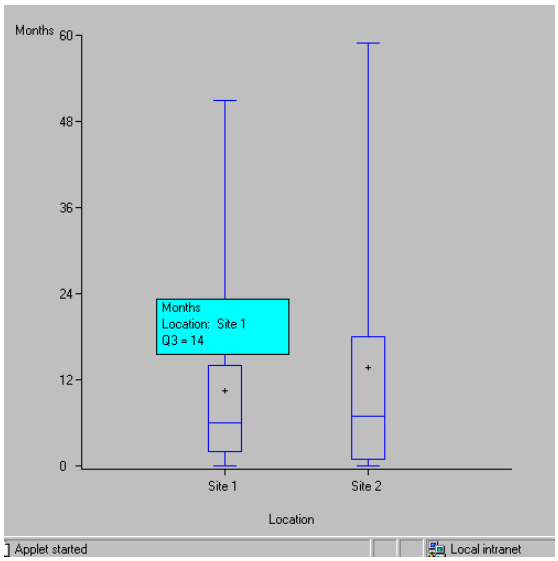
GRAPHS

SAS/IntrNet provides a choice of ways to display graph output in a Web browser. ODS can convert output from SAS/Graph* to graphics interchange format (GIF) and include the image in a web page. SAS/IntrNet also offers the use of the `%ds2graf(data=`*`dataset`*`, `*`other parameters`*`)` macro. This macro sends an applet that displays a histogram, pie chart, or scatter plot, to the Web browser. The variables of interest, as well as information about chart type and chart size, are passed as macro parameters. The applet is interactive: placing the mouse over a component of the graph displays information (variable levels and statistics) about that component; and a right mouse click displays popup menus to allow the user to change items such as graph type, 3D/2D, horizontal/vertical, and colors.

When these two methods, GIF and applet, were demonstrated to NIP, the applet was a big hit. It was decided that applets should be used whenever possible. (Loading times associated with dial-up connections may require us to rethink this decision.) The demonstrations to NIP staff elicited suggestions for types of graphs to show. One suggestion was a box plot showing age statistics for Vanderbilt and Rochester. SAS `proc boxplot` can fulfill this request, but since we are applet fans here at NIP, another method was needed to generate an applet displaying the requested box plot. With `%ds2graf` as inspiration, the `%ds2box` macro was written.

Developing this macro was a long process for a beginning Java programmer. The first step was to write an applet to display box plots with specific values for minimum, maximum, first and third quartiles, median, mean, and axis labels. Then, these values were replaced with the getParameter(String name) method; and the HTML that displayed the applet was changed to pass the necessary parameters. (This is the technique used by the `%ds2graf` macro.) Finally, `proc univariate`, `proc means`, and data steps were used to calculate these parameters and, using `put` statements, to write the HTML that displays the applet and passes the needed parameters. These

data steps were encased in the `%ds2box` macro, which only requires three parameters: a dataset name, a dependent variable, and an independent variable. So a SAS/IntrNet program that calls the `%ds2box` macro produces an applet, displaying box plots determined by the parameters passed to the macro. The applet is interactive, displaying precise values when the mouse pointer is placed over a portion of a box plot that represents a statistic. A right mouse-click opens a pop-up menu, allowing the user to change the applet colors.



DATA RETRIEVAL

SAS/IntrNet provides an easy way to download data to a client PC. The `%ds2csv` macro opens a dialog box that displays the user's directory tree. It allows the user to choose the name and location of the data file to be downloaded, and then writes the data to a comma-delimited text file. A file in this comma-separated-values format can be easily opened with Microsoft® Excel* and/or converted to a SAS dataset.

PUTTING IT ALL TOGETHER

Once the SAS/IntrNet product has been installed on the server, the broker.cfg file has been customized for the server, and the Appstart file has been edited to create program and data libraries, one needs only to know a little HTML to develop a functional Web-based data analysis and reporting system. For the NVSN system, the Appstart program designates a directory as the SAS program library, called NVSNP. All NVSN SAS programs are stored in this directory. The appstart program designates a directory to be the SAS data library called NVSND. All of the NVSN datasets are stored in this directory and the NVSND library is referenced in each program without submitting a `libname` statement. The SAS programs are run by executing the broker using common gateway interface (CGI) technology, and stipulating the desired port service and program. The NVSN data system uses two methods for accomplishing this. One is to simply write a hyperlink to call the application broker and designate the program. In each NVSN SAS program, except for one, the program that downloads the comma-separated-values file, this method is used. For example, the SAS program that performs the logic checks on the "screening" form data is run by writing the following HTML anchor tag:

```
<a href="http://directory/cgi-bin/broker?
_service=default&
_program=nvsnp.screening_checks.sas>
Screening
</a>
```

where *directory* is the root URL for our SAS/IntrNet system.

To download a file, the requestor needs to submit more information than for viewing a graph. The user must specify the site whose data are requested. The user must also specify which of the seven tables is wanted, and they also must submit their site's password. The NVSN system uses an applet to collect and submit this information. The site and table are selected from choice menus, and the password is typed into a text field. The applet will not

run the SAS program if any of these three fields are blank. The SAS program that contains the `%ds2csv` macro also performs the password validation. If the requestor submits the correct password, then the `%ds2csv` macro is executed. Otherwise, a SAS program is run that returns an HTML file stating that the password is incorrect. This task of selecting the site and table and entering the password could be more easily accomplished with an HTML form; however, the NVSN system is using an applet due to the programmer's short-lived experiment with client-side password validation. When executed, the applet points to the following URL:

http://*directory*/cgi-bin/broker?
_service=default&
_program=nvsnp.download.sas&
psswd=*password*&
location=*location*&
form=*dataset*

This is the same format as with the other NVSN SAS programs, except that values for the macro variables "psswd", "location", and "form" are passed to the download.sas program.

CONCLUSION

Web technologies provide a convenient and effective way of transmitting and storing data collected by NIP and its research partners. SAS/IntrNet, along with the ability of SAS to easily read data in a variety of commercial databases, provides an excellent mechanism for retrieving data in a Web browser. The data can be retrieved in the form of lists, tables, graphs, or any type of statistical analysis that SAS can perform. SAS/IntrNet also provides an easy way of delivering raw data to analysts who wish to perform their own analyses.

The development of the NVSN data system has been an excellent educational experience for everyone involved. It has also served as a demonstration tool to show NIP what we can do with relatively new technologies and it has given us a model on which we can base future projects.

RESOURCES

We took advantage of SAS training to learn some of the skills needed for this project. "SAS Web Tools: SAS/IntrNet Administration", and "SAS Web Tools: Static and Dynamic Solutions Using SAS/IntrNet Software" provided the knowledge needed to begin the SAS portion of the NVSN project. "SAS Web Tools: Advanced Dynamic Solutions Using SAS/IntrNet Software" covers the more advanced features such as the applet and comma-separated-values generating macros. The course notes from all three of these courses provide excellent references and the Web page, http://www.sas.com/rnd/web/intrnet/format/index.html is also a useful reference.

ACKNOWLEDGEMENTS

We would like to thank Henry Rolka, David Walker, and Gay Allen of NIP for making this project happen, Kimp Walton of NIP for sharing his SAS programming knowledge, and Fran Walker of NIP for helping to determine the reports to be generated.

CONTACT INFORMATION

John Copeland can be reached by phone at 404-639-8866, or by email at Jcopeland@cdc.gov.

Paper P316

The Beauty of OUT2HTM with Proc Report

David Steves, Suntrust Banks Inc., Atlanta, Georgia U.S.A.

ABSTRACT

Using Proc Reports to Create HTML pages via % OUT2HTM is very easy. This paper describes how Suntrust Bank utilizes this SAS tool to provide its internal customers from over 3600 Bank Branches with product information via the WEB.

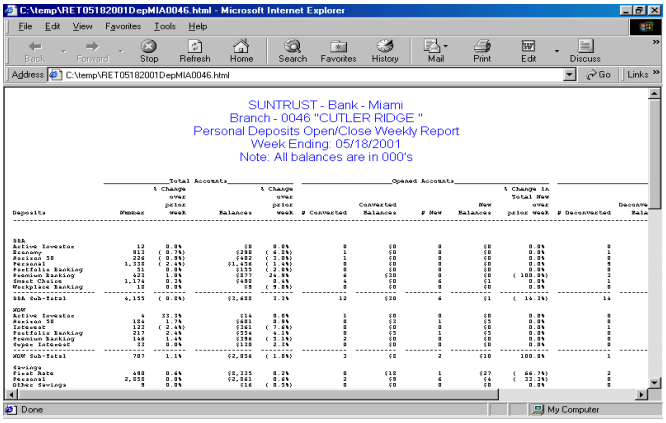
INTRODUCTION

The use of Proc Reports is widely accepted as an excellent report generator. Many SAS programmers have wanted to create these reports out on the WEB. The publication of these reports on the Intranet has enabled the business analysts to quickly review the results.

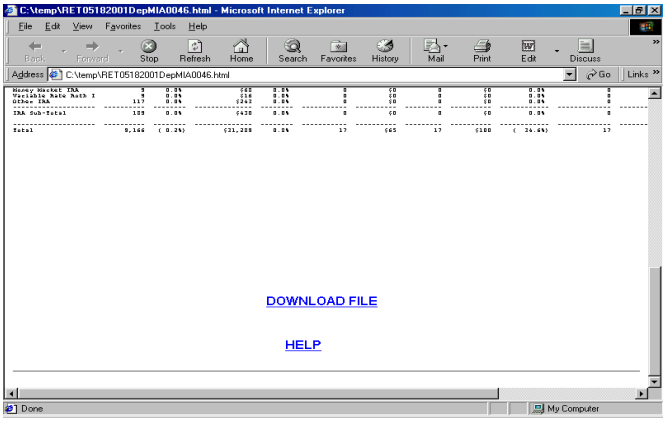
Suntrust Bank developed the weekly open/closed report to closely monitor its growing deposits. The report provides information on deposit account activity from a holding company view at its highest level to a branch view at its lowest level. This type of information will help management understand the weekly trends and performance in deposit product balances. Branch managers can review the state of deposits for their holding company, state, region, division, state, and branches.

Version 8 of SAS has an HTML formatting macro called %OUT2HTM to produce these Reports out to the WEB without having to know a large amount of HTML(Hypertext Markup Language). This paper focuses on how Suntrust Bank created over 3600 branch reports on the Intranet, as well as, little HTML tidbits that can make the report more attractive.

HTML FORMATING TOOLS



PICTURE 1



PICTURE 2

HTML Formatting Tools are a group of SAS macros that consist of (%OUT2HTM, %DS2HTM, and %TAB2HTM). The collection of macros enable the programmer to format and save SAS output for viewing in a WEB browser. The macros assign valid HTML tags creating HTML files. The HTML files that are created are static pages which can be pushed to the Intranet Server. Once on the Intranet server the SAS output can be viewed. The macro %OUT2HTM is a great formatting tool to get started in creating HTML pages. PICTURES 1 and 2 are HTML examples using %OUT2HTM.

SOURCE CODE FOR PICTURES 1 and 2:

/* use FORMCHAR='|----|+|---+=|-/\<>*' otherwise characters that you use may look unusual depending on your browser and the operating system used */

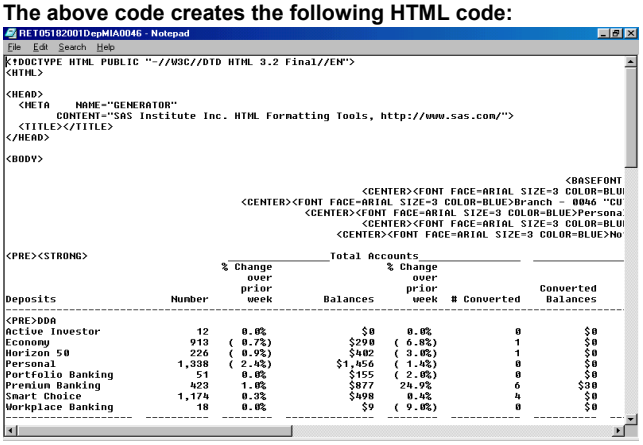
```
options ls=195 missing = '0' ps=100 nodate nonumber
 FORMCHAR='|----|+|---+=|-/\<>*';

    data bccom;
    set wext.bc wext.cdbc;
    format nord 8.;
    nord = prodord;
    run;

    data accom;
    set wext.ac wext.cdac;
    format nord 8.;
    nord = prodord;
    run;

    proc sort data=bccom;
    by acctbank grpord nord ;
    run;

    proc sort data=accom;
    by acctbr grpord nord ;
    run;

    /* used for date in the title of the proc
    report  */
    data _null_;
    x=&lastfri.;
```

```
call symput('dttitle',put(x,mmddyy10.));
run;

/* create proc format bank name commercial
data bktab1(keep = acctbank bankname);
set bccom(keep=acctbank bknam);
bkn = substr(bknam,1,3);

if bkn in ('STB') then do;
bankname = substr(bknam,5,40);
end;
else do;
bankname = bknam;
end;

run;

proc sort data=bktab1 nodupkey;
by acctbank bankname;run;

/* creates format for Bank names  */

data bankc;
  set bktab1(rename=(acctbank=start
bankname=label));
 fmtname='bankcom';
 type='C';
 keep fmtname label start type;
 run;

proc format cntlin=bankc;run;


***** branch section *****;
***** branch retail *****;

data branch(keep=acctbank acctbr brnam );
 set accom;
 run;

proc sort data=branch nodupkey;
by acctbank acctbr brnam;run;


/* very important step because it creates the
count needed for how many times the proc report
has to run - note the count and places names
for banks and branches */

data _null_;
 set branch end=no_more;
 call symput('ab'||left(_n_),acctbank);
 call symput('abn'||left(_n_),
                put(acctbank,    $bankcom.));
 call symput('br'||left(_n_),acctbr);
 call symput('brn'||left(_n_),brnam);
 if no_more then call symput('count',_n_);
run;

/* macro needed for the 3600 branches that
   Suntrust Banks have  */

%macro putloop;
   %local i;
   %do i=1 %to &count;

%macro
Branches(fdata,fname,brnam,hf,dlxls,namxls);

/* beginning of %OUT2HTM starts capturing
data */
%out2htm(capture=on);

/* note beginning proc report options
1. ls=195 the linesize max a report can
take for the web
2. nowindows is used so that the display
window for the proc report will not
produce an error during the running of
report */

proc report data=&fdata. split='*' ls=195
headline missing out=&dlxls. nowindows;
/* the basefont is used to make sure a
certain font is used to view the report */
title1 "<BASEFONT size=1.5>";


/* the following titles all use some HTML
in order to make sure the titles are
centered. Font size 3 is used for Letters
to appear bigger. The color blue is used
for the lettering */
title2 "<CENTER><FONT FACE=ARIAL SIZE=3
COLOR=BLUE>SUNTRUST - Bank -
&fname.</CENTER></FONT>";
title3 "<CENTER><FONT FACE=ARIAL SIZE=3
COLOR=BLUE>Branch - &&br&i
"&brnam."</CENTER></FONT>";
title4 "<CENTER><FONT FACE=ARIAL SIZE=3
COLOR=BLUE>Personal Deposits Open/Close
Weekly Report</CENTER></FONT>";
title5 "<CENTER><FONT FACE=ARIAL SIZE=3
COLOR=BLUE>Week Ending: &&dttitle.
</CENTER></FONT>";
title6 "<CENTER><FONT FACE=ARIAL SIZE=3
COLOR=BLUE>Note: All balances are in 000's
</CENTER></FONT>";

/* the following footnotes are hyperlinks
which are used in PICTURE2.
footnote1 is a hyperlink to a CSV file
which can be brought up in EXCEL.
footnote2 is a hyperlink to a USER GUIDE*/

footnote1 "<CENTER><FONT FACE=ARIAL SIZE=3
COLOR=BLUE><A HREF=./&namxls..CSV>DOWNLOAD
FILE</A></CENTER></FONT>";
footnote2 '<CENTER><FONT FACE=ARIAL SIZE=3
COLOR=BLUE><A
HREF="http://www.suntrust.com/WeeklyUserGu
ide.doc">HELP</A></CENTER></FONT>';

column grpord prodgrp prodroll onumcls
onumnew onumexs onumconv onumdcnv
       onewbal oexsbal oconvbal numcls
clsbal numnew newbal numexs
       exsbal numconv convbal numdconv
dconvbal
       ("_Total Accounts_" totaccts chg1b
totbals chg2b )
       ("_Opened Accounts_" numcon1
convbal1 numnew1 newbal1 chg3b)
       ("_Closed Accounts_" numdcon1
dconvbl1 numcls1 clsbal1 chg4b) ;
define grpord /order   noprint;
define prodgrp /order   noprint;
define prodroll / display format= $20.
width=20 'Deposits' left ;
```

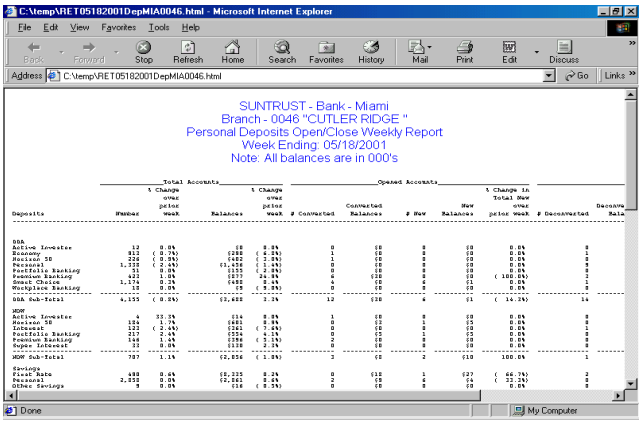
```
define onumcls / sum noprint;
define onumnew / sum noprint;
define onumexs / sum noprint;
define onumconv / sum noprint;
define onumdcnv / sum noprint;
define onewbal / sum noprint;
define oexsbal / sum noprint;
define oconvbal / sum noprint;
define numcls  / sum noprint;
define clsbal / sum noprint;
define numnew  / sum noprint;
define newbal / sum noprint;
define numexs  / sum noprint;
define exsbal / sum noprint;
define numconv  / sum noprint;
define convbal / sum noprint;
define numdconv  / sum noprint;
define dconvbal / sum noprint;
define totaccts / computed format=comma10.
width=10 "**Number" right;
define chg1b / computed format=percent7.1 width=8
"% Change*over*prior*week";
define totbals / computed format=dollar14.
width=14 "**Balances" right;
define chg2b / computed format=percent7.1 width=8
"% Change*over*prior*week";
define numcon1  / computed format=comma11.
width=11 '# Converted' right;
define convbal1 / computed format=dollar10.
width=10 "Converted*Balances" right;
define numnew1  / computed format=comma9. width=9
"# New";
define newbal1 / computed format=dollar10.
width=10 "New *Balances";
define chg3b / computed format=percent9.1
width=12 "% Change in*Total New*over*prior week";
define numdcon1  / computed format=comma13.
width=13 '# Deconverted' right;
define dconvbl1 / computed format=dollar11.
width=11 "Deconverted*Balances" right;
define numcls1  / computed format=comma9. width=9
"# Closed";
define clsbal1 / computed format=dollar10.
width=10 "Closed*Balances";
define chg4b / computed format=percent9.1
width=12 '% Change in*Total Closed*over*prior
week';

compute totaccts;
totaccts = numnew.sum + numexs.sum + numconv.sum;
endcomp;

compute totbals;
totbals = (newbal.sum + exsbal.sum +
convbal.sum)/1000;
endcomp;

compute numcon1;
numcon1 = numconv.sum;
endcomp;

compute convbal1;
convbal1 = (convbal.sum)/1000;
endcomp;

compute numnew1;
numnew1 = numnew.sum;
endcomp;

compute newbal1;
newbal1 = (newbal.sum)/1000;
endcomp;

compute numdcon1;
numdcon1 = numdconv.sum;
endcomp;

compute dconvbl1;
dconvbl1 = (dconvbal.sum)/1000;
endcomp;

compute numcls1;
numcls1 = numcls.sum;
endcomp;

compute clsbal1;
clsbal1 = (clsbal.sum)/1000;
endcomp;

compute chg1b;
if (onumnew.sum + onumexs.sum +
onumconv.sum) > 0 then do;
chg1b = (numnew.sum + numexs.sum +
numconv.sum - onumnew.sum - onumexs.sum -
onumconv.sum)/(onumnew.sum + onumexs.sum +
onumconv.sum);
end;
else do;
chg1b = 0;
end;
endcomp;

compute chg2b;
if (onewbal.sum + oexsbal.sum +
oconvbal.sum) > 0 then do;
chg2b = (newbal.sum + exsbal.sum +
convbal.sum - onewbal.sum - oexsbal.sum -
oconvbal.sum)/(onewbal.sum + oexsbal.sum +
oconvbal.sum);
end;
else do;
chg2b = 0;
end;
endcomp;

compute chg3b;
if (onumnew.sum ) > 0 then do;
chg3b = (numnew.sum -
onumnew.sum)/(onumnew.sum);
end;
else do;
chg3b = 0;
end;
endcomp;

compute chg4b;
if (onumcls.sum) > 0 then do;
chg4b = (numcls.sum -
onumcls.sum)/(onumcls.sum);
end;
else do;
chg4b = 0;
end;
endcomp;

break before prodgrp / ;
break after prodgrp / ol skip summarize ;

compute before prodgrp;
line @1  prodgrp  $20. ;
endcomp;

compute after prodgrp;
prodroll = trim(prodgrp)||' Sub-Total';
endcomp;
```

```
rbreak after  / ol skip summarize ;

compute after;
prodroll = 'Total';
endcomp;
run;

/* the final 4 lines of %out2htm.
capture=off - stops capturing output.
encode=n lets the browser render the title with
attributes specified.
htmlfile - lets you name the HTML file.
ttag= no formatting - tells the output formatter
not to assign any HTML tags.
** you could also add bgtype=image which
indicates an image as a background type, then
specify the name of image in the BG argument.
*/

%out2htm(capture=off,
         encode=n,
          htmlfile=/gs/&hf. ,
         ttag=NO FORMATTING);

/* this section is used to rename the variables
that will be used for a CSV file which can be
displayed in EXCEL */

data b&dlxls.(drop= totaccts chg1b totbals chg2b
numcon1 convbal1 numnew1 newbal1 chg3b
                    numdcon1 dconvbl1 numcls1
clsbal1 chg4b);
set &dlxls.(drop =grpord onumcls onumnew onumexs
onumconv onumdcnv
       onewbal oexsbal oconvbal numcls clsbal
numnew newbal numexs
       exsbal numconv convbal numdconv dconvbal
_break_);
if prodroll in ('  ') then delete;
total_num_accts       = totaccts;
tot_acct_perc_chg     = chg1b;
total_balances        = totbals;
tot_bal_perc_chg      = chg2b;
num_converted         = numcon1;
converted_balances    = convbal1;
num_new               = numnew1;
new_balances          = newbal1;
new_perc_chg          = chg3b;
num_deconverted       = numdcon1;
deconverted_balances  = dconvbl1;
num_closed            = numcls1;
closed_balances       = clsbal1;
closed_perc_chg       = chg4b;
run;

/* the creation of the CSV file used
   in the HYPERLINK on the HTML page. */

PROC EXPORT DATA= b&dlxls.
            OUTFILE="/gs/&namxls..csv"
            DBMS=CSV REPLACE;

RUN;

/* macro branches ending - which is used for
selection of acctbank and branch, name of
acctbank, name of branch, HTML file name, and
CSV file name */

%mend branches;
%branches(accom(where=(acctbank  in("&&ab&i")and
acctbr in ("&&br&i"
    ))),&&abn&i,"&&brn&i",RET&&fdt.Dep&&ab&i.&
    &br&i...html,t1,RET&&fdt.Dep&&ab&i.&&br&i.
    );
    %end;

    %mend putloop;
    %putloop;
```

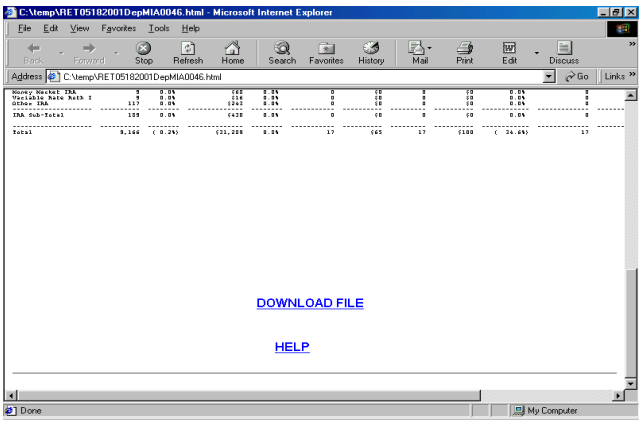
The above code creates the following HTML code:



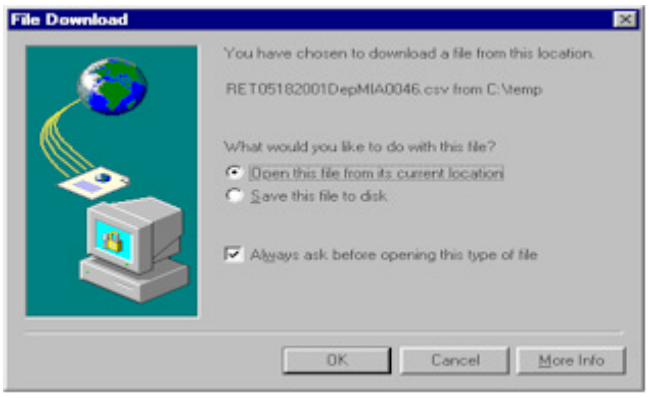
which appears on the web like this:



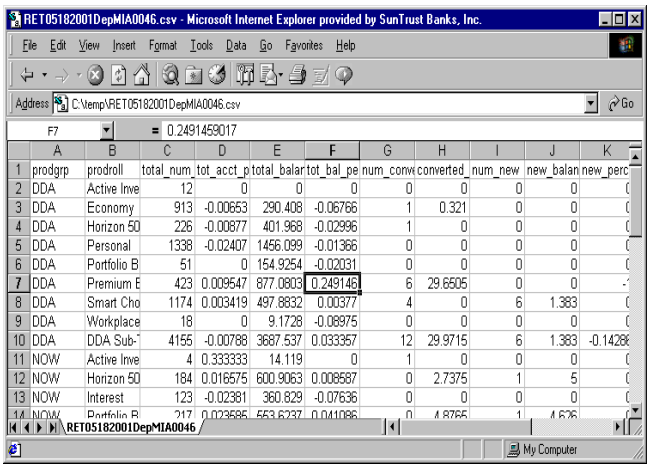
The hyperlinks are at the bottom of each report:



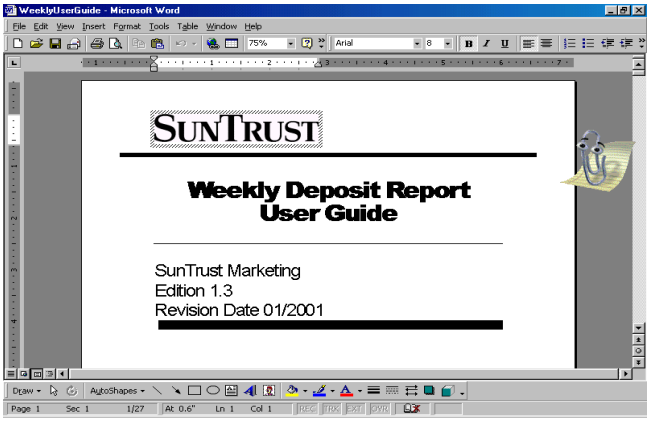
Once the download file hyperlink is clicked it appears as:



Select the 'Open this file from its current location' option and it appears as:



Once the HELP hyperlink is clicked it appears as a word document:



CONCLUSION

SAS version 8 provides an excellent tool for producing Proc Reports on the WEB. The %OUT2HTM macro is a very helpful HTML formatting tool that is beneficial to Suntrust Bank. Quite easy to use, 5 lines of codes can convert a simple Report to an HTML page with similar characteristics as the non-HTML type report. Remember, to first write the PROC REPORT then add the SAS macro %OUT2HTM . Next, add some HTML to your own specifications for your desired results.

ACKNOWLEDGEMENTS

SAS is a registered trademark of SAS Institute Inc. in the USA and other countries.

REFERENCES

SAS online support.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

David Steves
Suntrust Banks Inc.
P.0 Box 4418
Atlanta, Georgia 30302
Work Phone: (404) 827-6581

Email: david.steves@suntrust.com

Paper P317

Web Based Report Ordering combined with Base/SAS® Mainframe Batch Processing

Andre Brainard, Systems Engineering Services Corp., Reston, VA

ABSTRACT

This paper presents a web-based on-demand report ordering system that uses UNIX System Services (USS) and WebSphere to update and report data stored on an IBM OS/390 mainframe. The business user issues a request from a desktop PC and gets the results delivered via e-mail and imbedded URL hyperlinks.

INTRODUCTION

The Management Analytics Reporting System (MARS) was developed in response to a need for a centralized reporting system that would allow business users to request ad-hoc reports in an automated fashion.

Prior to the implementation of this system, ad-hoc reporting relied upon informal request procedures between business users and an often limited number of technical resource personnel. This approach made the process more error prone and time consuming. In addition, users from different business areas often duplicated requests, which resulted in higher costs and at times,inconsistent results.

NEW SYSTEM REQUIREMENTS

A new system would require the following capabilities:

- The ability for users corporate wide to access the report ordering facility
- The ability for the user to specify varied and multiple selection criteria such as date ranges, customer identification, and product identification
- The ability to query large volume mainframe data stored on multiple mediums including DASD and magnetic tape
- The ability to query data stored in various forms including relational database, VSAM files, GDG flat files, and IMS database
- The ability to query data residing on multiple platforms including mainframe and UNIX.
- The ability to query data using different access protocol such as Sybase, UDB/DB2, IMS, VSAM, SAS, flat files, GDG datasets, and PDS datasets
- The ability to leverage reuse of legacy mainframe ad-hoc reporting code including COBOL, SAS, and SAS format libraries
- The ability to provide automated delivery of the ordered report results to the user
- The ability to provide automated notification of job failures

THE FRONT-END – A WEB BASED USER INTERFACE

The **Web browser**, **e-mail** and **spreadsheet** were the three tools chosen to meet the front-end user interface requirements for several reasons:

- On an enterprise level these familiar and friendly tools were already in place and being used on a daily basis by the business areas.
- Little or no training would be necessary for the users and no installation or maintenance would be required for the users' PCs.
- The use of HTML and JavaScript would allow the technician to easily develop input forms for the query selection criteria.
- E-mail would provide both the automated notification and delivery of the query results via imbedded URL hyperlinks.
- Business users agreed that it would be desirable to receive the query results in spreadsheet form, which would facilitate viewing, printing, and further analysis.

THE BACK-END – BASE/SAS AND OS/390

The back-end tools and environment chosen were Base/SAS running on an OS/390 Enterprise Server (formerly known as the mainframe) with MVS, OpenMVS, UNIX System Services (USS), and WebSphere.

SAS batch processing running in the OS/390 MVS environment would provide the flexibility and power to process large volume data on multiple platforms residing in diverse storage form using a variety of access protocols. SAS also would provide the ability to easily sort, merge, and summarize large volume complex data.

The robust OS/390 MVS environment would provide the necessary system memory, disk workspace and processing power to accomplish complex queries in a reasonable time frame. OS/390 USS and WebSphere would provide the critically important bridge between the mainframe environment and the corporate wide intranet.

SAMPLES OF CRITICAL CODE

The body of this paper provides samples of the front-end JavaScript code, the back-end MVS JCL with associated PROCS, and the back-end Base/SAS code necessary for the development of a similar web-based on-demand report ordering system.

The complete REXX CGI script will be provided upon request.

JAVASCRIPT CALL OF WEBSERVER CGI SCRIPT

The front-end JavaScript does the following:

- Establishes input variables from Web page forms
- Performs minimal validations
- Assembles a delimited parameter list to be passed to the mainframe CGI script
- Substitutes imbedded space characters with a "+" in the parameter list
- If all data is valid, calls mainframe CGI script residing on the OS/390 OpenMVS webserver

```
function submitJob()
 {  var validData = true;

  var e = document.forms[0].seller;
   var seller  = e.value;
   if (seller == "") {var seller = "*";}

  if (validData && seller != "*" &&
      (isNaN(seller) || seller.length != 6 || seller < "000001"))
     { alert("Enter a valid Seller Number"); var validData = false;}

  var parmList = "?" + "MARS.CNTL(MARS001)" +
                       "?" + "MARS0001" + "?" + startdate +
                       "?" + enddate + "?" + seller + "?" + reportby;

  out = " "; // replace this
  add = "+"; // with this
  temp = "" + parmList; // temporary holder
  while (temp.indexOf(out)>-1)
   { pos= temp.indexOf(out);
     temp = "" + (temp.substring(0, pos) + add +
     temp.substring((pos + out.length), temp.length)); }

  var parmString = temp;
  var webServer = "http://omvs2.company.com/cgi-bin/";
  var cgiScript = "webparms.cgi";

   if (validData)  { alert("Your query results will be sent via e-
mail.");
            window.open(webServer + cgiScript + parmString) }
 }
```

THE CGI SCRIPT EDIT AND JOB SUBMISSION

The called CGI script written in the REstructured eXtended eXecutor (REXX) language performs the following:

- Retrieves the targeted mainframe job from the specified MVS library
- Edits the JCL scanning for the anchor point statement "//JS00 EXEC WEBPARMS" which is replaced with the following JCL and the parameters passed by the JavaScript
- Submits the edited Jobstream to the OS/390 MVS Job Entry System (JES) internal reader
- Returns notification of either success or failure to requesting user

```
//*** THE FOLLOWING INSERTED BY CGI SCRIPT ***
//WEBPARMS EXEC PGM=IEBGENER
//SYSUT1   DD *
 LIBRARY(JOBNAME)
 USERID
 CCYYMMDD
 HH:MM:SS
 Parm(s) each on separate line, start cc 2, max 70 char
/*
//SYSUT2   DD DSN=&&WEBPARMS
//        DISP=(NEW,PASS,DELETE),
//        SPACE=(TRK,1),UNIT=SYSDA,
//        DCB=(LRECL=80,BLKSIZE=0,RECFM=FB)
//SYSPRINT DD SYSOUT=*
//SYSIN    DD DUMMY
```

THE TARGET JCL STRUCTURE

The following mainframe batch Job Control Language (JCL) example would be stored as a member of an OS/390 MVS mainframe Partitioned Data Set (PDS) library. The JBS bind statement is essential to insure that the JES binds these jobs to the appropriate MVS environment, in this case production.

```
//MARS0001 JOB (000,RK),'MARS',CLASS=M,MSGCLASS=P
//*+JBS BIND SERVER.USSM2
//*
//*** ESTABLISH INPUT WEB PARMS ***
//JS00     EXEC WEBPARMS       <<== anchor for cgi script
//*
//*** SAS QUERY PROCESSING ***
//JS10     EXEC SAS
//SYSIN    DD DSN=MARS.SOURCE(MARS001S), DISP=SHR
//JOBINFO  DD *
//WEBPARMS DD DSN=&&WEBPARMS,DISP=SHR
//OMVSXFER  DD DSN=&&OMVSXFER, DISP=(NEW,PASS),
//                  UNIT=…,SPACE=…,DCB=(……)
//EMAILTO  DD DSN=&&EMAILTO, DISP=(NEW,PASS),
//                UNIT=…,SPACE=…,DCB=(……)
//*** INSERT DD STATEMENT(S) FOR ANY INPUT FILES
//*
//*** XFER RESULTS TO WEBSERVER USING OPUT ***
//JS20     EXEC WEBDATA,COND=(4,LT)
//SYSTSIN  DD DSN=&&OMVSXFER,DISP=SHR
//*
//*** SEND EMAIL QUERY RESULTS TO USER ***
//JS30    EXEC SENDMAIL,COND=(4,LT)
//SYSTSIN  DD DSN=&&EMAILTO,DISP=SHR
//*
//*** Optional Failure Notification goes here (refer later)
```

THE STORED JCL PROCEDURES

The following three mainframe JCL procedures will need to reside as members of the production OS/390 MVS PROC library.

```
//*** PROC:  WEBPARMS ***
//WEBPARMS PROC
//WEBPARMS EXEC PGM=IEBGENER
//SYSUT1   DD DUMMY
//SYSUT2   DD DSN=&&WEBPARMS, DISP=(NEW,PASS),
//  UNIT=SYSDA, SPACE=(80,(1,1)),
DCB=(BLKSIZE=0,DSORG=PS)
//SYSPRINT DD SYSOUT=*
//SYSIN    DD DUMMY
```

```
//*** PROC:  WEBDATA ***
//WEBDATA PROC
//WEBDATA EXEC PGM=IKJEFT01
//SYSPROC  DD  DISP=SHR,DSN=SYS2.CLIST
//SYSTSPRT DD  SYSOUT=*
//SYSPRINT DD  SYSOUT=*
//SYSTSIN  DD  DUMMY        <== USER OVERRIDES
```

```
//*** PROC:  SENDMAIL ***
//SENDMAIL PROC
//SENDMAIL EXEC PGM=IKJEFT01
//SYSPROC  DD  DISP=SHR,DSN=SYS2.CLIST
//SYSTSPRT DD  SYSOUT=*
//SYSPRINT DD  SYSOUT=*
//SYSTSIN  DD  DUMMY        <== USER OVERRIDES
//*********************************************************************
//*  PURPOSE:  TO SEND E-MAIL FROM THE MAINFRAME
//*  ----------------------------------------------------------------
//*  SYSTSIN DD MUST BE OVERRIDDEN BY USER SYSTSIN
//*   WITH EITHER A REAL OR TEMPORARY DSN.
//*   TEMPLATE OF SYSTSIN INPUT AS FOLLOWS:
//*  ----------------------------------------------------------------
//*  SENDMAIL TO(SOME_ONE@COMPANY.COM) +
//*                  CC(SOMEBODY_ELSE@COMPANY.COM) +
//*                  SUBJECT(SUBJECT MATTER) +
//*                  DATASET('DSN FOR E-MAIL BODY') +
//*                  BATCH
//*********************************************************************
```

THE SAS QUERY

PART 1 - PICKUP SYSTEM ASSIGNED JOB-ID

The first section of code performs the initialization necessary to support the remaining processing. In order to provide the pertinent job identification information in the e-mail notification, the MVS JOB-ID is picked up from the Job File Control Block (JFCB) by using a dummy DD JCL statement.

```
/* INITIALIZE:  PICKUP JOB-ID EXECUTION INFO */
DATA _NULL_;
INFILE JOBINFO JFCB=JFCB;  JOB_ID='      ';
IF INDEX(JFCB,'.JOB')
   THEN
JOB_ID='J'||SUBSTR(JFCB,(INDEX(JFCB,'.JOB'))+4,5);
CALL SYMPUT('JOB_ID',JOB_ID);
RUN;
```

PART 2 - WEB PARMS INTO GLOBALS

As part of initialization, this code inputs each Web parameter and converts it into a global variable to enable usage by the processing code sections that follows. Adding PUT PARMVAR= statements for each input parm would provide a nice audit trail in the SASLOG file.

```
/* INITIALIZE:  INPUT WEB PARMS AS GLOBAL VARIABLES
*/
DATA _NULL_;
INFILE WEBPARMS MISSOVER END=ENDPARMS;
IF _N_ = 1 THEN DO;        /* LIBRARY(MEMBER) */
            INPUT @2 JOB_RUN $50.;
            L_PAREN=INDEX(JOB_RUN,"(");
            R_PAREN=INDEX(JOB_RUN,")");
            JOB_LNTH=(R_PAREN - L_PAREN) -1;

JOB_NAME=SUBSTR(JOB_RUN,L_PAREN+1,JOB_LNTH);
            CALL SYMPUT('JOB_RUN',JOB_RUN);
            CALL SYMPUT('JOB_NAME',JOB_NAME);
            END;
IF _N_ = 2 THEN DO;        /* MAINFRAME USER-ID */
            INPUT @2 USER_ID  $7.;
            CALL SYMPUT('USER_ID',USER_ID);
            END;
IF _N_ = 3 THEN DO;        /* DATE STAMP CCYYMMDD */
            INPUT @2 IN_DATE  $8.;
            CALL SYMPUT('IN_DATE',IN_DATE);
            END;
IF _N_ = 4 THEN DO;        /* TIME HH:MM:SS */
            INPUT @2 IN_TIME  $8.;
            CALL SYMPUT('IN_TIME',IN_TIME);
            END;
IF _N_ = 5 THEN DO;        /* MARS REPORT-ID   */
            INPUT @2 MARS_ID $8.;
            CALL SYMPUT('MARS_ID',MARS_ID);
            END;

 /* -- USER INPUT PARAMETERS FOLLOW -- */
IF _N_ = 6 THEN DO;        /* BEGIN DATE */
            INPUT @2 IN_BEGDT $8.;
            CALL SYMPUT('IN_BEGDT',IN_BEGDT);
            END;
IF _N_ = 7 THEN DO;        /* END DATE */
            INPUT @2 IN_ENDDT $8.;
            CALL SYMPUT('IN_ENDDT',IN_ENDDT);
            END;
IF _N_ = 8 THEN DO;        /* Seller Number */
            INPUT
            @002 FOR_SLR $6.;
            CALL SYMPUT('FOR_SLR',FOR_SLR);
            END;
IF _N_ = 9 THEN DO;        /* Report-By Offering */
            INPUT
            @002 RPT_BY $3.;
             CALL SYMPUT('RPT_BY',RPT_BY);
            END;
IF ENDPARMS THEN DO;    /* Verify Parm Count */
             IF _N_ < 9 THEN ABORT ABEND 255;
             STOP;
             END;
RUN;
```

PART 3 – MAIN DATA QUERY PROCESSING

This section should include the queries, data extracts, sorting, merging and summarization to produce the specific request results. The power of SAS batch mainframe processing allows tremendous flexibility to access diverse and large sources of data residing on DASD, magnetic tape, and remotely via SAS/Connect.

```
/* PROCESS:  QUERY and DATA EXTRACT */

/* PROCESS:  SORT, MERGE, SUMMARIZE, ETC. */
DATA RESULTS;
```

PART 4 - ALLOCATE OUTPUT RESULTS FILE

This code allocates the output file dynamically in the SAS code in order to generate a unique “DELETE.USERID.*” DSN thus avoiding duplicate DSN conflicts.

Important reminders:

- The OPUT transfer command must reference a system catalogued dataset.
- Use of delete.userid. catalogued datasets are permitted for any user-id with only minimal security access.
- Use of delete.userid. datasets facilitates automatic file cleanup by MVS.

- Be sure to allow both an adequate logical record length (LRECL=) and an adequate DASD file space (SPACE=) as output needs dictate.

```
/* INITIALIZE: ALLOCATE UNIQUE DSN RESULTS  FILE */
DATA _NULL_;
MARS_DAT = "'DELETE.&SYSUID..D&RUN_JDAY..T" ||
           COMPRESS("&RUN_TIME",':') ||
".&MARS_ID..DAT'";
MARS_DAT = UPCASE(COMPRESS(MARS_DAT));
CALL SYMPUT('MARS_DAT',MARS_DAT);
RUN;

DATA _NULL_;
FILENAME &MARS_ID &MARS_DAT NEW
  DISP=(NEW,CATLG,DELETE) SPACE=(TRK,(100,10),RLSE)
  UNIT=SYSDA LRECL=800 BLKSIZE=0 RECFM=FB
DSORG=PS;
RUN;
```

PART 5 – GENERATE OUTPUT .CSV RESULTS

This code generates the output results data as a Comma Seperated Values (CSV) file. The .CSV file type extension is critical for enabling the Web browser to recognize this file type belongs to the spreadsheet application. When the user clicks on the results_file.csv hyperlink, inserted into the body of the e-mail, the browser will pass the .CSV file from the Webserver directly into the spreadsheet application.

```
/* OUTPUT RESULTS AS COMMA DELIMITED FILE */
DATA _NULL_;
SET RESULTS;
FILE &MARS_ID NOPRINT;
PUT ,,,,,, comma delimited output ,,,,,
RUN;
```

PART 6 - GENERATE FILE TRANSFER COMMANDS

The following SAS code will generate the OPUT command necessary for transferring the MVS output results over to the OpenMVS USS WebSphere area. The new file resides as an HFS file under WebSphere available for user access via the imbedded URL in the e-mail body.

```
/* OUTPUT COMMAND FILE FOR XFER TO WEBSPHERE */
DATA _NULL_;
MARS_CSV = "&SYSUID._d&RUN_JDAY._t"  ||
        COMPRESS("&RUN_TIME",':') || "_&MARS_ID..csv";
MARS_CSV = LOWCASE(COMPRESS(MARS_CSV));
CALL SYMPUT('MARS_CSV',MARS_CSV);
PUT MARS_CSV=;
RUN;

DATA _NULL_;
  IF &TRANOUT = 0 THEN STOP;
  OUT_UNIX = TRIM("&MARS_CSV");
  FILE OMVSXFER NOPRINT;
  PUT " oput &MARS_DAT +";
  PUT " '/u/mars/prod/reports/ondemand/" OUT_UNIX +(-1) "'";
```

PART 7 - GENERATE E-MAIL NOTIFICATION

This final section of the SAS Query program code generates the e-mail that serves as the notification and delivery vehicle of the query results. Assuming a successful query, the body of the e-mail will contain an imbedded hyperlink which the business user will click-on to conveniently have the results delivered directly in spreadsheet form.

Important points:

- The SENDMAIL command must reference a system catalogued dataset for the e-mail body
- The e-mail body DSN must exist after the job completes and be available to the queued SENDMAIL (up to 2 minutes)

```
/* GENERATE ROUTING AND BODY OF E-MAIL MESSAGE */
DATA _NULL_;
MARS_MSG = "'DELETE.&SYSUID..D&RUN_JDAY..T" ||
         COMPRESS("&RUN_TIME",':')       ||
         ".&MARS_ID..MSG'";
MARS_MSG = UPCASE(COMPRESS(MARS_MSG));
CALL SYMPUT('MARS_MSG',MARS_MSG);
RUN;

DATA _NULL_;
FILENAME EBODY &MARS_MSG NEW
     DISP=(NEW,CATLG,DELETE) SPACE=(TRK,(1,1),RLSE)
     UNIT=SYSDA LRECL=100 BLKSIZE=0 RECFM=FB
DSORG=PS;

DATA _NULL_;
  FILE EMAILTO NOPRINT;
  USER_ID = PUT("&USER_ID",$MARSUSR.);
  IF USER_ID = 'MISSING'
    THEN EMAIL_TO = "mars_project@company.com";
    ELSE EMAIL_TO = USER_ID||"@company.com";
  MARS_MSG = TRIM("&MARS_MSG");
  PUT "  SENDMAIL TO=" EMAIL_TO +(-1) ") +";
  PUT "  CC(MARS_PROJECT@company.COM) +";
  IF USER_ID = 'MISSING'
   THEN PUT "  SUBJECT(*ERROR* id: &USER_ID "
          "missing from MARS e-mail user table) +";
   ELSE IF &TRANOUT > 0
   THEN PUT "  SUBJECT(*SUCCESS* YOUR MARS0001
RESULTS ARE READY) +";
   ELSE PUT "  SUBJECT(*NOTICE* NO RESULTS RETURNED
FOR MARS0001) +";
  PUT "  DATASET(" MARS_MSG +(-1) ") +";
  PUT "  BATCH";

DATA _NULL_;
  FILE EBODY NOPRINT;
    SET CDW_FHA;
  IF _N_ = 1 THEN DO;
  PUT "Management * Analytics * Reporting * System" /;
  PUT "OS/390 ENTERPRISE SERVER:  "
     "MVS JOBNAME=&SYSJOBID JOBID=&JOB_ID
USERID=&SYSUID";
  REQ_DATE = INPUT("&IN_DATE",YYMMDD8.);
  PUT "SUBMITTED ON: " REQ_DATE WEEKDATE29. "  at:
&IN_TIME"
      "     by USER-ID:  &USER_ID " /;
  PUT "REPORT:  *MARS0001* REPORT TITLE" /;
     IF INPUT("&FOR_SLR",$1.) ne '*'
      then do;
           SELLER = TRIM(PUT("&FOR_SLR",$SLRNAM.));
           PUT 'FOR SELLER: ' SELLER;
           end;

 IF INPUT("&RPT_BY",$1.) NE '*'
```

```
      THEN BYDESC =
TRIM(PUT(INPUT("&RPT_BY",$3.),$RPTBY.));
      ELSE BYDESC = 'All Flow Offerings';
    PUT 'REPORT-BY: ' BYDESC;
    PUT "PERIOD: &BEG_DATE TO &END_DATE ";
    CUR_DATE = TODAY();
    PUT "STATUS AS OF: " CUR_DATE MMDDYY10. /;

  IF &TRANOUT > 0 THEN DO;
   PUT "CLICK ON LINK TO VIEW RESULTS:";
   PUT "http://omvs2.company.com/marsreportsp/&MARS_CSV"
/;
                END;
              ELSE DO;
  PUT "*NOTICE* NO RESULTS RETURNED FROM THIS
QUERY!" /;
                END;
RUN;
```

JOB FAILURE NOTIFICATION – JCL CODE

Add the following jobs steps to the end of the targeted JCL for the purpose of generating and sending e-mail notification of a job failure. Inclusion of job specific information, i.e. Job name and system assigned Job-ID, will help with providing efficient production support.

```
//*
//*** HANDLE JOB FAILURE NOTIFICATION ***
//JC01  IF ABEND=TRUE OR RC > 4 THEN
//*
//*** CREATE E-MAIL NOTIFICATION OF JOB FAILURE ***
//JS40     EXEC SAS
//SYSIN    DD DSN=MARS.SOURCE(MARS911S),DISP=SHR
//JOBINFO  DD *
//WEBPARMS DD DSN=&&WEBPARMS,DISP= DISP=SHR
//NOTIFY   DD DSN=&&NOTIFY,DISP=(NEW,PASS),
//            UNIT=…,SPACE=…,DCB=(……)
//*
//*** SEND E-MAIL NOTIFICATION OF JOB FAILURE ***
//JS50    EXEC SENDMAIL
//SYSTSIN  DD DSN=&&NOTIFY,DISP=SHR
//*
//JC01  ENDIF
```

JOB FAILURE NOTIFICATION – SAS CODE

The SAS program MARS911S code necessary for generating the e-mail notification of job failure is easily replicated from the first and last sections of the MARS001S SAS Query program code.

SAS NOTIFY

Part 1 – Pickup System Assigned Job-ID
Copy the “Part 1- Pickup Job-ID“ SAS code from the MARS001S SAS Query program above.

Part 2 – Web Parms into Globals
Copy the “Part 2 - Web Parms into Globals“ SAS code from the MARS001S SAS Query program above.

Part 3 – Generate E-mail Notification
Copy the “Part 7 - Generate Delivery E-mail” SAS code from the MARS001S SAS Query program above and then modify the subject and body to announce job failure along with action to be taken and relevant contact information.

CONCLUSION

The MARS project provides an enterprise level solution that has not only brought ad-hoc reporting under control, but also has established a new technical infrastructure paradigm. Centralizing ad-hoc reporting on the corporate intranet opens it to the widest possible audience, while at the same time dramatically reducing future development costs. Accessibility of existing ad-hoc reports encourages their reuse and helps to avoid duplication. With the business user now able to independently order ad-hoc reports using flexible selection criteria, the need for doing “what if” scenarios has been made easier and faster. And finally, the new technical architecture and toolset improves the collaborative JAD/RAD approach to meet future ad-hoc reporting needs.

ACKNOWLEDGMENTS

Thanks to William A. Mitchell, Lead Tech Analyst, Freddie Mac Corporation, MacLean, VA for developing the REXX CGI script and for providing invaluable technical support.

Thanks to Michael G. Sadof, a SAS Quality Partner, MGS Associates, Bethesda, MD for encouraging me to present this paper.

CONTACT INFORMATION

For those interested in attaining full version copies of the code, including the REXX CGI script , please make request to the e-mail address that follows.

Andre W. Brainard,
e-mail: andretech@onebox.com
Systems Engineering Services Corp.
On the Web at: http://www.sesc.com

A Generic Solution to Running the SAS® System on the Web Without SAS/Intrnet®

David L. Ward, InterNext, Inc., Somerset, NJ

ABSTRACT

Many organizations are not able to afford SAS/IntrNet but desperately need to provide dynamic web-based content to users, usually in the form of on-line reports. Sample CGI scripts that invoke the SAS system have been offered at various user group conferences, but a complete alternative has not been available. This paper will demonstrate the installation and usage of this alternative, as well as show examples of reports generated real-time in a web-browser using only Base/SAS. You will learn to create HTML forms that pass parameters to a SAS program that displays a table or graph according to what a user has chosen. You will also learn to use ODS to generate presentation-quality output.

INTRODUCTION

This paper will serve to show a number of things. First, the power and flexibility of the SAS System, specifically SAS/AF, in building complex Internet-based applications that communicate in a client-server architecture. Second, it will exhibit InterNext's product **Onyx**, including installation, administration, usage, and advanced features. Third, readers will see examples of how to build web-based SAS programs using either Onyx or SAS/IntrNet. ODS, SAS/Graph, and other techniques will be discussed.

OVERVIEW OF CONCEPTS

Before we explore the details of using SAS with the web you should familiarize yourself with some concepts pertinent to the topic at hand.

1. Web Server: A software application that allows users to connect to a computer and request web pages from it. Most web servers also allow users to request not only files but also the output of programs that are executed at the time the user makes the request.
2. Common Gateway Interface (CGI): This refers to a standard protocol that web servers use to invoke a program that a user requests. It does not actually refer to a specific programming language, but can be implemented in almost any. CGI defines what environment variables are passed to the program (usually called a script) so that it can process a user's request properly. The most popular CGI programming language is called Perl (the Practical Extraction and Reporting Language). Perl can be obtained freely and runs on virtually any computing platform.
3. Web Page: Refers to an HTML (Hypertext Markup Language) document usually given to a web browser from a web server. The HTML document can either exist on the web server or be the output from a CGI program.
4. Dynamic Web Page: An HTML document that does not exist on the web server, rather, it is generated by a program and returned to the user when requested.
5. TCP/IP Socket: A software object that allows one computer to communicate with another. TCP/IP sockets are how all Internet servers communicate with Internet clients (like web browsers). They can also be used to allow SAS to communicate with a web server to return the output of a SAS program very quickly.

WEB-ENABLING YOUR SAS PROGRAMS

This paper will use the term "web-enabling" to refer to moving functionality that exists in Base/SAS, SAS/Macro, or SAS/AF programs or applications into an environment where it can be executed on another computer by user via a web browser. The user would presumably not have the SAS System installed, and would not specifically need it to be. You may already have Base/SAS reports using data steps and procedures that users run either through a command-line interface or through the SAS display manager on hosts that support it. Users who submit these programs typically need to be familiar with the SAS system and somewhat technical, especially if parameters need to be passed to the program or set in the code. Web-enabling your programs can afford you with several key advantages:

1. Deployment flexibility. You have a number of options on how to build a front end for your application or reports. You can provide a web page or Java applet that allows your users to provide input, and that content can be accessible to users across operating systems and browser types.
2. Less training/software required. Users do not need access to SAS software or the data on their local machines. If you have many users accessing your application, this can be very important. The web has become a very familiar environment in which computer users feel comfortable submitting information and retrieving results. By letting them use this familiar environment, you reduce the amount of training needed for your specific application.
3. Less SAS licenses are required. Though you do need a server license of SAS to accommodate many users querying SAS software on one machine, this will typically be less expensive than purchasing individual client licenses.

REQUIREMENTS

What software and hardware is required to build an environment to execute web-based SAS applications? You need a web server, SAS software running on either this same server or another physical machine, and a way to pass requests from the web server to the SAS System. SAS/Intrnet, the Institute's solution, uses CGI. The product that is the focus of this paper, Onyx, takes a more "open" approach, allowing communication between the web server and SAS by either CGI or other technologies including Sun's Java Servlets or Java Server Pages, Microsoft's Active Server Pages, and others. Many of these approaches are superior to CGI because they do not invoke a separate process for each request, allowing many more concurrent requests to be handled. Onyx offers a documented protocol for communication with the SAS system called the Onyx SAS Application Protocol (OSAP).

ABOUT ONYX

WHAT IS ONYX?

Onyx is a full-featured application server that can be used to control the SAS system in a unique way from any other software application, even on remote machines. Onyx fully and in a special way supports communication with the web. An advanced multi-threaded application server running in Java provides a transparent and load-balanced way to process SAS requests on any number of servers.

Onyx gives you the power to run your SAS applications using any kind of front-end you desire, whether it be web pages (HTML and a scripting language), Java applets, Client-side applications, or anything else. It can also run on any operating system that the SAS system can run in.

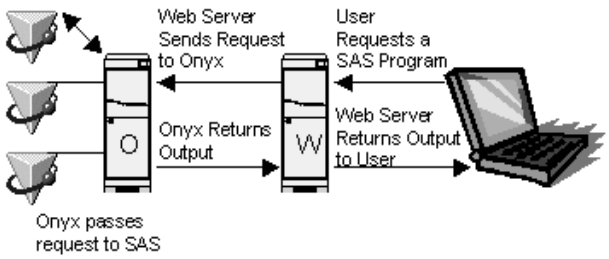
SOME FEATURES OF ONYX

- Includes an integrated, extremely fast, web server, making it very easy to develop and test SAS® code on individual PCs or laptops.
- Allows application developers to re-distribute their applications to sites that do not have Onyx licensed, giving you the freedom to require only base SAS® for your web applications.
- SAS® can be run from any number of different computers, even running different operating systems, with a load-balancing algorithm to distribute processing over the machines evenly.
- Comes with a built-in interpreter for the Onyx Dynamic HTML syntax which lets users use HTML editors to build pages that have embedded SAS® code within familiar <% %> tags.
- Is administered either via telnet or a web browser. Users can drop or add pooled SAS® sessions and check on the status of requests.
- Can easily be customized by SAS®/AF developers on the server side by adding new request types as SCL entries.
- Will run on any operating system that SAS® is licensed for.
- Requires nothing more than base SAS®.

Each of these features will be presented and discussed throughout the rest of this paper.

HOW ONYX WORKS

If you are familiar with the way in which SAS/Intrnet works, you should take special notice that Onyx has been designed with a different structure. Onyx includes a program called the "Application Dispatcher", which serves as the central nervous system and point of entry for making requests. The Application Dispatcher (hereafter referred to as simply the Dispatcher) functions almost exactly like a web server in that it waits for requests via the standard networking protocol TCP/IP, and allows a client to submit a request. In fact, the Dispatcher includes a built-in web server, which we will examine later. Here is an illustration to help you understand what happens when a user makes a request from Onyx:



This diagram illustrates how the various software applications that provide a link from the web to the SAS system function and in what order. It can also be misleading in that each step of the process is depicted as being handled by separate machines. In reality, every part of the process from the web browser to the SAS sessions can be run on one computer.

In Onyx terminology, SAS sessions that are configured to communicate with the Dispatcher are referred to as Drones, because they exist solely to do the bidding of the Dispatcher, their "queen". A significant difference between the architecture of Onyx and SAS/Intrnet arises from the fact that these Drones function as TCP/IP clients instead of servers. This means that each Drone connects to the Dispatcher and is immediately terminated if the Dispatcher application is halted. This can sometimes be beneficial to setting up a secure networking environment.

WHY SO MANY SAS SESSIONS?

You may wonder why several SAS sessions are depicted in the above diagram. The SAS System, up to the most recent version 8.2, has always been "single-threaded." This means that an individual SAS session is unable to run a procedure or data step and be waiting for another request at the same time. Nor is it able to run two separate data steps at once. This has the negative effect of requiring a separate SAS session for each concurrent request made to Onyx. If you had 10 users request a program at once, you would need 10 SAS sessions to serve these users instantly, or some users would have to wait until a SAS session finished processing the request of another user.

Unfortunately, an individual SAS session consumes a great deal of memory that cannot easily be controlled or limited without severely crippling the session. In order to allow many Drones to execute, even while each may consume a large amount of memory, Onyx allows the Drones to run on separate physical machines than the Dispatcher. In fact, you could even run Drones on machines all around the world via the Internet. The Dispatcher makes decisions about which computer and session to send a request to.

Even though you can run multiple SAS sessions that are all connected to the Dispatcher, you still may not be able to accommodate a large number of concurrent users. To solve this problem, the Dispatcher allows an option to be set that will cause users to wait for a specified length of time for an available session, after which they will receive a customizable busy message.

HOW IS ONYX DEVELOPED?

Only two programming languages are used in the development of Onyx, SCL and Java. In fact, they share so many similar features that they naturally work well together. Both SCL and Java can be compiled into operating system independent "byte-code", which allows multi-platform use. The syntax of the two is even strikingly similar. A key difference between them is the fact that Java applications can be run without purchasing Java while SCL applications require the SAS System. The simple CGI script included with Onyx is written in the ubiquitous scripting language called Perl. Perl is freely available and is installed on most web servers in the world.

WHAT INSTITUTE PRODUCTS CAN ONYX MIMIC?

Because Onyx provides a TCP/IP application server with no specific or forced client, it can be used in many different ways. The key comparison is, of course, SAS/Intrnet, but the functionality of SAS/Connect, WebAF and Integration Technologies can also be utilized with Onyx. The key feature of Onyx that enables this kind of connectivity is the Onyx SAS Application Protocol (OSAP).

ONYX SAS APPLICATION PROTOCOL

Virtually all Internet servers (web server, mail server, ftp server, etc.) use what is called a TCP/IP protocol to define the syntax of how clients and servers can understand each other. These protocols are typically created by panels of volunteers in documents called Request for Clarifications (RFCs). If application vendors use a standard protocol when designing a server, anyone can write a client that knows how to communicate with it. The same model has been used in developing Onyx. A custom, text-based protocol has been created so that anyone or anything can communicate with the SAS system in an easy and intuitive way. Such a protocol makes configuring firewalls easier as well because the syntax of messages can be anticipated.

What does an OSAP request look like? Well, it looks a lot like an HTTP request if you have ever seen one. Hypertext transfer protocol (HTTP) is the language that web browsers use to communicate with web servers. A typical request looks something like this:

```
GET /index.html HTTP/1.0
Host: 102.3.55.61
```

This instructs the web server to return the contents of the file named index.html to the host computer whose number is listed. An Onyx request looks like this:

```
OSAP/1.0 Request:
Program=/home/dward/run.sas
Session: 189317071541834809150562732 79
Content-Length: 2

State=NC
Product=Onyx
```

The first line (split into two to fit in a single column) always indicates that the client is making a request and includes the details of that request (the program run.sas). The concept of sessions (on the second line) will be introduced later. The remaining lines indicate input parameters that the user wishes to send to the program run.sas.

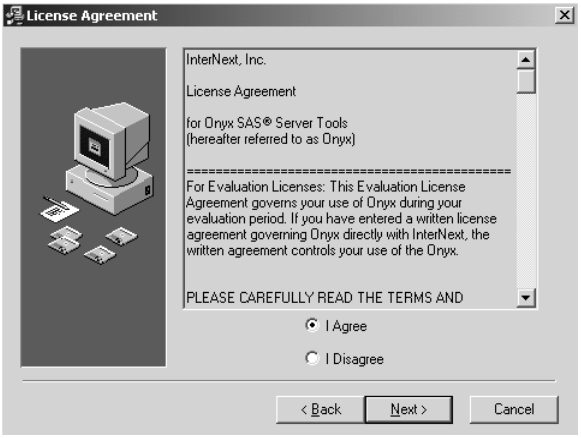
So what can you do with OSAP? Users can now telnet directly to the Dispatcher and submit requests. Onyx includes 5 built-in request types: PROGRAM (shown above), SQL, CODE, MACRO, and MACROVAR. Each one performs separate actions based on the content sent to it. The Program request type is used for web requests and would usually return HTML or graphic output. The SQL request type assumes that the content will include an SQL statement and the request details should include information about which format the user wants the results returned in. CODE allows users to submit either Base/SAS or SCL code directly to Onyx and have the results returned. MACRO allows users to request a SAS/Macro to be executed with the parameters named in the content of the request, while MACROVAR returns some or all macro variables from the SAS System. **SAS/AF developers can write custom request types that respond to OSAP requests in any way they choose, making Onyx infinitely extendable.**

INSTALLING ONYX

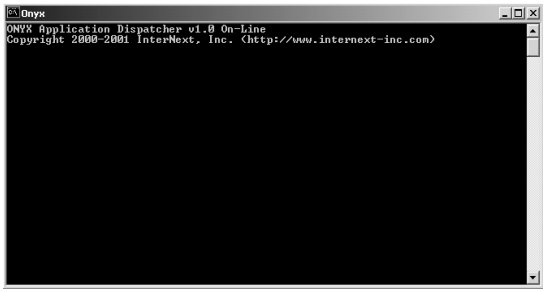
Onyx is very simple to install, particularly on Windows hosts. Simply run the Onyx installation program and follow the wizard that will guide you through setup. The installation wizard will perform the following actions:

1. Install the Java Runtime Environment, if it is not already installed
2. Copy all system files to the local computer
3. Modify configuration files with default settings and let the user change them if desired
4. Install a Windows NT/2000 service if desired

Here is a sample screen shot of the installation wizard:



After clicking on the Onyx shortcut that is placed on your desktop during installation, a command prompt window should show up in the Windows taskbar. Clicking on the window will reveal the following screen:



Once you see the message

```
ONYX Application Dispatcher v1.0 On-Line
Copyright 2000-2001 InterNext, Inc.
(http://www.internext-inc.com)
```

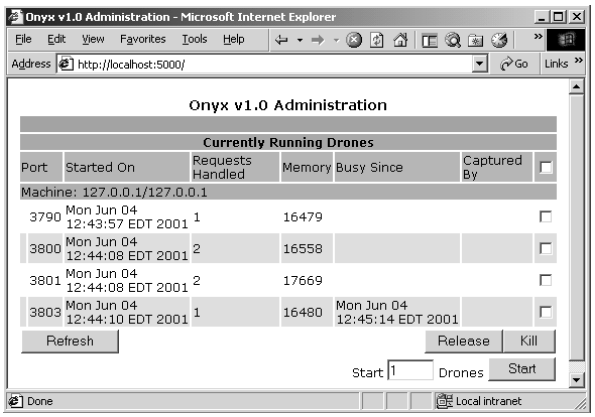
you know that Onyx is running. Onyx can be run even with the default settings. The test and demonstration programs will execute, all through the built-in web server. Once installed, simply navigate your browser to http://localhost:5000/ and enter the username "admin" and password "onyx". You will immediately see the Onyx web-

based administration utility, described in the next section.

ADMINISTERING ONYX

VIA THE WEB

Once you have Onyx installed and running, navigating your browser to http://localhost:5000/ as mentioned above will display the Onyx administration utility. It is from this interface that you can drop and add Drones (you can only add Drones running on the same machine as the Dispatcher) and check on the status of requests. A typical screen looks something like this:



Though you can't see it well at this size, this web page indicates that there are 4 Drones available. The number of requests each one has handled and the amount of memory each uses is displayed. Also indicated is a date/time value that the Drone began processing a request on. You can use this value to see if any Drones are currently busy (as the last one is in the screen shot). From this interface you can kill or drop existing Drones, add new ones (start new SAS sessions on the machine), and "release" Drones (discussed in detail later).

VIA TELNET

If administering the server through a web browser is not acceptable to you, you can choose to use the standard telnet utility to display a similar text-based interface. A sample telnet session:

```
>Enter Onyx Admin Command:
show
>
>                    Currently Running Drones
>
Address/Started/Requests/Memory/Busy/Captured
>=============================================
=======>Machine #0: 127.0.0.1/127.0.0.1
> #0: 1344/Fri Apr 06 13:08:57 EDT 2001/0/0//
> #1: 1347/Fri Apr 06 13:09:14 EDT 2001/0/0//
>
>Enter Onyx Admin Command:
kill 0.0/ (kill machine 0 drone 0)
kill 0.0/0.1 (kill machine 0 drones 0 and 1)
kill (kill all drones)
start 2 (start 2 drones)
start (start 1 drone)
release 0.0/ (release machine 0 drone 0)
```

In this case the administrator viewed available Drones, killed, started, and released several Drones.

USING ONYX

Enough super-techno babble. Let's get to some SAS programming! Onyx can be used to run existing SAS/Intrnet programs, or to create new programs that make use of more advanced features of Onyx. This paper will start with base/SAS programs then look at how to write SCL programs for use with the web.

BASE/SAS PROGRAMS

Our example Base/SAS programs will show how to pass form parameters to SAS, how to use ODS to generate advanced HTML output, how to use sessions, explain the special syntax known as Onyx Dynamic Html, and show unique debugging features of Onyx.

PASSING FORM PARAMETERS TO SAS

Just like SAS/Intrnet, HTML form parameters are sent to SAS as macro variables. In fact, web server environment variables and cookies are also sent as macro variables. Consider the following HTML form:

```
<FORM ACTION="report.sas" METHOD="GET">
<SELECT NAME="country">
<OPTION>USA<OPTION>Canada
<OPTION>Mexico
</SELECT> <INPUT TYPE="submit" value="Show
Report">
</FORM>
```

When this form is submitted to the program "report.sas", Onyx creates the following SAS macro variables:

```
GLOBAL _IP 127.0.0.1
GLOBAL _BROWSE Mozilla/4.0 (compatible;
MSIE 6.0b; Windows NT 5.0)
GLOBAL ONYXSESSIONID
369802410416145830052829111801727
GLOBAL COUNTRY Canada
GLOBAL ONYXSESSIONID_ C
GLOBAL _IP_ E
GLOBAL _BROWSE_ E
GLOBAL COUNTRY_ G
```

You will immediately notice that the form parameter named "country" has generated two macro variables, COUNTRY and COUNTRY_. COUNTRY holds the value chosen by the user and COUNTRY_ contains one letter indicating what type of information the macro variable COUNTRY holds. In this case the G indicates that it is form data sent via the GET method (indicated in the FORM tag). The other variables marked as E indicate that they are environment variables (notice the browser type). You can use these macro variables just as you would any other:

```
libname sasdata 'my-data-directory';
ods html file=_webout (dynamic);
proc means data=sasdata.sales;
  title "Sales summary for country
&country";
  where country="&country";
  var amount; run;
ods html close;
```

This simple proc means uses the macro variable to create a title and form a where clause. One item to note: using macro variables in this way poses a security risk. Since macro variable references with ampersands are compiled as part of the SAS code at run time, a malicious user could enter macro statements, unbalanced quotes, even their own procs or data steps directly into the variable. To check the contents of a parameter before using it you can use the symget()/symput() functions in a data step. These functions store the macro variables as data step variables which are immune to the same compiler dangers.

USING ODS TO GENERATE ADVANCED HTML OUTPUT

Since the birth of ODS in version 7 of the SAS System, there have been many conference papers, books, and tutorials explaining the details of how to use it. Instead of focusing on ODS this paper will simply present how to use it in conjunction with Onyx or SAS/Intrnet. A simple example was presented above. The proc means was enclosed in two ods statements, **ods html file**, and **ods html close**. Use the first statement to begin capturing procedure output and the second to finish capturing it. The keyword dynamic is important. It tells ODS to add a required HTTP header to the output it generates. See the SAS Online Doc for complete documentation on ODS. It includes many options and is a very powerful way to create HTML (and other formats) output.

USING SESSIONS

An inherent problem with web-based applications lies in the fact that they are "stateless". That is, each time a user requests a web page from a web server, the web server does not know it is the same person or "session" making another request. So how do programs on the server know that the same user is requesting a report that has just logged in? The simplest way that web applications track this information is through the use of cookies. Most of us are familiar with the cute term by now because of the issue of security. Web sites actually store information on our own hard drives and can read that information each time we request a page from their sites. Onyx uses only one cookie: the Onyx session ID. This value lets Onyx know which parameters and/or data sets correspond to the current user.

All information gathered during a session is stored in a directory on the server and assigned the libname SESSION. Thus, SAS programs can store data sets or other items in this libname and it will be available each time the user requests pages from the same browser and computer. Additionally, SAS macro variables are saved across requests in the same session if the special prefix _ONYXSESSION is used for each variable. Thus you could have one page that checks a username and password and if successful could store a session variable with the username. Any subsequent programs could check to make sure that session variable exists before granting rights to run the program. Sample code:

```
data _null_;
  length username password $50
         where $100;
  username=symget('username');
  password=symget('password');
  where='username='||quote(username)||
        ' and password='||
        quote(password);
  dsid=open(
    'sasdata.users(where=('||
    trim(where)||'))');
  if dsid then do;
    if attrn(dsid,'ANY') then
      call symput(
        '_onyxSessionUsername',
        username);
    rc=close(dsid);
  end;
run;
```

Another program could then check the value of &_onyxSessionUsername:

```
%macro report;
  %if %length(&_onyxSessionUsername)=0
%then %do;
    data _null_;
      file _webout mod;
      put 'Access Denied!';
    run;
  %end;
  %else %do;
    ** REPORT CODE HERE **;
  %end;
%mend;
%report;
```

USING ONYX DYNAMIC HTML

A unique and exciting feature of Onyx is the built-in support for the Onyx dynamic HTML syntax. This syntax lets programmers develop HTML pages with Base/SAS code embedded directly in the HTML. Simply enclose your SAS code in the now familiar <% %> tags (similar to Java Server Pages and Active Server Pages). Macro variables can also be resolved directly in HTML! Here is a sample of an ODHTML page:

```
<HTML>
<HEAD>
  <TITLE>My page</TITLE>
</HEAD>
Date/Time program executed:
%sysfunc(datetime(),datetime.)<br>
Date SAS session was started:
&sysdate9<br>

<%
proc sql noprint;
  select nobs into :nobs from
dictionary.tables where
    libname='DATA' and memname='CONTACTS';
quit;
%>
Observations: &nobs
Proc freq:
<%
ods html file=_webouta;
proc freq data=sasdata.contacts;
  table first last;
run;
ods html close;
%>
</HTML>
```

A major advantage to using this syntax is that developers can use standard web-page editors to build pages. Many commercial and free editors recognize the <% %> tags as server-side scripts and simply ignore the contents at design time. ODHTML also makes use of the natural iteration of the data step that allows HTML to be embedded directly into the data step itself. See the Onyx documentation for examples of this powerful feature.

DEBUGGING PROGRAMS WITH ONYX

Special consideration has been taken for debugging programs with Onyx. Simply include the following line of code in your program to return the SAS log to the browser:

```
%put NOTE: ONYX LOG;
```

Or if you would like to see the SAS log only if an error occurs:

```
%put NOTE: ONYX ERROR LOG;
```

After the program runs, the log is scanned for errors. If an error is found and the appropriate debugging directives are

found the log will be returned to the user. This is extremely helpful in testing your programs.

WRITING SCL PROGRAMS

Onyx fully supports the use of the SAS Component Language to develop web-based programming content. SCL is a rich language that allows rapid development of programs and is well suited for this kind of programming. We will take a look at obtaining form parameters via an SCL list, the structure of the Onyx Drone object, and how to use submit blocks effectively to generate HTML output or submit Base/SAS code.

OBTAINING FORM PARAMTERS FROM A LIST

Each time a user makes a request via the PROGRAM request type, SCL lists are generated that contain all parameters sent to Onyx. These values correspond to the macro variables created, in fact, they are used to create the macro variables. The list you can access is stored in the local environment list and is named simply ONYX. SCL code to obtain these values:

```
Dcl char username;
init:
  onyx=getiteml(envlist(),'ONYX');
  username=getnitemc(
           getniteml(onyx,'P'),
           'username');
return;
```

You can see from this example that the ONYX list is made up of sub-lists corresponding to the type of data being sent.

THE DRONE OBJECT

Because Onyx is developed using SCL, it is tightly integrated into the SAS/AF environment by exposing internal methods and properties of the Drone object to user programs. The Drone object is what powers each drone and contains a number of useful methods and properties that your programs can access. Two examples we will look are the methods clientData() and getVarS():

```
Dcl object drone;
Dcl char username password loggedin;
init:
  drone=getnitemo(envlist(),'Drone');
  ** GET SESSION VARIABLE LOGGEDIN **;
  drone.getVarS('loggedin',loggedin);

  ** GET USERNAME AND PASSWORD FROM FORM
**;
  drone.clientData(
    'username password',
     username,password);
  drone.clientData('G',
    'username password',
     username,password);
return;
```

Many other methods and properties are available. If you are interested, they are described in the Onyx documentation.

USING SUBMIT BLOCKS EFFECTIVELY

SCL submit blocks are a powerful feature of the SCL language. Most often, they are used to submit base SAS code or SQL code (hence the name "submit" block). But when used with no qualifier, the submit statement actually only places text into what is called the preview buffer. A buffer of text is stored in memory and the preview() function can manipulate that text. Submit blocks can also resolve SCL variables preceded by an ampersand. Here is a simple program that uses submit blocks to write an HTML page:

```
Dcl char date;
init:
control asis;
date=putn(datetime(),'datetime.');
submit;
<HTML>
My web page here - created on
&date
</HTML>
endsubmit;
rc=preview('file','_webout','append');
rc=preview('clear');
return;
```

A very important option when using submit blocks is to include the control asis statement (in bold above). This tells the submit blocks to leave the contents formatted as-is. Otherwise, it will attempt to format it optimally for SAS code, which may severely alter the HTML you wish to write. Submit blocks can be used inside of do loops and be executed conditionally, making them a powerful choice for HTML development.

ADVANCED FEATURES

CUSTOMIZING ONYX

Another exciting feature of Onyx is the ability for SAS/AF developers to change existing request types or add new ones. This means that new features can be added to Onyx. When customized, Onyx acts simply to connect the remote user to the SAS System and individual developers can decide how to act on the OSAP request. A number of built-in request types come with Onyx, described previously, which can be enhanced as desired.

USING OTHER CLIENTS THAN THE WEB

As mentioned on the first page of this paper, the use of the Application Dispatcher does not apply just to a CGI script on a web server. Users can telnet to the Dispatcher and enter OSAP requests directly, or other clients can use TCP/IP socket programming just like when reading from web sites. InterNext plans to develop a Java Servlet in the near future that will communicate with Onyx. Other products that could easily communicate with Onyx include Microsoft Office (through the use of Visual Basic), SAS (through the socket access method), or C/C++.

RE-DISTRIBUTING APPLICATIONS

A unique and powerful feature of Onyx is its re-distribution support. Built with the developers in mind, any license holder of Onyx can distribute applications to sites that do not have Onyx licensed. A special set of keys called app keys are generated that encrypt the program names, other program information, Onyx license holder information, and (optionally) the client SAS site ID. Thus you can develop web-based applications in SAS for clients that do not have any web product licensed.

Only SAS catalogs can contain app keys. If your applications makes use of Base/SAS programs (like in .sas files), place them in .source entries in a catalog. When your users request a .sas file from their web server, Onyx will automatically convert the reference into a .source reference and will be able to find your programs.

CUSTOMIZING ERROR MESSAGES

System error messages are taken from HTML files in the Onyx\Java\Errors\ directory. To customize the error messages change those files. AF developers can also add new error files and call them explicitly in their code by using Onyx.Main.Error.Scl. See the documentation for details.

CAPTURING DRONES

Onyx gives programmers the ability to "capture" a drone so that it can only be used by one session. You may want to capture a drone if you want to make sure a user will never have to wait for a session, or to save SCL lists or macro variables in memory or datasets or catalogs in the work directory. Captured drones are not cleaned up in between requests, so libnames, filenames, macro variables, etc. remain undeleted.

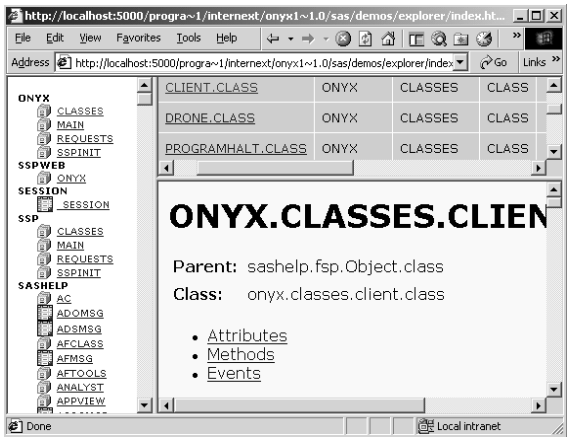
Since you are guaranteed the same session for each request, you can use SCL lists, but keep in mind that the lists may not get deleted properly when the capture is lost so you may consume unnecessary memory. The Application Dispatcher controls who has access to the captured drone. If no request has been sent to a captured drone for an indicated timeout period the capture will be released and a command is sent to the drone to clean itself up. The next time the client tries to use a captured drone they will potentially get another drone so you will need to check to make sure the drone is still captured by the intended user.

To capture a drone simply set the macro variable _onyxCapture to "Capture" or "Release" and the specified action will be performed:

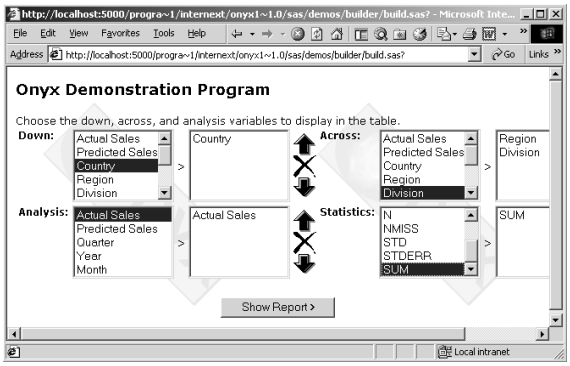
```
%let _onyxCapture=Capture; * Capture a
drone ;
%let _onyxCapture=Release; * Release a
drone ;
```

DEMONSTRATION APPLICATIONS

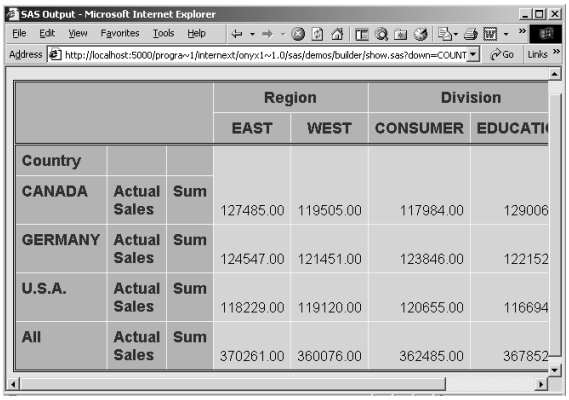
Onyx includes two demonstration applications with source code so that you can test Onyx and learn how to write web-based SAS programs. The first application is the Onyx explorer. This SCL application shows how to use SCL to develop web content. The explorer displays all data sets, catalogs, and catalog entries defined in a Drone. Upon clicking on individual items, users can view details such as proc contents, catalog entry contents, and actual observations from data sets. Here is a screen shot:



The second demonstration program is a simple report builder using data included with the SAS System installation. Users can choose which variables they would like displayed down the table, across the table, which analysis variables, and what statistics they would like. A proc tabulate is executed with the users choices:



And the output of the proc tabulate:



			Region		Division	
			EAST	WEST	CONSUMER	EDUCATI
Country						
CANADA	Actual Sales	Sum	127485.00	119505.00	117984.00	129006
GERMANY	Actual Sales	Sum	124547.00	121451.00	123846.00	122152
U.S.A.	Actual Sales	Sum	118229.00	119120.00	120655.00	116694
All	Actual Sales	Sum	370261.00	360076.00	362485.00	367852

CONCLUSION

Though many organizations are unable to obtain SAS/Intrnet, there are still alternatives for dynamic publishing of SAS information via the web. One such alternative, a fully supported product, is Onyx. This paper has presented an overview of Onyx and examples that take advantage of unique features it has to offer. Whether it is via SAS/Intrnet, Onyx, or some other method, you are encouraged to begin developing web-based content with the SAS System!

OBTAINING ONYX

We will have CDs available at the conference, or you can obtain Onyx software through the InterNext website at http://www.internext-inc.com. You can both download the software and obtain an evaluation key that will allow you to develop SAS programs for 30 days. You can contact the author with further questions.

ACKNOWLEDGMENTS

SAS and SAS product names are registered trademarks of the SAS Institute. Other trademarks are the properties of their respective owners.

CONTACT INFORMATION

Your comments and questions are valued and encouraged.
Contact the author at:

David L. Ward
InterNext, Inc.
254 Resnik Ct.
Somerset, NJ 08873
Work Phone: (732) 745-9823
Email (preferred): dward@sashelp.com

INTRODUCTION TO SAS

SECTION CHAIRS

Imelda Go
Lexington County School District One

Andrew T. Kulogowski
Neilsen Media Research

Thomas Winn, Jr.
Texas State Auditor's Office

INTRODUCTION TO SAS

Introduction to the SAS® Programming Language

Thomas J. Winn, Jr.
Texas State Auditor's Office, Austin, Texas

Abstract

This short paper is first in the "Introduction to SAS" sequence of papers. It includes some of a programmer's first steps toward learning SAS. The paper provides a brief overview of the SAS System, as well as an outline of the basic structure of the SAS programming language. In particular, it covers preliminary concepts regarding DATA and PROC steps, the different kinds of data, and SAS data files. This paper, and the associated presentation, is intended to provide a foundation for the next step of learning SAS, that of creating SAS data sets, using the INPUT statement and INFILE options.

A Very Brief History of SAS

While a graduate student in statistics at NCSU, Jim Goodnight wrote a computer program for analyzing agricultural data. After a few years, Jim's application had attracted a diverse and loyal following among its users, and the program's data management and reporting capabilities had expanded beyond Jim's original intentions. In 1976, Jim decided to work at developing and marketing his product on a full time basis, and SAS Institute was founded. Since its beginning, a distinguishing feature of the company has been its attentiveness to users of the software. Today, SAS Institute is the world's largest privately-held software company, and Dr. Jim Goodnight is its CEO. He continues to be actively involved as a developer of SAS System software.

The SAS System

The SAS System is an integrated suite of information delivery software products. It is a library of modular components, that are tied together by a central supervisory program. Applications of the SAS System include executive information systems; data entry, retrieval, and management; report writing and graphics; statistical and mathematical analysis; business planning; forecasting, and decision support; operations research and project management; statistical quality improvement; computer performance evaluation; and applications development.

Originally, "SAS" stood for "Statistical Analysis System", however, the applicability of SAS has grown far beyond that single purpose. Today, SAS is not an acronym for anything. Besides statistics, elements of the SAS System also include web enablement, data mining, data warehousing, and business intelligence solutions, a variety of industry-specific and business functional-area products (for managing organization, customers, and suppliers), as well as state-of-the-art applications development tools. Today, the SAS System is a comprehensive system for data management and analysis.

SAS software collects data from almost any platform and data format; it cleans and transforms the data into information which decision makers will understand; and it stores the information in an open and efficient data storage structure. To explore information, SAS software provides multidimensional data analysis, query and reporting, executive information systems, data mining, data visualization, and applications development capabilities. SAS solutions are client/server- and Web-enabled. SAS says about itself, "The Power to Know.™"

Components of the SAS System include (partial listing):

- Base SAS
- SAS/ACCESS
- SAS/AF
- SAS/ASSIST
- SAS/CONNECT
- SAS/EIS
- SAS/ETS
- SAS/FSP
- SAS/GIS
- SAS/GRAPH
- SAS/IML
- SAS/INSIGHT
- SAS/IntrNet
- SAS/LAB
- SAS/MDDB Server
- SAS/OR
- SAS/QC
- SAS/SHARE
- SAS/SPECTRAVIEW
- SAS/STAT
- SAS/TOOLKIT
- SAS/AppDev Studio
- SAS/Enterprise Guide
- SAS/Enterprise Miner
- SAS Universal ODBC Driver
- SAS/Warehouse Administrator

The "Introduction to SAS" section is designed to cover the fundamental elements of the SAS programming language. It includes elementary DATA step, and Base/SAS PROCedure programming.

Introduction to SAS Programming

All SAS jobs are a sequence of SAS steps, which are made up of instructions, which are called SAS statements. There are only two kinds of SAS steps: DATA steps are used to read, edit, and transform data (raw data or SAS data files), to prepare SAS data sets, PROC steps are ready-to-use procedures which analyze or process SAS data sets. In general, data must be in a SAS data file before they can be processed by SAS procedures.

Without going into the details at this time, here is a skeletal example of a SAS job:

```
DATA STUDENTS;
        INPUT  NAME $ 1-14  SEX $ 15
                SECTION $ 17-19 GRADE;
        DATALINES;
        . . . data lines . . .
        ;
PROC SORT DATA=STUDENTS;
        BY SECTION  DESCENDING GRADE;
 PROC PRINT  DATA=STUDENTS;
        BY SECTION;
 RUN;
```

There are two kinds of SAS data sets: SAS data files (or tables), and SAS data views. A SAS data file contains: the descriptor portion, which provides SAS procedures and some DATA step statements with descriptive information (data set attributes and variable attributes) about the data , and the data portion, a rectangular structure containing the data values, with rows (customarily called observations), and columns (customarily called variables); and which is passed to most procedures, observation by observation. A SAS catalog is a type of SAS file which stores many different types of information used by the SAS System. All SAS files reside in a SAS data library.

The SAS System processes the program in two steps: (1) it compiles the program, and (2) it executes the program. When the program is compiled, a program data vector (PDV) is constructed for each DATA step. It is an area of memory which includes all variables which are referenced either explicitly or implicitly in the DATA step.

At execution time, the PDV is the location where the current working values are stored as they are processed by the DATA step. Variables are added to the PDV sequentially as they are encountered during parsing and interpretation of SAS source statements. Each step (DATA or PROC) is compiled and executed separately, in sequence. And at execution time within each DATA step, each observation is processed iteratively through all of the SAS programming statements of the DATA step.

SAS procedures (PROCs) are programs that are designed to perform specific data processing and analysis tasks on SAS data sets. Base/SAS procedures fall into the following categories:
SAS Utilities -- APPEND, CATALOG, CIMPORT, COMPARE, CONTENTS, COPY, CPORT, DATASETS, DBCSTAB, DISPLAY, EXPLODE, EXPORT, FORMAT, FSLIST, IMPORT, OPTIONS, PMENU, PRINTTO, RANK, REGISTRY, SORT, SQL, STANDARD, TRANSPOSE, TRANTAB;
Descriptive Statistics -- CORR, FREQ, MEANS, SQL, SUMMARY, TABULATE, UNIVARIATE;
Reporting -- CALENDAR, CHART, FORMS, MEANS, PLOT, PRINT, REPORT, SQL, SUMMARY, TABULATE, TIMEPLOT.

Creating SAS Data Files

Since SAS procedures can operate only on SAS data sets, then the first step in processing any raw data using SAS will be to transform them into a SAS data set. Whenever the SAS System creates a SAS DATA file, it does the following:

1. it reads the DATA statement, creates the structure of a SAS data set, and marks the statement as the place to begin the processing of each line of data;
2. it uses the description of the data in the INPUT statement to read the data line, and to produce an observation;
3. it uses the observation to execute any other SAS statements that are in the DATA step;
4. it adds the observation to the data set being created; and
5. it returns to the beginning of the DATA step for the processing of the next observation.

All SAS DATA step statements are executed once for each observation.
All SAS statements begin with an identifying keyword, and end with a semicolon. SAS statements are free-format.

They can begin anywhere, and end anywhere. A single statement may continue over several lines. Several statements may be on a single line. Blanks (as many as desired) are used to separate fields. Other special characters also may be used to separate fields.

The data portion of a SAS data file is a collection of data values arranged in a rectangular table. The rows in the table are called observations. The columns in the table are called variables. There are <u>two</u> kinds of variables: character variables, and numeric variables. Each variable has a name.

There are rules for naming SAS data sets and variables: 1 to 32 characters in length (8 character maximum in Version 6 and earlier versions), start with A-Z or _ (underscore), continue with letters, numbers, or underscores. It is recommended that you choose meaningful names.

Character data can consist of up to 32,767 characters (max. of 200 characters in Version 6 and earlier versions). Character values may include letters, numbers, blanks and special characters, although generally you should not include any semicolons within the data. Numeric data values must be numbers, and they may be preceded by a + or -. Unless the data are being read using a special SAS informat, do not include commas or dollar signs in numeric data values. SAS assigns the value of a decimal point (“.”) to missing numeric values, and a blank (“ ”) to missing character values. You can enter these values into your data to indicate missing values.

Every SAS data set has a name and is physically stored on some type of media (disk, tape, etc.). In simple jobs, the SAS data sets are stored on temporary space, but they can be stored “permanently”. A temporary SAS data set exists only for the duration of the current SAS job, or interactive SAS session. A permanent SAS data set exists after the end of the current SAS job or interactive SAS session. Both types of SAS data sets have two-level names, of the form libref.data-set-name, where libref is a reference to the name of a SAS data library (a collection of SAS files).

With temporary SAS data sets, the SAS System automatically assigns the libref WORK and you specify the data set name. When you create a permanent SAS data set, you must specify both the libref and the data set name. The SAS System does not assign the libref for you. Although there are other methods for certain operating environments, the LIBNAME statement is the most universal method of assigning a libref. The general form is

```
LIBNAME libref ‘SAS-data-library’;
```

Here is an example of a LIBNAME statement in the Windows environment:

```
LIBNAME mylib1 ‘C:\mySASlib’;
```

Here is an example of a LIBNAME statement in certain mainframe environments:

```
LIBNAME mylib2 'data.set.name';
```

Here is an example of reading data from a "permanent" SAS data set (in the SAS data library whose previously defined libref is 'MYLIB'):

```
DATA EXAMPLE;
      SET MYLIB.STUFF;
```

Here is an example of creating a "permanent" SAS data set (also in the SAS data library whose previously defined libref is 'MYLIB'):

```
DATA MYLIB.TESTDATA;
      SET SAMPLE1;
```

A SAS DATA statement instructs the SAS System to create and name a SAS data set. It has the general syntax:

```
DATA data-set-name-1  (options-1)
      data-set-name-2  (options-2)
            . . .
      data-set-name-k  (options-k) ;
```

Many SAS data sets can be created in a single DATA step. DATA step options include such things as: DROP= , IN= , FIRSTOBS= , KEEP= , OBS= , RENAME= , WHERE= , and others.

The two major functions of the DATA statement are: to signal the beginning of the DATA step, and to name the data set(s) being created.

When creating temporary SAS data sets, the data set name can be supplied by the programmer:

```
DATA STUDENTS;
      INPUT  NAME $ 1-14  SEX $ 15
             SECTION $ 17-19 GRADE;
      DATALINES;
      . . . data lines . . .      ;
```

or, if the name is omitted in the DATA statement, the SAS System will provide a name (DATA1, DATA2, etc.):

```
DATA ;
      INPUT  NAME $ 1-14  SEX $ 15
             SECTION $ 17-19 GRADE;
      DATALINES;
      . . . data lines . . .              ;
```

A Few Words About Working With Dates and Times Using SAS

Whenever SAS reads date value inputs, it converts them into integers. SAS dates are positive or negative integers representing the number of elapsed days between January 1, 1960 and the specified date. Similarly, SAS converts time values into the number of seconds since midnight of the current day. SAS datetime values are the number of seconds since midnight on January 1, 1960. Since dates and times are numeric entities, one may use ordinary arithmetic to determine elapsed time, or future/past dates and times.

For examples,

```
AGEDAYS = THISDATE - BIRTHDATE;
AGEYRS = AGEDAYS / 365.25;
```

You may use date constants or time constants in a SAS expression by writing the date or time enclosed in quotes, and followed by a D (date), a T (time), or DT (date:time).

```
THISDATE = '20Aug2001'D;
SLEEPTIME = '23:59:59.9'T;
FAISDODO = '21Aug2001 20:30'DT;
```

To read data that are date or time values, SAS has a variety of informats. To write date or time values in reports, SAS has numerous formats. SAS also has several special functions for working with date or time values. We'll learn more about informats, formats, and functions in another presentation.

Summary

This short paper included some of a programmer's first steps toward learning about the SAS programming language. In particular, it covered the following items: an overview of the SAS System, a few fundamental ideas regarding SAS data sets, some preliminary concepts regarding DATA and PROC steps, and the different kinds of data in SAS.

Suggested References:

Ronald P. Cody & Raymond Pass, SAS Programming By Example (1995)

Lora D. Delwiche & Susan J. Slaughter, The Little SAS Book: A Primer, Second Edition (1998)

Frank DiIorio, SAS Applications Programming: A Gentle Introduction

SAS Institute Inc., SAS OnlineDoc, Version 8

SAS Institute Inc., Getting Started With the SAS System, Version 8

SAS Institute Inc., SAS Language Reference: Concepts, Version 8

SAS Institute Inc., SAS Language Reference: Dictionary, Version 8, Volumes 1 and 2

SAS Institute Inc., SAS Procedures Guide, Version 8, Volumes 1 and 2

Author Information.

Tom Winn
Texas State Auditor's Office
P.O. Box 12067
Austin, TX 78711-2067

phone: 512 / 936-9735
e-mail: twinn@sao.state.tx.us

The Input Statement: Where It's @

Ronald Cody, Ed.D., Robert Wood Johnson Medical School, Piscataway, NJ

Introduction

One of the most powerful features of SAS software is the ability to read data in almost any form. For example, you can have data values separated by blanks or other delimiters or you can arrange your data in columns, using one or more lines of data for each subject. You can also read selected data fields and then decide how to read the remaining data values.

This tutorial will give you an overview of the immense power and flexibility of the SAS INPUT statement.

Reading Space Delimited Data

A common form of data entry is to separate each data value by one or more spaces. This is handy for small data sets that are entered by hand, especially for test purposes. This arrangement of data is often called "list directed" data. The rule here is that you must specify all the variables in the data lines and all the data values must be separated by one or more spaces. You must also indicate which variables are to be read as character data. Look at the following example:

```
***LIST DIRECTED INPUT;
DATA LIST;
   INPUT X Y A $ Z;
DATALINES;
1 2 HELLO 3
4     5     GOODBYE   6
;
PROC PRINT DATA=LIST;
   TITLE 'LIST DIRECTED INPUT';
RUN;
```

Notice that you need to list the variable names on the INPUT statement and to place a dollar sign ($) after any variable that is to hold character values. Also notice that the second line of data which has multiple spaces between each data value causes no problems at all.

Delimiters Other Than Spaces

With just a small change to the program, you can indicate any delimiter you like, in place of the default blank. To understand how this works, we need to jump ahead a bit to see how a SAS program reads data from an external data file (as opposed to data following a DATALINES statement). You do this by including an INFILE statement which tells the program where to find the data. The INFILE statement has several options that provide additional information on how to read these external data lines, one of them being an option to define a delimiter other than a blank. The option has the form: DLM= '*your_delimiter*'. Since we want to demonstrate this option with "in-stream" data, we use the reserved fileref (file reference) DATALINES. This allows you to supply INFILE options with "in-stream" data. Here is the program:

```
***OTHER DELIMITERS;
DATA DELIM;
   INFILE DATALINES DLM='#';
   INPUT X Y A $ Z;
DATALINES;
1#2#HELLO#3
4   #   5   #  GOODBYE  # 6
;
PROC PRINT DATA=DELIM;
   TITLE 'OTHER DELIMITERS';
RUN;
```

In this example, we use a number sign (#) as the data delimiter. The INFILE statement uses the reserved fileref DATALINES followed by the DLM= option.

Special Case of Comma Delimited Files

There is a common data arrangement used by many personal computer applications. That is, to separate data values by commas and to place string values in double quotes. Furthermore, two commas together indicate that there is a missing value. The INFILE option DSD takes care of all of these features. Besides allowing commas as the data delimiter, this option reads character strings enclosed in

double quotes and strips off the quotes before assigning the value to the character variable. It also allows you to include commas within a character string. Finally, two commas together are interpreted as a missing value. The program that follows demonstrates all these features:

```
***SPECIAL COMMA DELIMITED FORMAT;
DATA SPECIAL;
   INFILE DATALINES DSD;
   INPUT X Y A $ Z;
DATALINES;
1,2,HELLO,3
4  ,   5  ,   GOODBYE ,  6
7,,"HI THERE",8
9,10,"HI,THERE",11
;
PROC PRINT DATA=SPECIAL;
   TITLE 'SPECIAL COMMA DELIMITED
FORMAT';
RUN;
```

To see that this program is working as expected, here is the output from PROC PRINT:

```
Special Comma Delimited Format

OBS    X     Y    A           Z

 1     1     2    HELLO       3
 2     4     5    GOODBYE     6
 3     7     .    HI THERE    8
 4     9    10    HI,THERE   11
```

When you use the DSD INFILE option, the default delimiter is a comma. You may use the DSD and DLM= options together to allow all the features just discussed but with a delimiter other than a comma.

Data Arranged in Columns

One of the most common forms of data entry is to arrange the data values in specified columns. This has several advantages over space or comma delimited data. First, you can pack more data values together without wasting space. Second, you can read only those columns of data that you want, and third, you can read the data values in any order you choose. Let's look at a SAS program that reads data arranged in columns:

```
***COLUMN INPUT;
DATA COL1;
   INPUT X    1-2
         Y      3
         A $ 4-10
         Z     11;
DATALINES;
 12HELLO  3
4 5GOODBYE6
;
PROC PRINT DATA=COL1;
   TITLE 'COLUMN INPUT';
RUN;
```

The rules are very simple. You list each of the variable names followed by the starting and ending columns (or just the starting column if there is only one column). You also place a dollar sign after any variable name that will hold character data. Notice the value of X in the two lines of data in this example. It doesn't matter whether this value is right adjusted (placed in the right-most columns of the field) or not. Good programming practice dictates that numbers should be right adjusted but SAS will read numbers correctly regardless.

Reading Only Selected Variables

As we mentioned earlier, once you have arranged your data values in columns, you can read only those variables that you need. So, using the same data lines as above, here is a program that only reads values for variables X and Z:

```
***COLUMN INPUT (SELECTED VARIABLES);
DATA COL2;
   INPUT X 1-2
         Z 11;
DATALINES;
 12HELLO  3
4 5GOODBYE6
;
PROC PRINT DATA=COL2;
   TITLE 'COLUMN INPUT';
RUN;
```

It's just as easy as that. You can also read these variables in any order you choose as demonstrated in the program below:

```
***COLUMN INPUT (DIFFERENT ORDER);
DATA COL3;
   INPUT Y      3
         A $ 4-10
         Z     11
         X    1-2;
DATALINES;
 12HELLO  3
4 5GOODBYE6
;
PROC PRINT DATA=COL3;
   TITLE 'COLUMN INPUT';
RUN;
```

Using Pointers and INFORMATS to Read Data

As an alternative to using column specifications, you can use column pointers and INFORMATS. This method is actually more flexible than using column specifications since you can use a SAS INFORMAT or a user-defined INFORMAT to specify how a data value is to be read. The example that follows uses almost the same data arrangement as the program that used column specifications except for the addition of a date value. We added that to demonstrate the advantage of this method.

```
***POINTERS AND INFORMATS;
DATA INFORM1;
   INPUT @1  X    2.
         @3  Y    1.
         @4  A    $7.
         @11 Z    1.
         @12 DATE MMDDYY10.;
   FORMAT DATE DATE9.;
DATALINES;
 12HELLO  310/21/1946
4 5GOODBYE611/12/1997
;
PROC PRINT DATA=INFORM1;
   TITLE 'POINTERS AND INFORMATS';
RUN;
```

The *@n* symbols are columns pointers. For example, @3 says to move to column 3. The INFORMAT following the variable name tells the program how many columns to read and how to read the data value. The INFORMAT n. indicates a numeric variable occupying n columns. The $n. INFORMAT is used for character variables and the MMDDYY10. INFORMAT converts dates in the form MM/DD/YYYY to a SAS date value (that's a topic for another talk). We chose MMDDYY10. instead of the more traditional MMDDYY8. because the year 2000 is fast approaching and, even with the YEARCUTOFF option, it is a good idea to use 4-digit years.

Using INFORMATS with List-Directed Input

You may want to read list-directed or some form of delimited data and still provide an INFORMAT. For example, you may want to read a character variable more than 8 bytes in length or one of the data values might be a date which needs an INFORMAT to be read correctly. Here comes the colon modifer to the rescue! By placing a colon (:) after the variable name, you can then supply an INFORMAT to be used. The program will search for a delimiter and begin reading the first non-blank data value according to the INFORMAT you supply. Look at the following example:

```
***USING INFORMATS: LIST DIRECTED
   DATA (COLON MODIFIER);
DATA COLON;
   INPUT X :    2.
         Y :    1.
         A :    $11.
         Z :    1.
         DATE : MMDDYY10.;
   FORMAT DATE DATE9.;
DATALINES;
1 2 HELLO 3 10/21/1946
4   5   ARRIVEDERCI  6  11/12/1997
;
PROC PRINT DATA=COLON;
   TITLE 'INFORMATS: COLON MODIFIER';
RUN;
```

Notice that variable A is now 11 bytes and the variable DATE will be a true SAS date.

An Alternate Method for Supplying INFORMATS

An alternative to the program above is preceed the INPUT statement with an INFORMAT statement, associating each of the variables with an INFORMAT. The program below will

produce exactly the same data set as the one above. Which method you choose is up to you.

```
***USING INFORMATS: LIST DIRECTED
   DATA (INFORMAT STATEMENT);
DATA INFORM2;
   INFORMAT X 2. Y Z 1. A $11.
            DATE MMDDYY10.;
   INPUT X Y A Z DATE;
   FORMAT DATE DATE9.;
DATALINES;
1 2 HELLO 3 10/21/1946
4   5    ARRIVEDERCI   6   11/12/1997
;
PROC PRINT DATA=INFORM2;
   TITLE 'INFORMAT STATEMENT';
RUN;
```

Space Delimited Data Values Containing Blanks

What happens if you are using blanks as data delimiters and you have a character value that contains a blank, such as a first and last name? By replacing the colon modifer with an ampersand (&), the system will continue reading a character value, even if it contains single blanks. The program will know that a data value is finished when it encounters two or more blanks. Notice variable A in the program below and the values of "HELLO THERE" and "A BIENTOT" in the data lines. When you use the ampersand modifier, be sure to remember to follow the variable with two or more blanks.

```
***USING INFORMATS: LIST DIRECTED
   DATA (AMPERSAND MODIFIER);
DATA AMPER;
   INPUT X : 2.
         Y : 1.
         A & $11.
         Z : 1.;
DATALINES;
1 2 HELLO THERE  3
4     5     A BIENTOT   6
;

PROC PRINT DATA=AMPER;
   TITLE 'AMPERSAND MODIFIER';
RUN;
```

A Shortcut Way of Specifying Variable Names and INFORMATS

First, look at the rather long and inelegant program below:

```
***WITHOUT VARIABLE AND INFORMAT
   LISTS;
DATA NOINLIST;
   INPUT @1 Q1 1.
         @2 Q2 1.
         @3 Q3 1.
         @4 Q4 1.
         @5 Q5 1.
         @6 A  $1.
         @7 B  $1.
         @8 C  $1.;
DATALINES;
12345XYZ
;
PROC PRINT DATA=NOINLIST;
   TITLE 'WITHOUT VARIABLE AND
INFORMAT LISTS';
RUN;
```

Even beginning SAS programmers know that whenever a program gets tedious to write, there is usually a better and shorter way to accomplish the same thing. In this case, you can use a variable list and an INFORMAT list to shorten the program. Take a look at this program:

```
***VARIABLE AND INFORMAT LISTS (1);
DATA INLIST1;
   INPUT @1 (Q1-Q5 A B C)
            (5*1. 3*$1.);
DATALINES;
12345XYZ
;
PROC PRINT DATA=INLIST1;
   TITLE 'VARIABLE LISTS (1)';
RUN;
```

You can place a list of variables, including the base*n*-base*m* notation (Q1-Q5 for example), in parentheses followed by a list of INFORMATS, also placed in parentheses. The notation 5*1. or 3*$1. means to repeat the INFORMAT 5 or 3 times respectively. Each variable in the variable list is read with the corresponding INFORMAT in the INFORMAT list. If the INFORMAT list is shorter than the variable list, the program will "recycle" the INFORMAT list, that is, go back to the first INFORMAT in the list and go through the list as many times as

necessary. You can take advantage of this feature to simplify the program like this:

```
***VARIABLE AND INFORMAT LISTS (2);
DATA INLIST2;
   INPUT @1 (Q1-Q5)(1.)
         @6 (A B C)($1.);
DATALINES;
12345XYZ
;
PROC PRINT DATA=INLIST2;
   TITLE 'VARIABLE LISTS (2)';
RUN;
```

By grouping variables together that use the same INFORMAT, you can list the INFORMAT just once and it will be used for each of the variables in the list.

Skipping Around Using Relative Column Pointers

Suppose you have several X,Y pairs (X1,Y1; X2,Y2; and X3,Y3 for example). You might think the only way to read these data would be an INPUT statement that looked like this:

```
INPUT X1 1 Y1 2 X2 3 Y2 4 X3 5 Y3 6;

OR

INPUT (X1 Y1 X2 Y2 X3 Y3)(1.);
```

Well, there is an easier way. By using a relative columns pointer (a + sign), you can move the column pointer right or left, relative to its last position. You can use this to read each of the X-values first, skipping over the columns occupied by the Y-values and then go back to column 2 (where the Y-values start) and read all the Y-values, skipping over the columns occupied by the X's. Here is the program:

```
***RELATIVE COLUMN POINTERS;
DATA RELATIVE;
   INPUT @1  (X1-X3)(1. + 1)
         @2  (Y1-Y3)(1. + 1);
DATALINES;
123456
;
PROC PRINT DATA=RELATIVE;
   TITLE 'RELATIVE COLUMN POINTERS';
RUN;
```

The INPUT statement above says to start reading in column 1 (@1) and then read a single digit value (1.) and then skip a column (+1). Do this for all the X-values. When you are finished, go back to column 2 (@2) and read the Y-values in the same way. If you had more than three X's and Y's, this technique could save you lots of typing.

Using a Text Pointer

There is an obscure but **really** useful way to read data values when you are unsure of where to find the data. This may seem impossible, but using a text pointer can sometimes solve this problem. A text pointer is of the form:

```
@"text_string" variable_name
```

The column pointer will be placed to the right of the text string (if found) and the next value on that line will be read into variable_name. An example should make this clear. Look at the following program:

```
DATA TEST;
   INPUT @ "XYZ" VALUE;
DATALINES;
THIS LINE HAS XYZ 76 NUMBERS
NONE ON THIS LINE
XYZ 20
```

The first line of data contains the string "XYZ," so the variable called VALUE will be 76. In the second line, there is no "XYZ" so no data values are read. Finally, in line three, VALUE will be 20. This method of input is especially useful when you are trying to extract numbers from SAS output (or output from some other program) where you can choose some key word that you know comes right before the value you want. Before ODS became available in version 8, using a text pointer with PROC PRINTTO was especially useful.

Reading Data from an External File

So far, all our examples used data lines "in-stream" or part of the program. Most SAS programmers, especially when large amounts

of data are involved, keep the data separately from the program. It is quite simple to have the program read data from an external raw data file instead of "in-stream" data. First, you use an INFILE statement to point to the data file and, second, you remove the DATALINES statement. It's as easy as that.

There are two ways that an INFILE statement can point to a file. The first is to name the file directly in the INFILE statement as follows:

```
***READING FROM AN EXTERNAL FILE
   (METHOD 1);
DATA EXTERN1;
   INFILE 'C:\SASTALKS\DATA1';
   INPUT X Y A $ Z;
RUN;

PROC PRINT DATA=EXTERN1;
   TITLE 'DATA IN AN EXTERNAL FILE';
RUN;
```

Notice that the file name is place in quotes (single or double). If the file is in the same subdirectory as the program, you can omit the full path description and just supply the file name. We prefer that you include the entire path and file name as above so that the program is transportable and can be run from another subdirectory without changes. A second method of supplying the file name is to first define a fileref (file reference) with a FILENAME statement. This fileref is then named on the INFILE statement. The very important difference between this method and the previous method is that the fileref is not placed in quotes on the INFILE statement. Here is an example of this method of specifying an external file:

```
***READING FROM AN EXTERNAL FILE
   (METHOD 2);
DATA EXTERN2;
   FILENAME PAT 'C:\SASTALKS\DATA1';
   INFILE PAT;
   INPUT X Y A $ Z;
RUN;

PROC PRINT DATA=EXTERN2;
   TITLE 'DATA IN AN EXTERNAL FILE';
RUN;
```

INFILE Options

Whether you are reading data following a DATALINES statement or from an external file, you may need to exercise some control on how the data values are read. There are a number of INFILE options that are useful. We will only discuss a few here. If you read complicated data structures or read data from multiple files, you will want to investigate all the possible INFILE options available to you.

The first option we will show you concerns data lines, either "in-stream" or from external files, that may be missing data values at the end of a line of data. In the example below, line 2 only contains 3 values (for X, Y, and A). When the program attempts to read this list-directed data, it goes to the third line to find a value for Z. Then, when the data step iterates again, the pointer tries to move to the next record but meets an end-of-file instead and the data step stops. To see this clearly, look at the output from PROC PRINT shown following the program.

```
***INFILE OPTIONS;
DATA INOPT1;
   ***PROGRAM WITHOUT MISSOVER;
   INPUT X Y A $ Z;
DATALINES;
1 2 HELLO 3
4 5 GOODBYE
7 8 LAST 9
;
PROC PRINT DATA=INOPT1;
   TITLE 'INFILE OPTIONS';
RUN;
```

```
Infile Options

OBS    X    Y       A       Z

 1     1    2    HELLO      3
 2     4    5    GOODBYE    7
```

To solve the problem of missing data at the end of a line, when using list-directed data entry, use an INFILE option called MISSOVER. The program below uses the reserved fileref DATALINES on the INFILE statement. If you are reading data from an external file, the MISSOVER option is used in the same way.

This option instructs the program to set any variables to a missing value if the end of a line (or record) is reached and values have not been read for all the variables in the INPUT list. Look at the program below and the resulting output.

```
***INFILE OPTIONS: MISSOVER;
DATA INOPT2;
   ***PROGRAM WITH MISSOVER;
   INFILE DATALINES MISSOVER;
   INPUT X Y A $ Z;
DATALINES;
1 2 HELLO 3
4 5 GOODBYE
7 8 LAST 9
;
PROC PRINT DATA=INOPT2;
   TITLE 'INFILE OPTIONS';
RUN;
```

```
Infile Options

OBS    X    Y    A          Z

 1     1    2    HELLO      3
 2     4    5    GOODBYE    .
 3     7    8    LAST       9
```

With column oriented data or when you use pointers and INFORMATS, the same problem can occur. The PAD option is the best choice for resolving this problem. This option says to PAD out short data lines with blanks to the length of the logical record (set by default or by the LRECL option).

```
***INFILE OPTIONS: PAD;
DATA INFORM;
   INFILE DATALINES PAD;
   INPUT  X    1-2
          Y      3
          A $ 4-10
          Z     11;
DATALINES;
 12HELLO  3
4 5GOODBYE
 78LAST   9
;
PROC PRINT DATA=INFORM;
   TITLE 'USING THE PAD OPTION';
RUN;
```

Reading Data "Blindly"

Suppose you have a data file and are not given information on the data layout. Are you dead in the water, finished, kaput? Not at all. Use the strange looking program below to first take a look at the file. Notice that there are no variables listed on the INPUT statement. This form of the INPUT statement will read an entire logical record (or line of data) but will not assign any values to variables. This does not seem very useful except for the fact that you can use a PUT statement with the keyword _INFILE_ to write out the contents of the data record to the log (the default output location) or to your output window (by using a FILE PRINT statement). Look at the program below and the resulting log file. (Note: this form of INPUT statement is obviously useful when you are reading data from and external file. The instream data in this example is for illustrative purposes only.)

```
***NO VARIABLES LISTED;
DATA READIT;
   INFILE DATALINES;
   INPUT;
   PUT _INFILE_;
DATALINES;
12345ABCDEXYZ
1122334455667
 12HELLO  310/21/1946
4 5GOODBYE611/12/1997
;
```

Here is a listing of the SAS log after running the program above:

```
44   DATA READIT;
45      INFILE DATALINES;
46      INPUT;
47      PUT _INFILE_;
48   DATALINES;

12345ABCDEXYZ

1122334455667

 12HELLO  310/21/1946

4 5GOODBYE611/12/1997

NOTE: THE DATA SET WORK.READIT HAS 4
OBSERVATIONS AND 0 VARIABLES.
NOTE: THE DATA STATEMENT USED 0.17 SECONDS.
```

Using More than One INPUT Statement: The Single Trailing @

There are times when you want to be able to read one or more values from a data line and then, depending on the value, decide how to read the remaining values. In the example that follows, there are two data layouts in a single file. Type 1 records have age in columns 1 and 2; type 2 records have age in columns 3 and 4. The record type is stored in column 6. You want to be able to read the record type first and then decide where to read the age value. If you use two INPUT statements like this:

```
INPUT @6 TYPE $1.;
IF TYPE = '1' THEN INPUT AGE 1-2;
ELSE IF TYPE = '2' THEN INPUT AGE 3-
4;
```

it will not work. After the first INPUT statement is executed, the pointer moves automatically to the next line. Age values are then read from the wrong line. You want to tell the program to "hold the line" after reading the value for TYPE. You do this with a single trailing @ as shown next:

```
***WHERE IT'S @;
DATA TRAILING;
   INPUT @6 TYPE $1. @;
   IF TYPE = '1' THEN INPUT AGE 1-2;
   ELSE IF TYPE = '2' THEN
      INPUT AGE 3-4;
   DROP TYPE;
DATALINES;
23   1
  44 2
;
PROC PRINT DATA=TRAILING;
   TITLE 'SINGLE TRAILING @';
RUN;
```

After a value is read for TYPE, the trailing single @ tells the program not to go to the next data line for the next INPUT statement in the data step. When the data step finishes, the pointer will then move to the next line of data for another iteration of the data step.

A very useful application of the single trailing @ is to decide whether to read a line of data or not. For example, suppose you only want to read data on females from a raw data file. One way to do this is to read a line of data in the usual way and to delete all observations where the value of GENDER is not equal to 'F'. A much more efficient way is to first read the value of GENDER and only read the remaining values if GENDER is female. Here is how it's done.

```
***ANOTHER @ EXAMPLE;
DATA TRAIL2;
   INPUT @1 GENDER $1. @;
   IF GENDER NE 'F' THEN DELETE;
      INPUT @3 AGE 2.
            @5 HEIGHT 2.;
DATALINES;
M 2368
F 4462
;
PROC PRINT DATA=TRAIL2;
   TITLE 'ANOTHER @ EXAMPLE';
RUN;
```

If there are a lot of variables in each observation this method can save considerable computer time. Remember that when the DELETE statement is executed, control returns to the top of the DATA step.

Creating Several Observations from One Line of Data: The Double Trailing @

What if you want to create several observations from a single line of data? In the example below, several X,Y pairs are placed on a single line to save space. A single trailing @ would not help here. After a value was read for X and Y, the data step would iterate again and the pointer would move to a new line. Try it. You will have only two observations, the first with X=1, Y=2 and the second with X=7, Y=8. To hold the line for multiple iterations of the data step, use the double trailing @ as shown in the example below:

```
***WHERE IT'S REALLY @@;
DATA DOUBLE;
   INPUT X Y @@;
DATALINES;
1 2 3 4 5 6
7 8
```

```
;
PROC PRINT DATA=DOUBLE;
   TITLE 'DOUBLE TRAILING @';
RUN;
```

Reading Multiple Lines of Data for a Single Observation

What do you do when you have more than one line of data for each observation you want to create? You use a line pointer #. In the example below, there are two lines of data for each observation. The line pointer, #, is used to tell the program that ID and DOB are on the first line and HEIGHT and WEIGHT are on the second line.

```
***READING MULTIPLE LINES FOR ONE
OBSERVATION;
DATA MULT1;
   INPUT #1  @1 ID $11.
             @13 DOB MMDDYY8.
         #2  @5 HEIGHT 2.
             @8 WEIGHT 3.;
   FORMAT DOB MMDDYY10.;
DATALINES;
123-45-6789 10211946
    68 158
253-65-5455 11111960
    62 102
;
PROC PRINT DATA=MULT1;
   TITLE 'READING MULTIPLE LINES';
RUN;
```

If you only want to read the first two lines of data but you have more than two lines of data for each subject, make sure you include a #n at the end of your INPUT statement, where n is the number of data lines for each subject (it **must** be the same for each subject). The program below demonstrates this with 4 lines of data for each subject with values being read from only the first 2.

```
***READING MULTIPLE LINES FOR ONE
OBSERVATION;
DATA MULT2;
   INPUT #1  @1 ID $11.
             @13 DOB MMDDYY8.
         #2  @5 HEIGHT 2.
             @8 WEIGHT 3.
         #4;
   FORMAT DOB MMDDYY10.;
DATALINES;
123-45-6789 10211946
    68 158
9879876987698769876987
0987098709870987098709
253-65-5455 11111960
    62 102
9876987698769876987698
0987098709870987098709
;
PROC PRINT DATA=MULT2;
   TITLE 'READING MULTIPLE LINES';
RUN;
```

Suppressing Error Messages in the SAS LOG

For our last two examples, we will show you how to eliminate error messages from being written to the SAS log when bad data values, such as character data in a numeric field, are encountered. In the example that follows, values of 'NA' and '?' indicate that a value was missing. A normal INPUT statement generates NOTES and listings of the offending data lines in the SAS log. The following program generates such error messages:

```
***SUPRESSING ERROR MESSAGES;
DATA ERROR;
   INPUT X Y Z;
DATALINES;
1 NA 3
4 5 ?
;
PROC PRINT DATA=ERROR;
   TITLE 'SUPRESSING ERROR MESSAGES';
RUN;
```

Here is a listing of the SAS log from the program above:

```
56   ***Suppressing Error Messages;
57   DATA ERROR;
58      INPUT X Y Z;
59   DATALINES;

NOTE: Invalid data for Y in line 60 3-4.
RULE:----+----1----+----2----+----3----+----4--
60   1 NA 3
X=1 Y=. Z=3 _ERROR_=1 _N_=1
NOTE: Invalid data for Z in line 61 5-5.
61   4 5 ?
X=4 Y=5 Z=. _ERROR_=1 _N_=2
```

```
NOTE: The data set WORK.ERROR has 2 observations
and 3 variables.
NOTE: The DATA statement used 0.22 seconds.
```

If you know the there are invalid data values, such as NA (not applicable) in a numeric field and you want to avoid all the NOTES in the SAS log, use a ? modifier in your INPUT statement. A single ? following the variable name tells the program to omit the NOTES from the log. Look at the program below and the resulting log.

```
***SUPRESSING ERROR MESSAGES;
DATA NOERROR1;
   INPUT X ? Y ? Z ?;
DATALINES;
1 NA 3
4 5 ?
;
PROC PRINT DATA=NOERROR1;
   TITLE 'SUPRESSING ERROR MESSAGES';
RUN;
```

```
67   ***Suppressing Error Messages;
68   DATA NOERROR1;
69      INPUT X ? Y ? Z ?;
70   DATALINES;

RULE:----+----1----+----2----+----3----+----4---
71   1 NA 3
X=1 Y=. Z=3 _ERROR_=1 _N_=1
72   4 5 ?
X=4 Y=5 Z=. _ERROR_=1 _N_=2
NOTE: The data set WORK.NOERROR1 has 2
observations and 3 variables.
NOTE: The DATA statement used 0.16 seconds.
```

Finally, to completely eliminate both the NOTES and the data listing, use a double ?? following the variable name like this:

```
***SUPRESSING ERROR MESSAGES;
DATA NOERROR2;
   INPUT X ?? Y ?? Z ??;
DATALINES;
1 NA 3
4 5 ?
;
PROC PRINT DATA=NOERROR2;
   TITLE 'SUPRESSING ERROR MESSAGES';
RUN;
```

Notice that the SAS log (following) indicates no errors. Use this with caution! Make sure you understand your data before overriding error messages.

```
78   ***Suppressing Error Messages;
79   DATA NOERROR2;
80      INPUT X ?? Y ?? Z ??;
81   DATALINES;

NOTE: The data set WORK.NOERROR2 has 2
observations and 3 variables.
NOTE: The DATA statement used 0.11 seconds.
```

One final example shows how a double ?? can allow you to read a date field with invalid dates such as 99/99/99, or MISSING, in place of a valid date.

```
***SUPRESSING ERROR MESSAGES;
DATA NOERROR3;
   INPUT  @1 DATE ?? MMDDYY8. @10 X;
   FORMAT DATE MMDDYY8.;
DATALINES;
10/21/46 3
MISSING  4
99/99/99 5
;
PROC PRINT DATA=NOERROR3;
   TITLE 'SUPRESSING ERROR MESSAGES';
RUN;
```

Conclusion

We have demonstrated some of the power and versatility of the seemingly simple INPUT statement. Remember that the INPUT statement is one of the strengths of the SAS language that allows you to read such diverse types of data easily.

Ronald P. Cody, Ed.D.
Robert Wood Johnson Medical School
Department of Environmental and Community Medicine
675 Hoes Lane
Piscataway, NJ 08822
(732)235-4490

cody@umdnj.edu

Manipulating Data:
Elements of the DATA Step Language

Paul M. Dorfman
CitiCorp AT&T Universal Card
Jacksonville, FL

ABSTRACT

In the SAS® DATA step, you can manipulate data by instructing SAS what to do. However, SAS understands only instructions given in its own tongue. The DATA step language (SAS DSL) is simple and powerful, its syntax is crisp and highly readable, and it is easy to learn. In this presentation, we will try to introduce the basics of speaking SAS. Just like in any language, a valid SAS phrase contains meaningful expressions that follow an intelligible sequence. These two principal parts will be considered one at a time:

1. Expressions - basic blocks from which statements are built: Constants - Variables- Arrays- Assignment and Sum- Operators- Functions.
2. Control Flow - the order of execution of instructions: Conditional execution-Branching-Repetition. The presentation should provide an idea about the overall structure and main building blocks of the language used in the SAS DATA step. Some aspects (functions, specific statements) will be touched just briefly as part of the big picture and covered in detail in other Intro to SAS presentations.

I. INTRODUCTION

In a software package as diverse as the SAS System, it would seem to be difficult to pinpoint the most important part. Yet in reality, it is surprisingly easy: The single most powerful element of the entire system is the SAS DATA step. The reason is simple: While SAS procedures and other pre-programmed SAS modules can be applied to solve a variety of particular problems incredibly well, none of them can do it all. Because the language of the SAS DATA step (from now on referred in this paper as SAS DSL: SAS DATA step Language) is an entire language in its own right, you could use it, in principle, to implement just about any programming algorithm of your own (whether it actually makes sense in a particular situation is a different story!). Thus in the Base SAS, the DATA step is the place where programming in its strict sense mainly occurs.

Becoming a professional SAS DSL speaker (or writer if you will) comprises two complementary components that should be embraced simultaneously: Applying it as quickly and as extensively as possible to solve a variety of practical tasks and learning SAS DSL grammar and usage. Restricting the learning process to the former will most probably result in a large amount of distasteful and inefficient code, while the reverse will make one a chef who knows all the recipes but has never actually cooked. Luckily, which part to start with is not a chicken-or-egg question, for it is impossible to speak a foreign language without learning a word. Likewise, you cannot dive head first into the SAS DSL programming without having digested at least a marginal amount of its vocabulary and grammar beforehand.

This is what we are going to concentrate on in this tutorial: Learning the basics of SAS DSL grammar and usage.

II. FACTS

1. WHAT KIND OF LANGUAGE IS SAS DSL?

Historically, SAS DSL has its roots buried in PL/I and resembles its syntax very closely. It seems all the more logical that initially, the SAS System was mainly written in PL/I as its underlying software. Whether by chance or on purpose, this choice has proven to be extremely successful. In many (not so humble) opinions, PL/I is the best and most powerful third generation language (3GL) ever created, and of all languages, its syntax, and therefore that of SAS DSL, is most wonderfully balanced. It is crisp and clear. It is English-like without the viscous wordiness of COBOL and technically concise without being cryptic like C.

From the standpoint of the language generation, it is hard to classify SAS DSL in any clear-cut way. It definitely possesses practically all the features of a 3GL, its level being sufficiently low for implementing just about any common algorithm - although not as low as that of PL/I or C. On the other hand, it is laden with 4GL features such as automatic control flow, list processing, and Output Delivery System.

Besides, SAS DSL comes equipped with an incredible number of ready-to-go programs on call, such as functions, call routines, formats, and informats. They make it possible to achieve, in a few keystrokes and with the utmost confidence in the final correct result, many programming goals that in other 3GLs have to be attained through tedious coding, whose actual quality, being dependent on the local programming expertise, very often goes very wrong. Thus, the variety of powerful and versatile programs on call is, in a sense, a 4GL feature as well.

2. WHAT IS THE SAS DATA STEP?

From the standpoint of a typical multi-step SAS program, the DATA step, along with SAS procedures, is one of separately compiled, top-down executed programs. From the standpoint of general programming, the DATA step is a complete, stand-alone computer program. As such, it has its memory managed separately from any other step, and, all by itself, the DATA step constitutes a complete programming environment equivalent to that of a whole language. For example, any COBOL program can be replicated in a single DATA step (only it is likely to have 1/5 the lines of code, take 1/5 the time to write, and work right the first time).

3. COMPILATION AND EXECUTION

Just as any other general computer program, the DATA step has two distinctive phases:

1. Compilation.
2. Run.

The compilation phase is necessary to do the following:

1. Check the program syntax and translate SAS DSL instructions into machine language (the one the computer understands).
2. Acquire enough computer memory necessary to run the step and organize it appropriately.

3. Perform actions and execute the instructions that should be carried out before the step has begun to run, for example, initialize selected variables and/or arrays.

The only difference between this process and a similar process used in other languages is that the object of compilation (executable file or load module) cannot be saved and subsequently run without the source code and SAS being active on the machine. This means (apart from the special way the SAS software is made available for use) that each time a DATA step is submitted, it has to be recompiled, and that once the source code is lost, the program cannot be run. However, neither could be considered a disadvantage. SAS DATA step compiler is extremely fast, and the compilation time is usually infinitesimally short, even for large DSL programs. As to the ability to run a program without the source code, it is considered insane even in the shops using languages that do provide for it, because it makes it impossible to change, update, correct – in other words, maintain the program.

4. THE SHELL

Any DATA step begins with the keyword DATA and ends when it has encountered the keyword RUN. It will also stop if it sees another keyword DATA or keyword PROC, or if it is the end of the entire batch SAS program, because in this case the SAS Supervisor (the internal program SAS uses to coordinate all steps) understands that either the next step is about to begin, or the current step is final.

There are plenty of good reasons to always finish a step up with the RUN keyword. For instance, if you are running SAS interactively, and the last step in the program is not closed with RUN, you will be waiting forever before seeing a result, even though there is a message displayed telling that the step is running. There also exist situations when establishing an explicit step boundary with RUN is required for other things to happen after the step has executed. Here it is assumed that it is always the case, and each DATA step has RUN as its last statement. In other words, we will view the space between the DATA and RUN keywords as a *shell*, within which a DSL program is written.

Hence, in principle, any conceivable DATA step program looks like this:

```
Data < output SAS data set list > ;
  …
  < SAS DSL program instructions >
  …
Run ;
```

In the simplest case, when no output SAS data sets are to be created, it is indicated by the keyword _NULL_. The simplest DATA step program containing no instructions at all is thus:

```
Data _Null_ ;
Run ;
```

III. THE COMPONENTS

1. AB OVO

The usefulness of the program above is somewhat limited by the fact that at run time, it does nothing, for there are no instructions to execute. However, for our current purposes, it is not completely useless, because there are a number of things one can ascertain from it.

First of all, there are two distinct *language statements* starting with DATA and RUN.

Secondly, each statement is separated from the subsequent statements with a semicolon. And, in fact, one of the fundamental SAS syntax rules mandates, that, yes,

- Each DSL statement must end with a semicolon.

Thirdly, there are certain words in DSL that mean something special to the DATA step compiler; that is, unlike other words, they tell the compiler to do concrete things. Such words are called keywords. SAS relies on their presence, sequence, and place in the program to figure out what it is that you are trying to tell the software to do.

Finally, if we change the entire program to the upper case, or reverse the case of each letter, it will still run the same way. To SAS, the words case, Case, and CASE all mean the same. In other words:

- You can write a DATA step program in any case you want - SAS could not care less.

Now let us write a more involved (and useful) program that will let us notice, examine, and learn more interesting things. Suppose you want to submit a SAS program at any time during the working hours, and you want it to tell you, depending on the current time, what action to take and how much money you have earned so far, given an hourly rate. Here it is:

```
Data _Null_ ;
  Retain Hour_Rate 100.00
     Start_Time '09:00't ; Length Message $ 15 ;

  Hours_Worked = ( Time() - Start_Time ) / 3600 ;
  Money_Earned = Round (Hours_Worked*Hour_Rate, 0.01) ;

  If      Hours_Worked => 8 Then Message = 'Go Home!'  ;
   Else If Hours_Worked => 4 Then Message = 'Lunch!'    ;
      Else                        Message = 'Keep Coding!' ;

       Put Message Money_Earned = Dollar7.2 ;
Run ;
```

This, still a very basic program, already incorporates almost all components shared with its more complex siblings.

2. FREE-FORM

First, let us notice two quite convenient features of SAS DSL:

- There is no particular column in the code where a statement must begin.
- A statement can be split between two (or more) lines of code.
- More than one statement can be located on a single line, as a semicolon delimits the statements from each other.

So, even though it is a *good programming practice to have one statement (or several short statements with identical meaning) per line*, this discretion belongs to the programmer, not the compiler. To the latter, a statement is a group of symbols between two semicolons; the rest is up to the programmer. All this can be summed up as follows:

- SAS DSL is a free-form language.

Being free form is an extremely important and convenient feature of the language. It lets *you* decide what style of coding – spacing, indentation, blocking – you prefer, and it does not force you to start certain statements in certain columns. Just ask any SAS or PL/I programmer who has also programmed in COBOL.

3. TOKENS

Looking at the above program, a naked human eye will immediately distinguish a variety of different things seeming to possess an intrinsic meaning. For example, in the group

Hours_Worked*Hour_Rate,

Hours_Worked, the asterisk, and Hour_Rate all appear to mean something specific, although there are no blanks between them to tell them apart. In part, the SAS compiler thinks the same way. When it looks at the program, it separates the asterisk, based on its special meaning, from the rest of the words, and interprets them based on the syntax rules and context.

Such smallest program unit, which SAS compiler perceives as having an intrinsic meaning, is called a *token*. There are only three types of tokens in SAS DSL:

- Words
- Symbols
- Constants

Words differ in nature, depending on whether they are *keywords* or *names*. Names and symbols combine in *expressions*. Keywords and expressions make up *statements*. Let us consider the SAS DSL elements one at a time.

4. WORDS

As noted above, all words in a DATA step program fall into two distinct categories:

- Keywords
- Names

5. KEYWORDS

Keywords are words that have a special predefined meaning to the SAS compiler – and hence in the SAS syntax. Most often a keyword is used to begin a SAS statement, but they also occur at particular locations inside statements. In the sample program above, the keywords are DATA, RETAIN, LENGTH, ELSE, PUT, RUN (at the beginning of statements), IF (at the beginning and inside statements), and _NULL_ (inside the DATA statement). There are dozens of other keywords in SAS DSL – listing all of them here would mean retyping a good chunk of the SAS Language reference, and it is not the goal here. However, we will meet more keywords while discussing other language elements and sequence control.

6. NAMES

A name in a SAS DATA step program is a word the programmer comes up with to identify an object intended to be utilized or manipulated by the program. Looking a bit forward, such objects, for instance, may include variables, SAS data sets, statement labels, and arrays.

This definition does not imply, however, that any collection of symbols of any length can be used for a name: The name must *at least* be a *valid SAS name*, plus other limitations may apply, depending on the object type.

What is a valid SAS name? It is an identifier that:

1. Is no longer than 32 characters.
2. Begins with a letter or underscore.
3. For other characters, has letters (A-Z, any case), digits (0-9), or underscores.

For example, the names used in the sample program and shown below on the left are valid; a slight alteration in the direction of not complying with the rules above (the column on the right below) makes them invalid:

Valid Names	Invalid Names
Hour_Rate	Hour Rate
Start_Time	Start-Time
Message	1Message
Hours_Worked	#HoursWorked
Money_Earned	Money$Earned

Note that the case of the identifiers is irrelevant: They can be all in upper case, lower case, title case, sentence case, or mixed.

As mentioned, there are objects whose naming conventions are even more restrictive than a valid SAS name. Such is the case with formats and informats allowing only 6 to 7 characters in their names. File references (filerefs), i.e. names used to point to external files, mainly have to conform to the naming conventions of the operating system used. Thus, under OS/390 (or z/900, for that matter), they are limited to 8 characters, allowing, on the other hand, the dollar and pound signs in the names of filerefs.

At the first glance, the rules above seem to impose a lot of restrictions on SAS names. However, it is a wrong impression. They are no more restrictive than names in other languages, yet enjoy a tremendous privilege: SAS DSL has virtually no *reserved words*. That is, the SAS compiler is smart enough to understand the phrase like

IF THEN = 10 THEN IF = THEN

unambiguously from the context, by identifying the first IF and second THEN as a keyword, and giving the first and third THEN the meaning of a variable. Note that even though *it is not recommended* to code this way, the feature completely rids the SAS programmer of the pesky burden imposed by hundreds of reserved words in some other languages (notably COBOL), where using a reserved word for a user-defined name means a failure to compile. Once again, SAS shifts the responsibility of maintaining the correct syntax from the programmer to the compiler. Let us summarize it as another SAS DSL advantage:

- SAS DSL has virtually NO reserved words to remember and/or to avoid.

6.1. SPECIAL NAMES

The above notion is all the more amazing in the light of the fact that there are also dozens of *special names* in DSL that SAS uses for its internal purposes. Most of them begin and end with an underscore, but only few of them are genuinely reserved, that is, a programmer cannot use them arbitrarily as user-defined names. However, even in these cases, there is usually a highly logical reason for having it this way. For example, such reserved words as _ALL_ (points to all variables currently in memory), _ NUMERIC _ (the same for all numeric variables) cannot be used as a variable name - but it is illogical to use a name for a single variable and a collection of variables at the same time, hence such usage is banned. On the other hand, such reserved names as _N_, _IORC_, _I_, and numerous others can be actually used by a programmer for uses other than intended (*sometimes*, it makes sense) with no harmful consequences whatsoever.

7. EXPRESSIONS

An alert reader has no doubt noticed that symbols and constants have been left out of discussion so far, even though the logic would certainly dictate it. This is because it appears even more logical to discuss them while considering SAS DSL expressions – the largest meaningful blocks of which SAS statements are composed.

A natural language sentence consists of one or more expressions. A natural expression is, loosely speaking, a group of words and punctuation symbols, collectively bearing a concrete meaning. The same is basically true about a SAS DSL statement. Defined in a more formal manner, an expression is a group of words and symbols resolving to a concrete value.

Let us identify a couple of expressions in the sample code above:

```
( Time() - Start_Time ) / 3600
Round (Hours_Worked*Hour_Rate, 0.01)
```

Actually, each of these expressions are *composite expressions*, because they are in turn composed of expressions conjugated by SAS operators - in the case above, arithmetic operators (-, /, and *). So, the question is, what kind of tokens can a *simple expression* contain? A simple expression can only incorporate:

- Constants
- Variables
- Array references
- Function calls

Any expression, by its definition, resolves to a value. But in SAS, all values are of two data types only: Numeric and character. Accordingly, there are only two types of SAS expressions:

- Numeric, i.e. resolving to a number.
- Character, i.e. resolving to a character string.

For instance, both expressions given as an example above are numeric. However, in the expression

'Character ' || 'expression' ,

the concatenation operator || combines two character constants producing a (longer) character value; hence, this is a character expression. Now let us look at the components making up expressions separately.

7.1. CONSTANTS

A constant is a value in a DATA step program that the program cannot change. Once again, as it is the case with all SAS values, there are two major types of constants: Numeric and character. In the sample program above, all of the

100.00, '09:00'T, 15, 3600, 0.01

are numeric constants, while all of the

'Go Home!'
'Lunch!'
'Keep Coding!'

represent character constants. In turn, each constant type has several subcategories. First, let us consider numeric constants.

7.1.1 Numeric Constants

When you code something like 0.035, or '05AUG2001'D, or even 1.34E-15, its meaning is obvious to you, but the compiler first must convert each of these things into a form suitable for SAS internal consumption. SAS interprets them using special subroutines called informats. At compilation time, the informats take numeric constants coded in the program and convert them into 8-byte floating point numbers. Numeric constants can be one of the following:

1. Decimal/scientific notation. This is a conventional way you would write a number using a decimal point and a leading sign if necessary, with no commas separating orders of magnitude. Or, if a number is quite large, you may choose to write 1230000000, say, as 1.23E+10 (plus sign is not mandatory). E.g., 1.234, 2.34, -3.1416, 0.001956, 1.6E-19, 4.8E+13 are all valid decimal/scientific notation constants. SAS interprets their meaning using a standard numeric informat W.D.
2. SAS date. Date constants are written as '05aug2001'D or '17jan97'D. SAS uses DATE9. and DATE7. informats to convert date constants into the number of days since the beginning of 1960, and then stores the number internally, just as it would store any numeric value.
3. SAS time. When you write '23:10:44.123'T, you are telling SAS that it is 23 hours, 10 minutes, 44 seconds, and 123 milliseconds since the beginning of the day. However, SAS must understand it only as the number of seconds since the beginning of the day – and hence performs the necessary transformation using its TIME. informat before the constant can be used in the running program. The number of seconds thus obtained is stored internally as a numeric variable.
4. SAS datetime. This is written as, for example, '31aug2000 18:27'DT, and its meaning is obvious for a human. SAS makes it obvious to the computer by turning it into the number of seconds from the beginning of 1960 using the informat DATETIME15. Then it stores the number internally.
5. Hexadecimal. If you have a constant number in the hexadecimal notation and need to use it in a program, SAS will use its HEX. informat to interpret it for you (so you do not have to convert it to a decimal or binary base yourself) and store it appropriately. Just append an X to the end of the number; and if it starts with a letter (letters from A through F used as digits from ten to fifteen in hex), precede it with a 0 – otherwise the compiler will be confused thinking that it is a valid SAS name. Thus, 5A7F8x is a valid hex constant, but the hex number DEC45x is not. To make it valid, write it as 0DEC45x.

SAS DSL approaches the issue of interpreting constants quite efficiently, by converting them to their proper internal representations at compile time. It makes it unnecessary to do the transformations at run time – possibly many times if, as it often happens, a statement containing a constant is executed repeatedly.

7.1.2. Character Constants

There are only two kinds of character constants:

1. Character literal. It is the most usual character constant written as a character string enclosed in single or double quotes. 'I am hungry…', '12345', 'SAS Version 9' are examples of character literals, called literal perhaps because they are *literally* the values they represent – not just values whose actual meaning to the computer must be further interpreted.
2. Character hexadecimal. Each character in a character string has a predefined number (rank) from 0 to 255 in a so-called collating sequence. Any number up to 255 can be written as a 2-position hexadecimal number. The lowest character is thus ranked 0, and it is 00 in hex. The highest character is ranked 255, which is FF in hex (16*15+15). So, each character can be written as a 2-position hex number. A character hex constant thus always has an even number of hex digits enclosed in quotes and is marked with the letter X at the end. For example, '000000'x, 'FFFFFFFF'x, '40'x are valid character hex constants. Most often, such constants are used to indicate unprintable characters that, by their nature, cannot be typed in the program explicitly.

7.2. VARIABLES

To manipulate data, the programmer needs a data object whose memory contents can be changed, altered, replaced, copied, or erased by a program. This all-important purpose is served by SAS *variables*.

What is a variable? The loose definition just given is actually not all that bad. But let us take a little bit more mechanical approach and examine the first three statements after the DATA statement in the sample program:

```
Retain Hour_Rate 100.00 Start_Time '09:00't ;
Length Message $ 15 ;
Hours_Worked = ( Time() - Start_Time ) / 3600 ;
```

7.2.1. Variable Mechanics

The SAS compiler parses the program in a strictly top-down manner. From the context of the first statement, it understands that RETAIN is a keyword, and 100.00 and '09:00'T are constants. Hour_Rate is the first valid SAS name in a right place to be interpreted as a variable. It is the first variable in the entire program; it has not been referred before; and it is being asked that it should be assigned a numeric value. To SAS, that means an order to organize an 8-byte cell for a numeric variable in the area of memory where retained numeric variables should dwell. Boxes in memory have no names, but only numbers called addresses. Thus SAS must prepare a table in memory where the number of the cell allocated for Hour_Rate will be associated with this name. Being logical, SAS stores some other useful information in the table together with the name and address – in this case, that the variable is numeric, and hence it is 8 bytes long. The table (called *symbol table*) is organized in such a way that given a variable name, all associated information can be retrieved almost instantly. Having been done with all that important work, SAS can now convert 100.00 to its 8-byte internal number and move it into the cell in memory it has prepared for it.

Essentially the same process will be repeated with Start_Date; only because it is not named in any RETAIN statement, nor is it retained by default, SAS allocates a cell for it at the beginning of a different area in memory, where all non-retained numeric variables belong. Yet another memory area will be used to store the variable Message. Upon seeing that in the LENGTH statement, Message is declared as character (because of the dollar sign preceding it), SAS allocates a 15-byte long cell in the domain where non-retained character variables will dwell. At this time, SAS does not yet know what to move there (the value is *missing*), so it uses blanks – standing as missing values for character variables.

A similar situation occurs with Hours_Worked. SAS knows that it is numeric because there is a numeric expression waiting to be resolved and assigned to it, but it can only happen at the run time – the TIME() function returns the time at the moment when the instruction is being executed. Therefore, so far, at compile time, the value for Hours_Worked is missing. This variable is numeric, so the corresponding missing value should be numeric. For this purpose, SAS uses one of the special NAN (not a number) binary values called *standard numeric missing value* and moves it into the cell (now residing in the non-retained numeric area of memory) prepared for Hours_Worked. Using its standard numeric format, SAS *prints* it as a period (.). This very period should be used as a constant if you want to explicitly initialize a variable to a standard missing value during compile time.

7.2.1. Automatic Variables

You may not have defined any variables in your DATA step program, yet there are always at least three automatic variables in the step DSL allocates no matter what: _N_, _ERROR_, and _IORC_. At compile time, _N_ and _IORC_ are set to 1, and _ERROR_ is set to 0. At run time, _ERROR_ is changed to 1 if either a run-time error condition occurs, or an I/O operation sets _IORC_ to a value other than 0. _N_ is assigned the number of times (accumulated independently in some internal variable) program control is transferred to the instruction immediately following the DATA statement.

Also, the DATA step compiler creates other automatic variables, such as _I_ and _IORC_, and many others when it detects certain statements in the step.

Automatic variables reside in areas of memory from where nothing is written to any output SAS data set - which is another way to say that they are *automatically dropped*.

7.3. ARRAYS

In most cases, variables in a program are called by name when they are incorporated in an instruction prescribing what to do with their values. The same most often happens in life with things less abstract than SAS variables, for example, with months, when we mark dates in a calendar. However, one may find it more convenient to call the months by their ordinal number instead of by name. SAS arrays provide an opportunity to do exactly that with SAS variables in a program.
For example, if we have seven variables SUN, MON, TUE, WED, THU, FRI, SAT, and if we need to do something with them in a program, we could call them only by their names. However, if we add one of the statements
ARRAY Day (*) SUN, MON, TUE, WED, THU, FRI, SAT ;
ARRAY Day (7) SUN, MON, TUE, WED, THU, FRI, SAT ;

before any of the variables above have been referenced, we will be able to refer to them *both by their native name and ordinal number*. To SAS, SUN and Day(1), TUE and Day(3), and so on, will mean all the same. The number inside the parentheses is called *array index or subscript*. It does not have to be a constant; instead, it can be any numeric expression resolving to a number between 1 and 7. Note that the array bounds are set at compile time, and they must be integer constants (or macro variables resolving to such) – the compiler does not tolerate even a decimal point.

Array references can be used in expressions just as well as variables. Their advantage over non-arrayed variables is that references to arrayed variables can be made dynamic by changing the value of the index. For instance, if at run time, you need to move the value of 123 to all seven variables, you can set X to 1, and then instruct SAS to repeat the instruction Day(X)=123 seven times, each time changing the value of X up by a unity.

Arrays are data structures so powerful and with so many more features and uses in SAS DSL that this little array subsection could be easily expanded into a thick tome. Here, we have just enough room to briefly sketch where arrays belong in the DATA step.

7.4. FUNCTIONS

A function is an encapsulated stand-alone program that return a value given (an)other value(s) called argument(s). In real-word SAS programming, it is quite difficult to find an expression without a function call. Even in the program as basic as the one being used as a sample, TIME() and ROUND() are both function calls.

There are no user-defined functions in SAS DSL (which is certainly a disadvantage compared to some other tongues, notably PL/I or C). However, this is to a large degree compensated by the fact that SAS provides hundreds of intrinsic (i.e. coming with the language, pre-programmed, guaranteed-to-work-right) functions, from finding a character in a string, to evaluating the distance between words, to computing almost anything that could be thought of being necessary to calculate in business and statistics.

7.5. OPERATORS

To make a new expression out of several existing expressions in a natural language, people use prepositions. In programming in general and SAS DSL in particular, this role is played by the tokens called *operators*. Expressions they conjugate are called *operands*. In the sample program, for example, the function Time(), variable Start_Time, and constant 3600 are all elementary expressions (consisting of a single element). The subtraction operator creates the expression Time() - Start_Time, and then the division operator uses it and 3600 as operands to create the expression (Time()-Start_Time) / 3600. Finally, the assignment operator (=) tells SAS to take the value, to which the expression on the right resolves, empty the memory cell belonging to the variable Hours_Worked, and fill it with the resolved value instead.

Operators discussed above are represented by symbols, but it does not have to be the case. They can also be keywords. For instance, MAX, NOTIN, AND, OR operators are keywords. However, they, as well as many others, can also be written as symbols. Thanks to the wisdom of DSL developers, the symbols and keywords used for SAS operators make their purpose pretty much self-explanatory. The way arithmetic operators are used in formulae, such as shown above, conforms to the standard algebraic rules, as well as the priority of their evaluation. If there is any question about the default priority of evaluating an expression, it can be always enclosed in parentheses, which will force SAS to evaluate it first.

7.6. ASSIGNMENT

Assignment statement is used to replace the value of a variable in its memory cell. The assignment operator looks like an equal sign, but by no means is it an equivalent of an ordinary algebraic assignment. This is compounded by the fact that in SAS, an equal sign may be also used as a comparison operator, also written as EQ. The general assignment form is as follows:

< Variable > = < Expression > ;

What occurs here is basically very simple:

1. The expression on the right is evaluated.
2. The resulting value is stored in some intermediate memory location.
3. The content of the variable cell is erased.
4. The new value is moved to the variable cell.

For example, in the light of this process, let us dissect an assignment operation

X = X * Y ,

Algebraically, it is meaningless. However, it is perfectly meaningful programmatically: It retrieves the value of X from the X-cell and multiplies it by the value extracted from the Y-cell; then the result is used to replace whatever value originally resided in the X-cell (and was multiplied by Y) before. Any confusion can be avoided if the above 'formula' is thought of as 'Set X to the product of X and Y'.

7.7. SUM STATEMENT

There is another form of assignment statement pertaining to addition only (or subtraction, for that matter):

< Variable > + < Expression > ;

When the compiler sees such a thing, it:

1. Allocates a cell for the variable in the area of memory segregated for retained numeric variables.
2. Moves 0 to the cell.

This is logical because in most cases, the SUM statement is used to accumulate numeric stuff, and such fields are usually zeroed out before the accumulation begins. At run time, the expression on the right of the plus sign is evaluated, the result is added to the value of the variable, and the sum is assigned to the variable.

If instead of adding the value to the variable, the same amount needs to be subtracted, a unary operator can be appended to the left of the operand-expression:

< Variable > + - < Expression > ;

Those who care about the symmetry and would like to make SUM statements stand out in the code more prominently, might want to double the single plus used in the additive statement:

< Variable > ++ < Expression > ;

It has the same effect, as a single plus but is very easy to spot. And, of course, any C/C++ programmer will understand its meaning without saying.

II. CONTROL FLOW

Now that we have seen, albeit briefly, what sort of stuff SAS DSL instructions comprise, it is necessary to get an idea how to control the order in which they are executed at run time. Such order is called *control flow*. The instruction currently being executed is said to have control. After it has been executed, control is transferred in the top-down manner straight to the next instruction, in the absence of special branching instructions altering the top-down sequence control.

1. SELECTION

The sample program we started with contains no such provisions until the IF-THEN-ELSE structure is encountered:

```
If      Hours_Worked => 8 Then Message = 'Go Home!';
Else If Hours_Worked => 4 Then Message = 'Lunch!'  ;
Else                          Message = 'Keep Coding!' ;
```

At this point, depending on the time of the day, one of the three instructions is performed. This is termed *conditional execution*; and such a structure itself is called *selection*. After the selection has been made, the remaining instructions are executed in a straight sequence, control reaches the bottom of the step, and the program stops.

The instruction executed when a selection is made does not have to stand-alone. It can be combined with other instructions executed when the same criterion is met. However, in this case syntax demands that all of them must be enclosed in a so-called DO-END block. For example, if in the case #2, you would like to display the values of all variables at this point in the program, it could be coded as follows:

```
If Hours_Worked => 8 Then Message = 'Go Home!';
Else
If Hours_Worked => 4 Then Do;
       Message = 'Lunch!' ;
       Put _All_ ;
End ;
Else Message = 'Keep Coding!' ;
```

When numerous changes to the program's logic can be anticipated, it makes sense to place even a single conditional instruction within a block, for then it is much easier to insert any number of additional instructions into the same block as needed.

2. PLAIN BRANCHING (GoTo)

The excerpt above could be written in a different style using the GOTO statement. GOTO performs what is known as simple branching:

```
If Hours_Worked => 8 Then GoTo One ;
If Hours_Worked => 4 Then GoTo Two ;
If Hours_Worked => 4 Then GoTo Three ;
One: Message = 'Go Home!'; GoTo Exit ;
Two: Message = 'Lunch'; Put _All_; GoTo Exit ;
Three: Message = 'Keep Coding!' ;
Exit:
```

To mark the point to which a GOTO transfers control, statement labels are used. Above, they are represented by the keywords One, Two, Three followed by the colons. A label must be a valid

SAS name, and of course, labels must be unique. Any statement can be preceded by a label without the harm of affecting control flow. They become operational only if a GOTO transfers control to one of them. Note that the logic of simple branching renders the ELSE keywords unnecessary.

Programming this way instead of using a selection structure is not particularly recommended. Moreover, SAS provides a special SELECT structure specifically designed to program this kind of situation without a single GOTO or IF-THEN-ELSE:

```
Select ;
  When (Hours_Worked => 8) Message = 'Go Home!';
  When (Hours_Worked => 4) Do;
        Message = 'Lunch!' ;
        Put _All_ ;
  End ;
  Otherwise Message = 'Keep Coding!' ;
End;
```

In actuality, GOTOs have advantages, and do no harm when used with discretion and understanding of control flow.

3. REPETITION

In programming, the concept of the repetitive execution is one of the most significant. It is because of the repetitive execution that the computer can be instructed to perform almost the same stuff millions of times, altering its modus operandi with each iteration just as much as prescribed by the programmer – which is exactly the kind of thing computers are needed for in the first place.

The GOTO command is the most elementary instruction that, in a combination with IF, provides for organizing a controlled repetition. In the example of the array Day(*), filling all its elements with 123 could be coded this way:

```
ARRAY Day (7)  SUN, MON, TUE, WED, THU, FRI, SAT ;
X = 1 ;
Assign: Day(X) = 123 ;
       X = X + 1 ;
       If X < 8 then GoTo Assign ;
```

Once control reaches the conditional statement, it evaluates the current value of X. If it is still within the array boundaries, control is transferred back to the ASSIGN label, and the process repeats until X has become equal to 8. After that, control simply goes to the next instruction. Since it will inevitably happen, it prevents the structure from iterating endlessly.

Luckily, SAS DSL provides a separate piece of syntax, the so-called *DO loop structure*, solely devoted to the task of organizing repetitive execution in a flexible and robust manner, thus rendering the IF-GOTO pair almost unnecessary for this purpose (with quite rare exceptions of intricate, high-performance cases). Let us see how it might look like using the array example above:

```
Do X=1 By +1 Until ( X = 7 ) ;
  Day (X) = 123 ;
End;
```

The block of statements between the DO and END statements is called the *body of the loop*. In this case, it consists of a single statement, Day(X)=123. How does SAS know that this syntax instruct it to execute the body repeatedly – as opposed to a mere DO-END block we have seen earlier? The whole trick is in the presence of one of the two keywords – BY and UNTIL – in the head of the loop. Either one can cause the loop to iterate, albeit in different modi operandi.

The first thing that occurs in the loop head is 1 being moved to the variable X. That this *loop counter* is also an array index has no special meaning: To SAS, it is just another numeric variable. The presence of such a *FROM-value* in the head of the DO loop makes it being termed an *iterative DO* (although there is no significance in such terminology: The ability to iterate is the main feature of any loop). It is important that the FROM-value is assigned only once and forever, before the loop begins to iterate. The presence of the UNTIL condition tells SAS to:

1. Execute the body of the loop.
2. *At the bottom of the loop*, increment X by 1.
3. Still at the bottom of the loop, check whether the condition in the parentheses is true.
4. If it is, terminate the loop hand control over to the next instruction.
5. Otherwise, transfer control back to the top of the loop.

Apparently, the logic is precisely the same as that of IF-GOTO, but important advantages lie in not having to organize the loop by hand and eluding the code inquisition looking for GOTOs as if they were witches. Moreover, using a DO loop does not preclude one from exiting it at any point from within the loop by using the same old good GOTO, even though it is almost never warranted.

The UNTIL condition can be any conditional expression, not necessarily an array index comparison. In fact, for the latter, SAS provides a special keyword, TO, where you can specify the upper value of the index to be tested. The loop above could thus be (and usually is) written as

```
Do X=1 By 1 To 7 ;
  Day (X) = 123 ;
End;
```

The TO clause can follow the BY clause and vice versa. Just like with the FROM-value, a *TO-value* is assigned only once, before the loop starts iterating. This means, in part, that one should not be fearful to use expressions as TO and FROM values, for they are evaluated but once.

Obviously, with UNTIL and in the absence of a TO-value, the loop will iterate forever unless the UNTIL condition becomes true at some point in time. Oftentimes, though, it is more convenient to make a loop iterate *while* a condition is still true – which is, quite logically, achieved by coding WHILE instead of UNTIL. In this case, the condition is checked *before the body is executed even once* – so there is a possibility that the loop will never iterate once. Coding our sample loop using DO-WHILE will look like this:

```
Do X=1 By +1 While ( X < 8 ) ;
  Day (X) = 123 ;
End;
```

In the UNTIL loop, when X has become 8 at the bottom, control is immediately moved to the instruction following END. In the WHILE loop, control instead goes to the top of the loop where the condition X < 8 is tested. Since now it is false, the body is skipped, control is transferred all the way past the END, and the loop stops.

3.1. ITERATING AD INFINITUM

So, if a DO-UNTIL loop stops iterating when and if its parenthesized expression evaluates to true (that is, to any number *but* zero or any missing value), a DO-WHILE loop ceases repetition when and if its condition becomes false. Conversely, if the UNTIL condition is always false, or the WHILE condition is always true, the loop will iterate forever. Since 0 is always false, and 1 is always true, an infinite looping may be organized as either of the following:

```
Do Until (0)  ;
  < body of the loop >
End ;

Do While (1) ;
  < body of the loop >
End ;
```

The same effect can be achieved by placing FROM- and BY-values in the loop header all by themselves: The loop will iterate forever, even if the BY-value is 0. This comes in very handy when it is desirous to organize an infinite repetition and initialize a numeric value before the loop starts in the loop head, without a separate statement:

```
Do X=1 By 0 ;
  < body of the loop >
End;
```

However, the presence of a FROM-value without a BY-value will cause the loop to iterate maximum once, no matter what an UNTIL or WHILE clause may contain, if anything (and of course it will not iterate once if the WHILE condition is false a priori).

3.2. GETTING OUT OF A LOOP

What is a practical value of organizing an endless iteration? It adds another layer of flexibility to the repetition structure by making it possible to stop the loop using a condition *in its middle*, rather than on the top or at the bottom already provided by WHILE and UNTIL. SAS provides two principally different ways of branching from inside a loop, regardless of whether it is infinite or not:

1. LEAVE statement.
2. CONTINUE statement.

Suppose that in our sample loop, we want to fill the array with 123 only up to the current weekday. Of course, its number can be put in the UNTIL condition or TO-value. But it can be also be achieved in the following manner:

```
Do X=1 To 7 ;
  Day (X) = 123 ;
  If WeekDay ( date() ) = X then Leave ;
End;
```

The LEAVE statement transfers control immediately past the END statement to the next instruction. It is precisely the same as if we coded:

```
Do X=1 To 7 ;
  Day (X) = 123 ;
  If WeekDay ( Date() ) = X then GoTo Exit ;
End;
Exit :
```

So, in cases when it is easier to organize logic in an infinite loop than in a top- or bottom-terminated loop, or when it is necessary to branch out from the middle, a conditional LEAVE allows doing so without resorting to a GOTO.

The CONTINUE statement also provides for a kind of branching, but instead of transferring control *past the bottom* of the loop (past the END statement), it transfers it *to the bottom* of the loop (just before the END statement) without affecting any things that occur at the bottom of the loop by default (incrementing the loop index, comparing it to the TO-value, or evaluating an UNTIL condition, if any). As an example, suppose that we want to fill out only the array elements corresponding to the even days (in other words, skip the odd ones):

```
Do X=1 To 7 ;
  If Mod (X, 2) > 0 Then Continue ;
  Day (X) = 123 ;
End ;
```

The Mod(X,2) function returns the remainder of X divided by 2, i.e. a number greater than zero if X is odd. If this is the case, the CONTINUE statement causes control to jump right before the END statement, thus skipping the assignment when X is odd. Of course, by logic, it is exactly the same as if we wrote using a label for the END statement:

```
Do X=1 To 7 ;
  If Mod (X, 2) > 0 Then GoTo Bottom ;
  Day (X) = 123 ;
  Bottom :
End ;
```

Of course, above, the goal could be reached without either CONTINUE of GOTO, by imposing an opposite condition on the assignment statement itself, but for illustration purposes, this example is perhaps as good as another.

4. FILE PROCESSING

One of the most compelling reasons why we need iterative structures is the necessity to process files with a great number of records. It can be an external file, a SAS data file, a view to an RDBMS – it does not really matter. What does matter is that in most cases, the file processing means to do the following things:

1. Perform some preparatory work.
2. Read all or many records one by one and apply the same logic to each of them, for instance:
    a. Identify relevant data coming with each record as variables
    b. Manipulate certain quantities according to certain rules
    c. Output information for each or selected records (or none)
3. Maybe output summary information, i.e. pertaining to all records collectively.

4.1 PRACTICAL EXAMPLE OF FILE PROCESSING

As a practical example of applying this principle, let us imagine a SAS data file (set) called STATES having 2 variables: 1) 2-byte character variable STATE containing a state abbreviation code 2) numeric variable WATER representing the annual precipitation in millimeters for a certain year. What we want to do is:

1. Print the time when the step started running in the SAS log.
2. Read the file STATES and calculate the average WATER across all states beginning with the letter A, accounting only for non-missing values of WATER.
3. Write the observations containing the states starting with A and with non-missing WATER values to a SAS data set AWATERNMISS.
4. Print the average amount in the log in whole mm.
5. Print the total processing time in the log in seconds and milliseconds.
6. Stop.

Here is one way to write a SAS DATA step conforming to these specifications:

```
Data AwaterNmiss ( Keep = State Water )  ;
  Time = Time() ;
  Put Time = Time. ;
  Do Until ( End_of_File = 1 ) ;
    Set States End = End_of_File ;
    If State NE: 'A' or Water = . then Continue ;
    NonMiss     ++        1 ;
    TotalWater ++ Water ;
    Output ;
  End ;
  Average = Round ( TotalWater / NonMiss, 1 );
  Put Average = Comma. ;
  Time = Round (Time() – Time, .001 ) ;
  Put Time = ;
  Stop ;
Run ;
```

It deserves some explanation:

1. Since an output SAS data set is needed, it is named on the DATA statement.
2. Function TIME() obtains the current time from the computer clock. It is assigned to the variable TIME.
3. PUT statement prints TIME in the log using TIME. format.
4. DO starts the loop reading the file record after record, but the loop should know where to stop. The last record on the file is marked by the system. To recognize it, SAS provides an END=<variable> option on the SET statement. The variable named in END= (here chosen as End-of-File) has the value 0 for all records but the last. When the last record has been read, the software detects the end-of-file marker and moves 1 to End-of–File. At the bottom of the loop, it makes the UNTIL condition true, and the loop terminates.
5. The SET statement always reads the *next record* from the file (at the beginning of the file, it is the first record), so that in the next iteration of the loop, the next record is read. When the record is read, its values for STATE and WATER are moved into the like named cells in memory the compiler has prepared, thus replacing the previous contents of the cells.
6. NE: operator compares the first byte of STATE with 'A'. If a match is not found, CONTINUE transfers control to the bottom of the loop, and the next iteration begins (unless the end of file has been detected). Otherwise the condition after the OR operator is checked. If true, CONTINUE executes as described. If neither condition is true (STATE does begin with A and WATER is not missing), the next body statement is executed.
7. The Sum statement adds 1 to the variable NONMISS used to count the number of A-states with non-missing WATER, and WATER is added to TOTALWATER used as an accumulator.
8. OUTPUT statement moves the current values of STATE and WATER from their cells in memory to the output record and writes the record (observation) out.
9. After the loop is terminated (all records have been processed), AVERAGE is calculated. The ROUND function rounds the value to the nearest integer, because of 1 as the second argument.
10. AVERAGE is printed in the log using the COMMA. format to show commas separating integer orders of magnitude.
11. The start time determined at the beginning of the step is stored in the memory cell called TIME. The meaning of the statement Time = Round(Time()-Time,.001) is this. Retrieve the value of TIME from its cell; subtract it from the current time; use the result to replace the original value of TIME in its cell. Finally, round the result to the nearest thousandth (because of .001 as the second argument).
12. Print the execution time in the log.
13. STOP. This statement is necessary, otherwise the statements from the top of the step to the SET statement will be re-executed.

4.2. AUTOMATIC CONTROL FLOW

There exists a much trumpeted SAS DSL feature instructors routinely use to underline differences between DSL and other languages: The *automatic control flow*. Contrary to the pretty common faith, it only manifests itself when the compiler detects a file-reading keyword, such as SET, MERGE, UPDATE, or INPUT. In this case, the compiler constructs the loop exactly similar to the DO-UNTIL loop (together with end of file checking) by default, and all programming statements written in the step end up being encapsulated in this *automatic 'observation loop'*, as it is frequently called. In simple cases, making use of this invisibly present loop allows to avoid writing an explicit loop as shown above. In addition, SAS busies itself with two things:

1. Moves missing values to all variables not residing in the memory areas the compiler has designated as 'retained only'.
2. Counts the number of times control is transferred to the top of the loop by incrementing an internal counter and moving its value quantity to the automatic variable _N_ right before the next iteration begins.

However, SAS leaves no gaps between the DATA statement and the beginning of its automatic loop, or between the bottom of the loop and the RUN statement. That is, no programming statements can be inserted before and after the automatic loop. Emulated explicitly, the automatic loop might look like this:

```
Data ….. ;
  < Move missing to all non-retained variables >
  Do Internal = 1 By +1 Until ( End_of_File ) ;
    _N_ = Internal ;
    < Top of the step as we see it >
     ……
    < Our instructions >
     ……
    <If no explicit output statement elsewhere OUTPUT>
    < Bottom of the step as we see it >
  End ;
  Stop ;
Run ;
```

It is all dandy if the body of the loop is only what matters, because then the automatic loop makes things a little bit more concise, e.g. by saving DO, END, OUTPUT, and maybe STOP statements. However, all the advantage in thus attained parsimony evaporates as soon as there is something to be done before and/or after reading the whole file. Indeed, how do we do it, being limited only to the possibility of writing instruction inside the loop? Here is how the difficulty is usually circumvented:

```
Data AwaterNmiss ( Keep = State Water )  ;
  If _N_ = 1 Then Do ;
    Time = Time() ;
    Put Time = Time. ;
  End ;
  If End_of_File Then Do ;
    Average = Round ( TotalWater / NonMiss, 1 );
    Put Average = Comma. ;
    Time = Round (Time() – Time, .001 ) ;
    Put Time = ;
  End ;
  Set States End = End_of_File ;
  If State NE: 'A' or Water = . Then Delete ;
  NonMiss    ++       1 ;
  TotalWater ++ Water ;
Run ;
```

Note that above, the DELETE statement is an exact counterpart of the CONTINUE statement in that it transfers control directly to the very bottom of the loop (in this case, implicit one), whence it goes back to the top, thus bypassing the (implicit) OUTPUT and having the rather oblique meaning of 'deleting the observation'.

If this, 'conventional', style and this kind of programming logic (or lack of it thereof) suits one better, it can be of course perceived as a matter of personal preference. Keep in mind, though, that it makes the computer ask the question about _N_=1 and End_of_File=1 just as many times as there are records to read. So, if you have, as it often happens nowadays, 100,000,000 or so observations to process, you might want to rethink the 'standard' approach. Moreover, it is not accidental that when using the automatic loop, the testing for the end of file must be placed at the top of the step, contrary to the stream of consciousness. If it were located after the file processing statements, where it logically belongs, you might not see AVERAGE computed and AVERAGE and TIME printed at all! The reason is simple. Suppose the values in the last record satisfy the condition

If State NE: 'A' or Water = . Then Delete ;

If that is the case, control goes to the top of the implicit loop; from there, all instructions are executed serially until control bumps into the SET (of other I/O) statement having nothing to read, and thus the step is terminated. No instruction placed after the conditional sentence are executed if it evaluates true, which would be the fate of AVERAGE and TIME, were not they placed *before* SET.

4.3. LINK SUBROUTINES

Those familiar with GOSUB subroutine in BASIC or COBOL paragraphs will be pleased to know that this kind of functionality is fully supported in SAS DSL. You can define a block of statements, supply a statement label for its first statement, and close it with a RETURN statement:

```
CALLME: < SAS statements> RETURN ;
```

Then anyplace in the code CALLME can be invoked as follows:

```
LINK CALLME ;
```

In fact, it can be called from CALLME itself, albeit no more than 10 nested levels are allowed. When CALLME is called, control is transferred to the statement following the label and on until the entire block is performed; then control is returned (hence the RETURN verb) to the instruction immediately after LINK.

4.3.1. 'Logical'

If you decide to use the 'logical' SAS DSL programming (in the sense above), the most natural place for a subroutine is after the STOP – for it guarantees that no subroutine can be executed unless called. For example, if you wanted to combine the two SUM statements above in a LINK module ACCUM, and the statements computing and printing average – in a module MEANS, it might look like this:

```
Data AwaterNmiss ( Keep = State Water )  ;
  Time = Time() ;
  Put Time = Time. ;
  Do Until ( End_of_File = 1 ) ;
    Set States End = End_of_File ;
    If State NE: 'A' or Water = . then Continue ;
    Link ACCUM ;
    Output ;
  End ;
  Link MEANS ;
Stop ;
    ACCUM: NonMiss ++1 ; TotalWater ++ Water ;  Return ;
    MEANS: Average = Round ( TotalWater / NonMiss, 1 ) ;
              Put Average = Comma. ;
Run ;
```

In reality, RETURN closing a LINK module is only needed to separate it from other module defined right after it – otherwise they would be perceived by the compiler as one. That is why after MEANS, RETURN can be painlessly omitted, but after ACCUM, it is required.

4.3.2. 'Traditional'

It gets less attractive if the automatic loop is used. In this case, LINK module definitions must be placed after yet another RETURN, its role being no different from 'GO TO TOP' .

```
Data AwaterNmiss ( Keep = State Water )  ;
  If _N_ = 1 Then Do ;
    Time = Time() ;
    Put Time = Time. ;
  End ;
  If End_of_File Then Do ;
  Link MEANS ;
    Time = Round (Time() – Time, .001 ) ;
    Put Time = ;
  End ;
  Set States End = End_of_File ;
  If State NE: 'A' or Water = . Then Delete ;
  Link ACCUM ;
RETURN;
    ACCUM: NonMiss ++1 ; TotalWater ++ Water ;  Return ;
    MEANS: Average = Round ( TotalWater / NonMiss, 1 ) ;
              Put Average = Comma. ;
Run ;
```

Finally, a LINK module can be principally placed anywhere in the code if it is enveloped in a never-executable shell. In the case of the ACCUM routine, for example (either of the following two variants are workable):

```
If 0 Then Do;
   ACCUM: NonMiss    ++1        ;
             TotalWater ++ Water ;
End ;

Do While (0);
    ACCUM: NonMiss     ++       1 ;
              TotalWater ++ Water ;
End ;
```

Of course, if each module is defined this way, no RETURN is required *anywhere*. This method makes it possible to define LINK routines *before* they are called (analogous the SAS macro-style, for that matter). However, one should bear in mind that this kind of definition contains an extra comparison, so care should be taken to place any such definitions outside of any frequently iterating structure.

Whether this way of defining LINK modules can be seen as an advantage or not, depends on the personal programming style and one's propensity to split a program in a number of hierarchical paragraphs similar to the way COBOL programmers perceive as the 'structured programming'. (As a matter of course it has nothing to do with the structured programming whatsoever). There indeed may be situations where it may be more logical to arrange the program around several LINK modules, rather than macros, say. However, the functionality of a LINK routine is limited by the fact that it cannot be parameterized and shared all its variables with the caller.

IV. CONCLUSION

Even if you have had enough patience and courage to get here without using a GOTO CONCLUSION instruction, but rather by faithfully executing all instructions all the way from INTRODUCTION, you have only seen a shining tip of a giant SAS DATA step iceberg. Hopefully, it has given you a little bit of idea what kind of beauty dwells within.

V. REFERENCES

1. Rick Aster, Rhena Seidman. Professional SAS Programming Secrets, Wincrest/McGraw-Hill, 1991.
2. T.W. Pratt. Programming Languages. Design and Implementation, 2nd Edition, Prentice-Hall, Inc.
3. SAS Language. Reference. Version 6, 1st Edition, SAS Institute, Inc., Cary, NC.

VI. AUTHOR CONTACT INFORMATION

Paul M. Dorfman,
SAS Programmer

10023 Belle Rive Blvd. 817
Jacksonville, FL 32256

(904) 564-1931 (h)
(904) 905-5428 (o)

sashole@bellsouth.net
paul_dorfman@hotmail.com
paul.dorfman@bcbsfl.com

Passing Along SAS Data – SET, MERGE, and UPDATE

Andrew T. Kuligowski – Nielsen Media Research

Abstract / Introduction

Once a SAS dataset is created, the user will often want to modify it or combine it with other datasets. This presentation will cover three basic commands that can be employed to use, reuse, and combine existing SAS datasets: SET, MERGE, and UPDATE.

The following topics will be covered in this presentation:

- The **LIBNAME** statement – permanently storing your SAS dataset.
- The **SET** statement – reusing a SAS dataset.
- The **MERGE** statement – bringing two SAS datasets together.
- The **BY** statement – controlling the order of integration.
- The **UPDATE** statement – changing an existing SAS dataset.

The goal of this presentation is to provide the attendee with the background necessary to permanently catalog and reuse their SAS data.

LIBNAME Statement

All SAS datasets are stored in SAS Data Libraries. In fact, the standard specification for a SAS dataset contains two levels – the SAS Data Library name, and the individual SAS dataset name, separated by a period, or "dot" if you prefer. Of course, even the newest of SAS users will realize that many SAS datasets are represented by only one name. This is because the default SAS data library is "WORK". WORK is automatically defined when SAS is invoked, and the absence of a Data Library name in a SAS dataset name causes SAS to use the WORK library.

Each SAS Data Library must be denoted by a LIBNAME. A LIBNAME, also commonly known as a LIBREF, is a shorthand representation, or nickname if you prefer, of the actual dataset name as defined by your particular operating system.

There are several ways to define a LIBNAME to the SAS System. One of them is to provide a file reference to the external file using a host system command outside of the SAS System - a JCL "DD" statement under IBM's MVS, for example. To cite an example that has already been referenced, this is the method by which the WORK library is allocated to the SAS session.

An alternate method is to use the **LIBNAME** statement under SAS. The LIBNAME statement is a global statement, and is executed outside of the DATA step. The generic syntax for this statement is:

```
LIBNAME libref  <engine>
        'external file' <options> ;
```

"Engine" and "options" will not be discussed at this time; they will be deferred to a more advanced presentation. As one might expect, however, many of the options for the LIBNAME statement are dependent on the operating system. Consult the appropriate "Companion" document for your operating systems for details.

It is also worth noting that the LIBNAME statement can be executed with two special keywords under any operating system:

- **CLEAR** will remove a reference to an existing SAS Library:
  ```
  LIBNAME libref CLEAR;
  ```
- **LIST** will write the attributes of the specified LIBNAME to the SASLOG:
  ```
  LIBNAME libref LIST;
  ```

There is also a **LIBNAME** *function* that can be invoked from within a DATA step to perform the same purpose. The syntax is similar to the LIBNAME statement:

```
LIBNAME(libref, 'external file',
        ,<engine>, <options> );
```

Details on how and when to use functions will be discussed as part of another presentation in the "Introduction to SAS" series.

SET Statement

According to the *Version 6 SAS Language Reference*, the **SET** statement "… reads observations from one or more existing SAS datasets." Under ordinary circumstances, each variable, on each observation, on each SAS dataset specified by the SET statement, is read into the SAS

Program Data Vector, and each is subsequently written out to the new output SAS dataset - although this can be overridden with other logic in the DATA step.

The basic SET statement is very simple:

```
DATA  newdata;
   SET olddata;
   /* additional statements */
RUN;
```

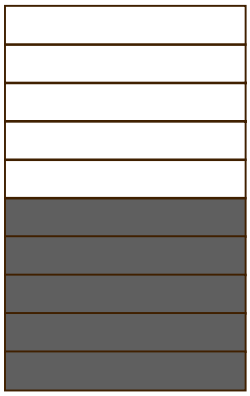
In this simple example, the SET statement introduces each record in SAS dataset *olddata* into the current DATA step. By default, those records are subsequently written out to SAS dataset *newdata,* although any or all of those records can be either output or discarded on a conditional basis if desired.

The SET statement is commonly used to *concatenate* two or more SAS datasets together. Let us look at a basic example:

```
DATA  newdata;
   SET  olddata1  olddata2;
   /* additional statements */
RUN;
```

In this example, the SAS Data Step will first read in each record from SAS dataset *olddata1* and, by default, write each of those records to SAS dataset *newdata*. Then, after the last record in *olddata1* has been processed, each record in *olddata2* will be read in and subsequently written out to *newdata*. By default, each observation in *newdata* will contain every variable in *olddata1* and every variable in *olddata2*. If there are variables that are defined in *olddata1* but not *olddata2*, the values will default to missing for those variables on the observations that are copied from *olddata1*. The opposite will be true for variables defined on *olddata2* but not on *olddata1*.

Depicted graphically, a single SET statement combines two or more SAS datasets *vertically*. Using the simple routine listed above, we would get:



There is an important condition that the SAS coder must be aware of when combining two or more SAS datasets. Variables that are common to two or more SAS datasets must have consistent definitions in each of those datasets in order to be used successfully in the SET statement (or, as will be discussed later, the MERGE or UPDATE statements). The most severe situation is when a variable is defined as character in one SAS dataset, but as a numeric in another. Should this occur, SAS will trip the internal _ERROR_ variable to 1 (for "true), set the return code to 8, and write the following ERROR message on the SASLOG:

```
ERROR: Variable variablename has
       been defined as both
       character and numeric.
```

If the common variables are both character or both numeric, but have different lengths, the output dataset will use the first length it encounters for that variable. This will be typically be in the first dataset in the SET statement. However, this default action can be overridden by inserting a LENGTH statement prior to the SET statement, as follows:

```
DATA  newdata;
   LENGTH commonvr $ 25.;
   SET  olddata1  olddata2;
   /* additional statements */
RUN;
```

It should be noted that PROC APPEND can also be used to concatenate two SAS datasets, and it can often be more efficient than using a SET statement within a DATA step. However, PROC APPEND can only be used on two SAS datasets, while the SET statement can be used on three or more SAS datasets. In addition, the SET statement can be used in conjunction with other DATA Step statements to further massage the input data and to perform conditional processing; this additional functionality is not possible using PROC APPEND.

The SET statement can also be used to *interleave* two or more SAS datasets. In order to do this, each of the desired SAS datasets must contain the same key variables, and each must be sorted by those key variables. Then, the SET statement must be immediately followed by a **BY** statement listing those key variables, as follows:

```
DATA  newdata;
   SET  olddata1  olddata2;
   BY  keyvar1  keyvar2;
   /* additional statements */
RUN;
```

In the preceding example, *newdata* will contain the observations from both *olddata1* and *olddata2*. It

will be sorted by the variable(s) referenced in the BY statement, in this case, *keyvar1* and *keyvar2*.

The addition of a BY statement causes the records to be interleaved, but does not alter the fact that the records are combined vertically. Graphically, this would look as follows:

At this point, let us briefly diverge from discussion of the SET statement to review the **BY** statement. When used in a DATA step, the BY statement is used to control the operation of the SET, MERGE, UPDATE, or MODIFY statements. A BY statement applies only to the statement that immediately precedes it in the DATA Step; that statement must be one of the four listed in the previous sentence. The basic syntax of the BY statement is trivial, simply listing one or more variables after the word "BY" as follows:

```
BY  <DESCENDING> var1
   <<DESCENDING> var2> …;
```

The **DESCENDING** keyword, as one might expect, is used if the data is to be found in reverse sequence.

Please note that there is also a **GROUPFORMAT** keyword that can be used to process a variables formatted value rather than its stored values. There is also a **NOTSORTED** keyword that can be used under certain circumstances. However, discussion of these two features is best left to a more advanced presentation.

Returning to the SET statement, it is also possible to use the SET statement to perform *direct access* (also known as *random access*) queries against a SAS dataset in order to retrieve a specific record. This is done by using the POINT= option on the SET statement. POINT= is followed by the name of a temporary SAS variable, which must have an integer value between 1 and the maximum record number in the SAS dataset being processed. This number can be easily obtained by using another option, NOBS=. NOBS= specifies a temporary variable that is populated during DATA step compilation with the number of observations in the applicable SAS dataset.

This concept is sometimes difficult to explain and to comprehend with only words; an example is needed for clarification. The following routine reads in every other record of a SAS dataset:

```
DATA  newdata;
   DO recno = 2 TO maxrec BY 2;
      SET  olddata  POINT=recno
           NOBS=maxrec;
      /* additional statements */
   END;
   STOP;
RUN;
```

Note that a STOP statement is required in order to terminate the current DATA step. This is because DATA step processing is concluded when the INPUT statement reads the End-of-File. However, when using the POINT= option, the INPUT statement never processes the End-of-File marker. Therefore, the DATA step is never terminated – not unless the coder specifically orders it via the STOP statement.

As stated earlier, NOBS= is populated at compile time, rather than during execution. This allows us to reference the number of records in a SAS dataset without actually executing the SET statement! The following example demonstrates this principle:

```
DATA  _null_;
   PUT maxrec 'Recs in olddata.';
   STOP;
   IF 1 = 0  THEN
      SET  olddata NOBS=maxrec;
RUN;
```

In this example, the SET statement defining the variable *maxrec* is coded AFTER the variable is to be displayed. Furthermore, the "IF 1 = 0" condition can obviously never be true, so the SET statement never actually executes. However, the variable *maxrec* is correctly populated with the record count of *olddata* during the compliation of the DATA step. This allows the record count to be printed, using the PUT statement, in the first line of the DATA step.

END= is another useful option. It specifies a temporary SAS variable that is normally set to 0 ('false'). However, it is "tripped" and reset to 1 ('true') when the INPUT statement encounters the last record in the SAS dataset being read (or, if multiple datasets are specified, the last record in the last SAS dataset on the INPUT statement) This can facilitate any extra end-of-file processing that is required for the current DATA step.

```
DATA  newdata;
   SET  olddata  END=lastrec;
   /* additional statements */
   IF  lastrec = 1  THEN  DO;
      /* additional end-of-file */
      /* processing statements  */
      /* go here.               */
   END;
RUN;
```

There are a number of SAS Dataset options that can be used to enhance the processing of the SET statement. Dataset options must be enclosed in parentheses, and must immediately follow the dataset which they are describing. For example, a common request is to create a subset of a SAS dataset. This can be done by inserting an IF statement or a WHERE statement in the DATA step. However, it is also possible to do this by using the WHERE= dataset option in conjunction with the SET statement.

```
DATA  newdata;
   SET  olddata
        (WHERE=( keyvar < 10 ));
   /* additional statements */
RUN;
```

Dataset options can also be used to limit the number of records processed by the SET statement. FIRSTOBS= causes processing to start at a specified observation number, while OBS= causes processing to start at a specified observation number. They can be used together; the following example only processes the 1000th through the 2000th observations of a SAS dataset:

```
DATA  newdata;
   SET  olddata
        (FIRSTOBS=1000 OBS=2000);
   /* additional statements */
RUN;
```

There are several instances when it can be advantageous to use two or more SET statements within the same DATA step. For example, one coding technique is to store constants in a separate dataset, bringing them in at the beginning of a DATA step, as follows:

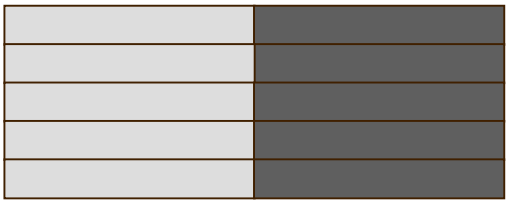
```
DATA  newdata;
   IF  _N_ = 1  THEN DO;
      RETAIN konst1-konst5;
      SET konstant;
      END;
   SET  olddata;
   /* additional statements */
RUN;
```

In this example, note that the SAS dataset *konstant* is only read during the first iteration of the DATA step, and that only one record is processed from *konstant*. The values that were read in from this record will be available throughout the entire execution of the DATA step, due to the RETAIN statement.

It is also possible to conditionally use the contents of one dataset to process another dataset, as in the following example:

```
DATA  newdata;
   SET  employee;
   IF  married = 'Y'  THEN
      SET  spouse;
   IF  childcnt > 0  THEN
      DO  i = 1 TO childcnt;
         SET  children;
      END;
   /* additional statements */
   /* go here.              */
RUN;
```

In this example, it is assumed that all 3 datasets are sorted by some common key variable. Following through the logic, it is assumed that every married employee has one and only one spouse, and the appropriate record is obtained. Furthermore, the routine determines the number of children that the employee has, and processes one record for each of them.
Depicted graphically, two or more SET statements combine SAS datasets *horizontally*. Basically, it would look as follows:



It should be noted that multiple SET statements could actually produce undesirable results. Since DATA step processing stops after the SET statement encounters an End of File marker – the *first* End of File marker in the case of multiple input datasets – the *smallest* dataset drives the number of iterations for the DATA step. In addition, the logic can get very cumbersome. There is an easier way to accomplish the same thing …

MERGE Statement

The **MERGE** statement " … joins corresponding observations from two or more SAS datasets into

single observations in a new SAS dataset," to quote from the *Version 6 SAS Language Reference*.

The simplist example of the MERGE statement would be a *one-to-one* merge, as follows:

```
DATA  newdata;
   MERGE olddata1 olddata2;
RUN;
```

In this example, this routine takes the 1st record in *olddata1* and the first record in *olddata2*, and joins them together into a single record in *newdata*. This is repeated for the 2nd records in each dataset, the 3rd records, etc.

This technique is not often used in the real world, or more accurately, not often used intentionally. The main problem is that it is grounded in the assumption that there is a record-to-record relationship in each of the datasets to be merged, regardless of the contents of those records. This is not often the case.

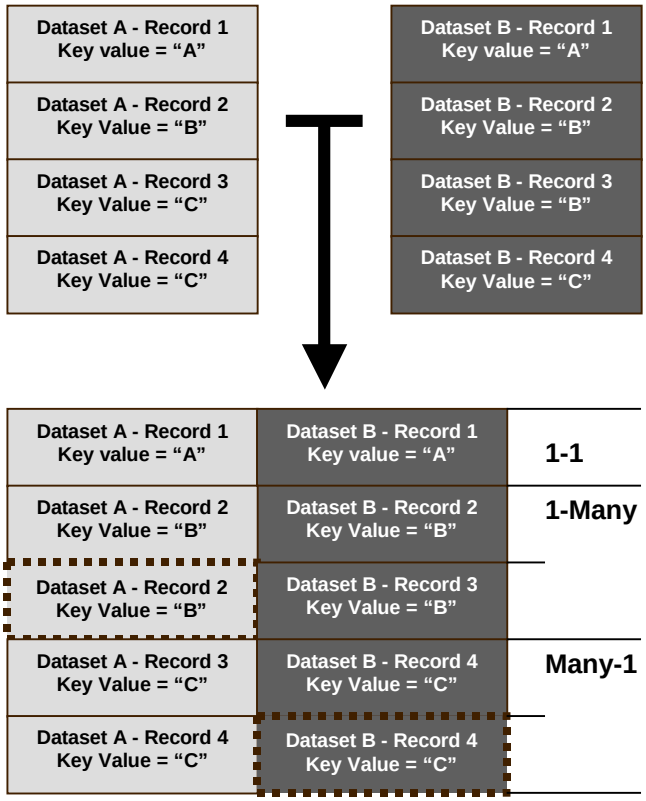
It is much more common to find that there is a record-to-record relationship based on the values of key fields found in both files. This approach, known as *match-merging*, requires all of the files in the MERGE statement to be sorted by the same variable(s). To show a simple example:

```
DATA  newdata;
   MERGE olddata1 olddata2;
   BY  keyvar(s);
RUN;
```

Match-merging is a very powerful tool. With two statements - MERGE, followed immediately by BY – the DATA step can handle cases of:

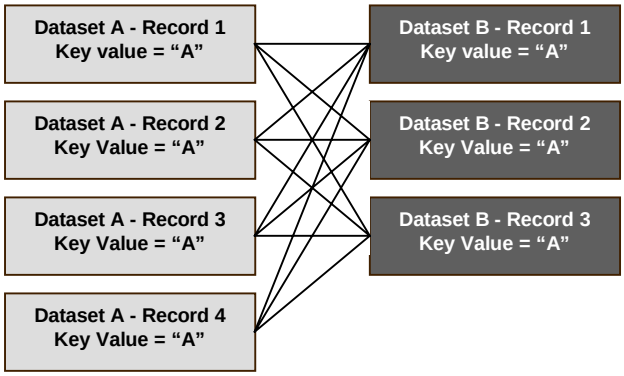
- One-to-one matching, where there is one record in each file containing the same key values.
- One-to-many matching, in which the first file has one record containing a particular set of key values and the second (or subsequent) file has multiple records containing those same key values, and
- Many-to-one matching, where the first file has multiple records containing a unique set of key values and the second file has only one record with those key values.

This is best shown graphically. The following example shows 2 datasets, both with 4 records each. Both files begin with a first record containing the same key value, illustrating 1-1 matching. The first file then contains one record with a different key value while the second file contains two records with that key, showing 1-many matching. Finally, the first file contains two records with another unique key, while the second file only has one record with that key, demonstrating many-1 matching:



It is also significant to note what the example does not show. There are no examples of "1 to null" or "null to 1", or "many to null" or "null to many" merges, although these are all valid. These occur when a given key value exists in one file, but not the other. Each of these is a valid MERGE condition, however, and will result in record(s) being written to the output file. In this case, any variable that is only present on the file without the current key value will contribute null values for those variables to the final output dataset.

We do also not have any examples of many-to-many merges. This condition occurs when both files have multiple record with the same key values present. To illustrate this graphically:



SAS cannot process many-to-many merges with the MERGE statement. They result in an ominous note to the SASLOG, and undesirable output results. However, be warned - the routine does NOT terminate with a non-zero condition code. In fact, processing continues on as though nothing is wrong, even though the results of the merge are almost definitely NOT what the author intended! Note that PROC SQL is capable of processing many-to-many merges. However, this is outside the scope of this presentation.

It is possible to programmatically prepare your data to avoid the risk of attempting a many-to-many merge by removing multiple records with the same key values, using the NODUPKEY option on PROC SORT, as follows:

```
PROC SORT DATA=olddata1  NODUPKEY;
   BY  keyvar(s);
RUN;
```

However, this approach can also delete records that should actually be processed. The reader is encouraged to read this author's "Pruning the SASLOG…" presentation, as cited in the "References" section at the end of this presentation, for techniques to eliminate many-to-many merges.

Variable names that are common to two or more SAS datasets on the MERGE statement must have the same definition on each of those datasets, just like with the SET statement. However, unlike the SET statement, the value from the second dataset will overlay the value from the first dataset with MERGE. Sometimes this is desirable, while other times it is unwanted. In the latter case, the RENAME= option will allow the variables from *both* datasets to be written to the new output dataset, albeit with different names.

The use of the RENAME= option is simple:

```
MERGE olddata1
        (RENAME=(oldname=newname))
      olddata2;
```

The output dataset will contain the values of the variable *oldname* from both *olddata1* and *olddata2*, although the value from *olddata1* will be stored in the new variable, called *newname*.

It is possible to perform conditional processing, based on the source dataset for each record, by using the IN= dataset option. The IN= parameter creates a temporary variable associated with the dataset for which it is specified. The variable is set to 1 (for "True") if the specified dataset is contributing data to the current observation; otherwise it is set to 0. This allows for specialized processing, depending on the source of the current data. IN= can be used with the SET statement, but it is much more commonly found along with the MERGE statement.

Let us look at a simple example of IN=, with subsequent conditional processing:

```
DATA  newdata;
   MERGE olddata1(IN=in_old1)
         olddata2(IN=in_old2);
   BY  keyvar(s);
   IF in_old1 AND in_old2 THEN
      /* additional statement(s)*/
   IF in_old1 AND NOT in_old2 THEN
      /* additional statement(s)*/
   IF NOT in_old1 AND in_old2 THEN
      /* additional statement(s)*/
   IF NOT in_old1 AND
      NOT in_old2  THEN
      /* additional statement(s)*/
RUN;
```

The variable *in_old1* will be set to 1 (true) when the current observation in *newdata* is being fed from *olddata1* and 0 (false) when it is not. The same thing is true for variable *in_old2* and SAS dataset *olddata2*. The subsequent IF statements allow for special processing if the current observation contains data from both *olddata1* and *olddata2*, from *olddata1* but not *olddata2*, from *olddata2* but not *olddata1*, and from neither *olddata1* nor *olddata2*.

Of course, alert readers will quickly realize that the 4th and final IF statement is not necessary in the previous example. Since the DATA step is being fed from observations in *olddata1* and *olddata2*, there will never be an instance where the routine will be processing a record that is not present in either dataset! (It is sometimes beneficial to embed this explanation in a comment within your code, depending on your audience.) In addition, the experienced coder will quickly realize that this example could be made more efficient by using the ELSE statement; however, that is outside the scope of this presentation.

It should be noted at this point that most options available for the SET statement are also valid for use with MERGE, and vice verse. (Notable exceptions are POINT= and NOBS=, which are exclusive to the SET statement.) These topics are introduced in this paper under the command where the author has personally found them to be of most use in his daily activies.

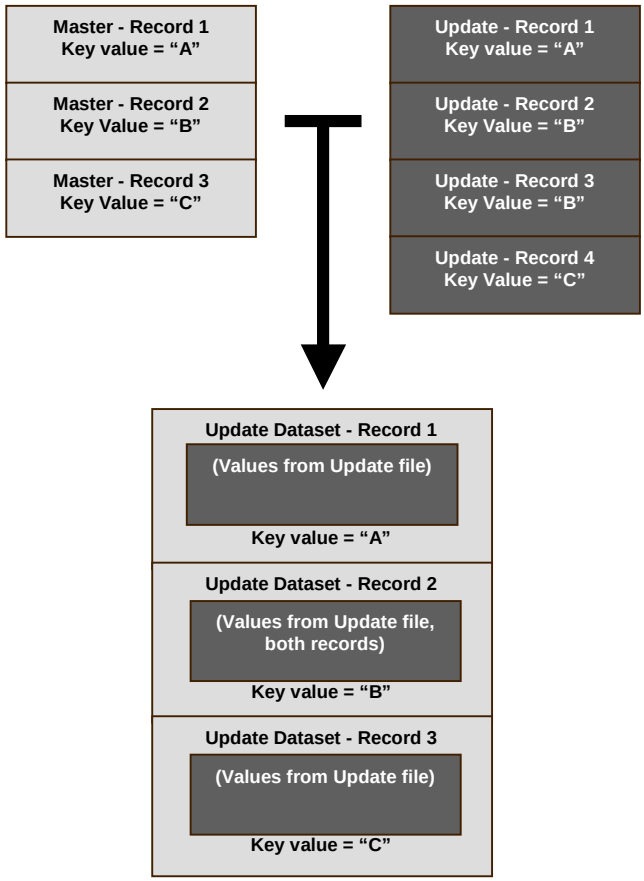
UPDATE Statement

The **UPDATE** statement is similar to the MERGE statement, "… but the UPDATE statement performs the special function of updating master file information by applying transactions …" to quote once more from the *Version 6 SAS Language Reference*.

The UPDATE statement combines records from two files in a horizontal fashion, like the MERGE statement. However, there are a number of significant differences between the two statements:

- UPDATE can only process two SAS datasets at a time – the *Master* dataset and the *Transaction* dataset. A single MERGE statement can process 3 or more SAS datasets.
- The BY statement is optional (although typically used) with MERGE, but is required with UPDATE.
- UPDATE can only process one record per unique BY group value in the Master dataset. It can process multiple records per unique BY group value in the Transaction dataset. However, in this case, each Transaction record is applied to the same record in the Master dataset, which means that transactions can be overlaid by subsequent transactions within the same DATA step.
- The UPDATE statement can avoid overlaying any given value in the Master dataset with a value in the Transaction dataset by setting the corresponding value in the Transaction dataset to missing. This would require more complex conditional logic to accomplish via the MERGE statement.

Depicted graphically, the UPDATE statement performs a modified version of a horizontal merge, in which values on the original records are overlaid with new information:



The UPDATE statement is simple to code – in fact, the only difference between it and the earlier MERGE statement / BY statement example is the word "UPDATE":

```
DATA  newdata;
   UPDATE olddata1 olddata2;
   BY  keyvar(s);
RUN;
```

However, the true utility of the UPDATE command becomes apparent after examining "before" and "after" sample data:

SAS Dataset: MASTER

OBS	KEY1	UPDT1	UPDT2
1	AL	1001	2
2	FL	1002	4
3	GA	1003	8
4	MS	1004	16

SAS Dataset: XACTION

OBS	KEY1	UPDT1	UPDT2	NEW3
1	AL	1111	52	A
2	GA	1133	54	
3	GA	2133	.	C
4	MS	.	.	D
5	LA	4555	65	E

SAS Dataset: UPDATIT

OBS	KEY1	UPDT1	UPDT2	NEW3
1	AL	1111	52	A
2	FL	1002	4	
3	GA	2133	54	C
4	MS	1004	16	D
5	LA	4555	65	E

In this example, the first record in the Master File (for KEY1=AL) has its values changed for fields UPDT1 and UPDT2, as well as a value added for newly created field NEW3. The second record (KEY1=FL) is untouched, and NEW3 is set to missing. The third

record (KEY1=GA) is actually changed twice due to two separate records in the Transaction dataset, with only the final set of changes stored to the output file. The fourth record (KEY1=MS) does not have its original two values adjusted, because the values are set to missing in the transaction dataset. However, a value is inserted for the new variable NEW3. The last record in the Transaction file (KEY1=LA) does not exist in the Master dataset, so it is added.

The question arises – what if you WANT to replace an existing value with a missing value? This is possible, but requires a little extra coding. The MISSING statement, added to the DATA step, will define special missing values that can be used in the Transaction data. The values "**A**" through "**Z**" will display as coded, while an underscore "_" will invoke the standard missing data representation of a dot "**.**". Note that the MISSING command must be present during both the creation of the transaction data and in the UPDATE DATA step.

MODIFY Statement

The **MODIFY** statement was added in Version 6.07 of the SAS System. It has many of the same capabilities of the SET, MERGE, and UPDATE statements – with one important difference. To quote from *SAS Technical Report P-222*, the MODIFY statement "… extends the capabilities of the DATA step, enabling you to manipulate a SAS data set in place without creating an additional copy."

Neither SET, nor MERGE, nor UPDATE has the ability to update a dataset in place. They give the appearance of doing that when the DATA statement specifies a dataset name that is the same as one in SET, MERGE, or UPDATE. However, in actuality, the DATA step creates a new dataset in this instance, and then replaces the old dataset upon completion. The MODIFY statement avoids the creation of this temporary dataset, along with the extra temporary disk storage that it requires and the time – processing and clock – it takes to make it.

However, as one might expect, there is a trade-off that must be considered before using the MODIFY command. To quote from the manual: ***'Damage to the SAS data set can occur if the system terminates abnormally during a DATA step containing the MODIFY statement."*** The user must take special care to prevent their dataset from being corrupted while being MODIFY-ed, whether the problem is caused by the execution of buggy code or from the careless absence of a UPS upon a power outage.

The most basic form of the MODIFY statement is quite simple:

```
DATA  dataset;
   MODIFY dataset;
   /* additional statements */
RUN;
```

It acts much like the SET statement discussed earlier. However, there are some major differences between the two:

- As mentioned above, the MODIFY statement causes the current dataset to be changed in place -without expending the I/O and disk space to generate a new dataset. Therefore, as should be expected, the name of the output dataset *must* be the same as the one being modified.

- Additional variables cannot be added and unneeded variables cannot be deleted from a MODIFY-ed dataset. It should be noted that no ERROR or WARNING statement will be generated if these statements are present in the SAS code; the requested structure alterations will simply not be applied.

- The **REPLACE** statement causes the current record to be rewritten in the MODIFY-ed SAS dataset with any changes applied. The **OUTPUT** statement causes a new record to be written to the MODIFY-ed dataset – however, the original record will still be present on the output file, resulting in two separate records upon completion of the DATA step. This difference cannot be sufficiently emphasized – the OUTPUT statement, when used with MODIFY causes a second, duplicate record to be added to the end of the current SAS dataset. The user must be certain not to use OUTPUT instead of REPLACE, which changes the current record in place and does not result in an additional record.

- You cannot DELETE an observation when using the MODIFY statement. You can, however, use the **REMOVE** command to eliminate unwanted observations.

The MODIFY statement can be used for sequential (random) access. The syntax is similar to that of the SET statement as discussed earlier:

```
DATA  dataset;
   DO recno = 2 TO maxrec BY 2;
      MODIFY  dataset  POINT=recno
              NOBS=maxrec;
      /* additional statements */
   END;
   STOP;
RUN;
```

The need for the STOP statement is also similar to that of the SET statement - assuming the coder wishes to avoid the pitfalls of an infinite loop.
It is also possible to join two or more SAS datasets using the MODIFY statement. As with the other examples in this section, the syntax is similar to that of the SET and UPDATE statements.

```
DATA  master;
   MODIFY  master  transact;
   BY  keyvar1  keyvar2;
   /* additional statements */
RUN;
```

Both the master and transaction datasets must contain the same key variables. However, they do *not* need to be sorted by those variables – the presence of a BY statement causes the SAS System to invoke a dynamic WHERE clause. Please note that, although not required, it is highly recommended for efficiency sake that the datasets be sorted or indexed by the key variables in the BY statement.)

There is one other important difference - multiple records with the same key values act differently when processed with the MODIFY statement. Duplicate key values in the master file will *not* be altered – only the first record of occurrence is updated due to the aforementioned WHERE clause processing. (As one might expect, multiple records in the transaction file will overwrite each other, so that only the changes in last transaction record in the series will be available in the master dataset at the end of the DATA step.) These differences provide a "safety valve" to prevent unwanted alterations to your permanent SAS data.

Speaking of "safety valves", the user is highly encouraged to incorporate the automatic variable _IORC_ into their routines. _IORC_ contains the return code for every I/O operation that is performed by the MODIFY statement – or rather, that for each one that is attempted. The simplest use would be to simply ensure that the field contains a zero before continuing on with the routine. It can be made more complex, with logic handling specific errors, or by using the IORCMSG function to obtain and display the associated error message for the return code. (Note that IORCMSG is not available under Version 6 of the SAS System.)

Conclusion

This presentation is designed to be a brief introduction to the SET, MERGE, UPDATE, and MODIFY commands. It is not a "shopping list" of the various options available for each command; that information is readily available in the manuals, as listed in "References" below. The reader is encouraged to sit down at the computer and try examples of each command to facilitate his or her learning of the subject; only after hands-on trial will the information truly be meaningful to the reader.

References / For Further Information

Kuligowski, Andrew T. (1999), "Pruning the SASLOG – Digging into the Roots of NOTEs, WARNINGs, and ERRORs". *Proceedings of the Seventh Annual Conference of the SouthEast SAS Users Group*. USA.

Riba, S. David. "The SET Statement and Beyond: Uses and Abuses of the SET Statement". *http://www.jadetek.com/download/jade_set.pdf*

SAS Institute, Inc. (1990), *SAS Language: Reference, Version 6, First Edition.* Cary, NC: SAS Institute, Inc.

SAS Institute, Inc. (2000), *SAS OnlineDoc, Version 8.* Cary, NC: SAS Institute, Inc.

SAS Institute, Inc. (1994), *SAS Software: Abridged Reference, Version 6, First Edition.* Cary, NC: SAS Institute, Inc.

SAS Institute, Inc. (1991). *SAS Technical Report P-222, Changes and Enhancements to Base SAS Software, Release 6.07.* Cary, NC: SAS Institute, Inc.

Special thanks to Dave Riba and Ian Whitlock for their suggestions while preparing this paper.

The author can be contacted via e-mail as follows:

Andrew T. Kuligowski
A_Kuligowski@msn.com
kuligoat@tvratings.com

Understanding and Using Functions

Frank C. DiIorio
Advanced Integrated Manufacturing Solutions, Co.
Durham NC

Introduction

Let's start by admitting that programmers are, at heart, rather lazy. We want procedures to do our sorting, printing, and analysis. We want formats to display the number '5' as 'Well above average'. In essence, we don't want to code any more statements than we have to. With that in mind, let's look at another code and time-saving feature of the SAS System.

This paper gives an overview of functions, a powerful set of tools in the SAS System. Functions are a set of predefined routines that come with the SAS System. They perform a wide range of activities, and often reduce complex computations that would require arduous and error-prone DATA step coding to a single, simple statement. Not even a novice SAS programmer's toolbox is complete without a basic knowledge of system functions.

This paper introduces SAS novices to functions. Basic terminology is reviewed first, followed by usage issues common to nearly all functions. The last section of the paper describes the purpose and syntax of some of the more commonly used functions. Bear in mind that this paper is simply an overview of a broad and sometimes complex topic. The reader should consult SAS Institute documentation for the definitive, exhaustive description of the purpose, limitations, and uses of the functions.

Fundamentals

The logic that's common to all functions is straightforward. Two components of function syntax – the function name and its parameters – identify what is to be performed. The first component is the function name. It identifies the action that the function performs – identify the minimum of a list of numbers (the MIN function), locate the third word in a character variable (the SCAN function), and so on. The name usually gives some idea of the activities performed by the function.

The second function component is a list of *parameters* (sometimes referred to as *arguments*) enclosed in parentheses. Notice in the above description of the function names we said "minimum *of* a list" and "third word *in* a character variable." The "of" and "in" identify *what* the function should operate on. Putting these two pieces together, let's look at two complete uses of these functions:

```
min_year = min(fy2001q1, fy2001q2,
               fy2001q3);
file_name = scan(dir_line, 3, ' ');
```

The first statement creates the numeric variable MIN_YEAR, which is the minimum of three arguments, fy2001q1, fy2001q2, and fy2001q3. The second statement creates a character variable, FILE_NAME, which is the third piece of variable DIR_LINE, the pieces being delimited by blanks.

The functions are said to be "called" – in the first statement, we called the MIN function, in the second, we called the SCAN function. Functions are said to "return" a value – in the first statement, we call the MIN function and it returns a value, stored in MIN_YEAR, that is the minimum of the three arguments passed to it.

Both the idea and the syntax are simple – specify the appropriate function name and give the function parameters and results are returned. Just as procedures perform a great deal of work with relatively few statements, so do functions simplify potentially tedious calculations. Before looking at what specific functions do, let's look at some logical and syntactical issues common to all of them.

Syntax and Usage

Before using this paper or the SAS documentation's description of the *many* available functions, consider the following points carefully:

Functions Can Be Used Pretty Much Anywhere. Functions usually perform some form of calculation, and calculations are usually considered in the context of the DATA step. Keep in mind that, with some documented exceptions, functions can be used anywhere an evaluation of constants and/or variables can take place.

Here are some non-DATA step examples. Look at them more for their use outside the DATA step than for the actual operation they perform:

```
proc print data=subset
     (where=(index(vin, 'NR') > 0));

proc freq data=mast01;
tables grp / missing;
where nmiss(of fy1999q1–fy2001q4) > 2;
```

Names Matter (Gotcha #1). There are specific names for specific function activities. These cannot be reassigned (changing MIN to MINIMUM, for example). What you can do, and probably won't want to, once you see the result, is define an array with the same name as a function. Watch what happens:

```
data phase1;
set master;
array min(4);  /* each group's minimum */
do i = 1 to 4;
   min(i) = min(of group1-group4);
end;
```

SAS gets confused – is MIN a reference to the function or the array? The default action is to recognize MIN as an array, effectively disabling the MIN function. The following message is printed in the SAS Log:

```
WARNING: An array is being defined with the
same name as a SAS-supplied or user-defined
function. Parenthesized references involving
this name will be treated as array refer-
ences and not function references.
```

Bottom line: there are lots of words in the language. Define arrays with names that don't conflict with function names.

Names Matter (Gotcha #2). Programmers new to SAS, or those regularly moving back and forth between SAS and other languages, must be careful not to assume that function *x* in one language does the same thing in another language. Don't make assumptions. Read the documentation carefully, and be sure the functionality is identical.

If you need motivation in this regard, consider the subtle difference in this example: the TRIM function is in EXCEL and SAS, and performs basically the same activity (trimming blanks from a character variable). In EXCEL, both leading and trailing blanks are trimmed, but SAS trims only the *trailing* blanks. It's usually easier to review documentation for the functions than it is to debug their unadvised usage.

Look at Data Types Carefully. Some functions require numeric arguments, others require character arguments. Yet others require a mix of these data types. The type(s) of the argument(s) does not influence the value returned by the function. The LENGTH function, for example, returns the location of the last non-blank character in a variable. The function requires a character argument but returns a numeric value.

The Number of Arguments Varies. The number of arguments required by a function will, of course, depend on the type of work the function performs. Even using the same function, though, the number of arguments can vary. MIN and other descriptive statistic functions can handle a varying number of arguments, provided there are enough values to perform the required task (you need at least three arguments to calculate skewness, for example). Other functions expect "n" arguments and will make assumptions if they do not receive the full "n" – the third argument to the SUBSTR function, for example, is the number of positions to extract from a character variable. If omitted from the SUBSTR call, the default behavior is to subset to the rightmost position in the variable.

Parameter Order May Matter. Some functions do not care about the order in which arguments are specified. The SUM function, for example, performs an action (addition) that is, by nature, indifferent to order. Other functions are not so forgiving, and assign specific meanings to arguments. The first argument to the ROUND function, for example, is a numeric value. The second argument is the rounding unit ("round to the nearest …"). SAS is often unable to detect misspecified parameters because they may make *syntactical* sense but do not have *logical* validity. Consider the following statements:

```
inc1 = round(income, 1000);
inc2 = round(1000, income);
```

INC1 is variable INCOME rounded to the nearest 1000, while INC2 is 1000 rounded to the nearest INCOME. Both statements are syntactically valid, but only INC1 makes sense. It's important to realize that from SAS' perspective, *both* statements are acceptable. It's up to the programmer to become familiar with parameter order and meaning (and then, of course, follow through and write the statement correctly!).

Watch Out for Range Restrictions. Some functions will process any values, provided their data types are correct. Others require one or all values to be in a range of values. The restriction may be known prior to coding (the square root function SQRT cannot process a negative value), while other limits are imposed by the nature of the operation (you cannot use SUBSTR to go beyond the length of a character variable). In all cases, if you specify one or more invalid arguments, SAS will issue a message in the Log and the function will return a missing value. Suppose we specify this statement:

```
rounded = round(rate, rate_factor);
```

If RATE_FACTOR is a missing value or negative, ROUNDED will be set to missing, and the SAS Log will contain a message similar to:

```
NOTE: Argument 2 to function ROUND at line
1449 column 11 is invalid.
```

Missing Values Sometimes Matter. Missing values in one or more arguments may influence the value returned by the function. If we specify a missing value where we should have entered the starting location of a substring, then SAS will display an error message and the function will return a missing value.

Other functions are not as fussy. Most descriptive statistic functions – SUM, MEAN, RANGE, and the like – will operate on any non-missing arguments. This is an important distinction, since a simple assignment statement not using functions will create a missing value if *any* of its operands is missing. Examine the following code:

```
q1 = 300;  q2 = 350;  q3 = 250;  q4 = .;
year_tot_1 = q1 + q2 + q3 + q4;
year_tot_2 = sum(of q1-q4);
```

The first assignment statement has a form which requires all operands to be numeric and non-missing. Since Q4 is missing, the result, YEAR_TOT_1, will be missing. The second assignment uses the SUM function. Its parameters match the operands of the previous statement, but it returns a value because the function uses only non-missing values. YEAR_TOT_2 is 900. The impact of missing values is significant here and in other statistical functions.

Here, as in the points noted above, we emphasize the need to carefully review the function's documentation prior to writing the program.

Specify Character Variable Lengths. If the function is returning a character variable, specify the length of the variable to avoid unanticipated padding. In this example, variable QUOTED is length 200, regardless of the length of TEXT.

```
data revised;
set temp2;
```

```
quoted = quote(text);
run;
```

Adding a LENGTH statement to the program brings QUOTED to a more reasonable length (TEXT's length – assume $20 – plus two positions for quotes):

```
data revised;
set temp2;
length quoted $22;
quoted = quote(text);
run;
```

Parameter Specification Can Vary Greatly. Arguments to most functions can be constants, variables, or expressions (including other function calls). In general, the function will accept a value as long as the specification results in a value that is appropriate. Here are some examples. As before, look at them for style rather than exact meaning:

```
small_pair = min(min(c1,c2), min(d1,d2));
piece = substr(line, 1, length(line) - 3);
tot = sub + reg + sum(ot1, ot2, ot3);
```

The first two statements are examples of functions being used as arguments to other functions. This is commonly referred to as "nesting." The statements are concise, but bear in mind the difficulty of debugging them. What is the minimum of C1 and C2? Of D1 and D2? With the statement written as is, you can't tell. It may be easier in the long run to break up the statement:

```
min_c = min(c1, c2);
min_d = min(d1, d2);
small_pair = min(min_c, min_d);
```

Finally, remember that SAS will do its best to reduce each argument to its simplest form. This behavior is predictable, but can lead to unexpected results if only casually recalled. Consider this code fragment:

```
v1 = 3;  v2 = 4;  v3 = 10;  v4 = 6;
max_v = max(v1-v4);
```

Looking at the assignment statements, you would expect the value of MAX_V to be 10. Instead, it is –3. Why? Because SAS will resolve the argument before it is passed to the function. Instead of seeing a list – "V1 through V4" – SAS sees an arithmetic expression – "V1 minus V4." Thus only 3 minus 6, or –3, is passed to the MAX function and –3, by definition, is the maximum, since no other parameters are available for evaluation. Fortunately, MAX and other statistical functions have a way to specify V1-V4 as a list, rather than an expression. It is shown below and also noted in the last section's description of various functions:

```
max_v = max(of v1-v4);
```

Functions Can Be Used "On the Fly." The previous point's examples hinted at a powerful capability of using functions within SAS. Rather than store a function result in a variable, it's possible to make it transient, available only for purposes of evaluation and not for storage as a variable. Here we show the ability to use functions in decision-making statements.

```
if index(line, '.txt') > 0 then do;
   code, code, and more code

select (quarter(start_date));
  when (1) do;
   code, code, and more code
```

CALL Routines Are Close Cousins of Functions. A separate set of routines, called CALL routines (for obvious reasons that we'll soon see) are equivalent in spirit, if not syntax, to functions. The idea is the same as functions – create a value by passing a certain number of parameters of a certain data type in a certain order. The principal difference is in their invocation, as shown in the examples below:

```
call label(var_names(i), label_text);
if eof then call symput
   ('count', put(_nread, 3.));
```

Functions would return a transient, "on the fly" value or have their value stored in a variable. By contrast, CALL routines typically specify operands and results in the parameter list.

For the purposes of this paper, CALL routines and functions are treated identically. Their syntactical differences and distinctions are clearly highlighted in SAS documentation.

Commonly-Used Functions

A complete list of functions, grouped by category, is found in the Appendix. This section takes a closer look at some of the more commonly used functions and gives examples of their use. The order and category names correspond to the SAS Online Doc. Yet again – refer to SAS Institute documentation for a description of parameters and other usage notes.

Category/Name	Description and example of usage
array	
dim	Number of elements in an array. do i = 1 to dim(list);
character	
compbl	Removes consecutive blanks from a string. old = 'much extra space'; new = compbl(old); NEW becomes 'much extra space'

compress	Removes characters from a string.
	old = 'Chapel Hill, NC – 27516';
	new1 = compress(old);
	new2 = compress(old, ',-');
	NEW1 becomes 'ChapelHill,NC-27516'
	NEW2 becomes 'Chapel Hill NC 27516'
	You could use COMPBL on NEW2 to remove consecutive blanks.

index	Gives the starting position of a string within a string.
	string = '\temp\examples1.sas';
	loc1 = index(string, '.sas');
	loc2 = index(string, '.SAS');
	loc3 = index(upcase(string), '.SAS');
	LOC1 equals 1, LOC2 equals 0 (not found), LOC3 equals 16

left	Left-aligns a string old = ' leading blanks'; new = left(old); NEW equals 'leading blanks'
length	Returns the length (rightmost non-blank character) of a string. length old $40; old = 'Short'; len = length(old);

Category/Name	Description and example of usage
	LEN equals 5.
lowcase	Lower-cases a string. old = 'Mixed Case'; new = lowcase(old); NEW equals 'mixed case'
quote	Encloses a string in double quotes. old = 'WUNC'; /* Length of OLD = 10 */ length new $12; /* +2 to allow for quotes */ new = quote("WUNC"); NEW equals "WUNC " /* quotes are part of the value */
repeat	Repeats a string "n" times. old = 'dog'; new = repeat(old, 4); NEW equals 'dogdogdogdogdog' Remember to set a length for NEW!
reverse	Reverses a string. old = 'Looks OK'; new = reverse(old); NEW equals 'KO skooL'
right	Right justifies a string. old = 'Cats and Dogs'; /* OLD is $15 */ new = right(old); NEW equals ' Cats and Dogs'

scan	Scans a string for a character expression ('word') using a default or user-specified word.
	old = 'temp\filexxx.dat';
	new1 = scan(old, 1);
	new2 = scan(old, 2, '.\');
	new3 = scan(old, -1, '.\');
	NEW1 equals 'temp\filexxx'
	NEW2 equals 'filexxx'
	NEW3 equals 'dat'

substr	Extracts or replaces a portion of a string.
	old = 'Chapel Hill NC 27516';
	new = substr(old, length(old)-4);
	NEW equals '27516'
	old = 'J*V87';
	if substr(old, 2, 1) = '*' then do;

translate	Changes all occurrences of one character in a string to another. old = 'Line1*Line2*Line3'; new = translate(old, '/', '*'); NEW equals 'Line1/Line2/Line3'
tranwrd	Similar to TRANSLATE, but at the word level. Parameter order is different (from-to, rather than to-from!) old = 'Mrs. Smith'; new = tranwrd(old, 'Mrs.', 'Sra.'); NEW equals "Sra. Smith"
upcase	Upper-cases a string. old = 'Mixed Case'; new = upcase(old); NEW equals 'MIXED CASE'
date and time	
date	Returns the current date as a SAS date value today = date() TODAY is 15138 (June 12, 2001 if formatted)
datetime	Returns the current date-time as a SAS datetime value. rightnow =datetime(); RIGHTNOW is 1307985980.7 (12JUN2001:17:26:21 if formatted)
day / month / year	Extract the day, month, and year numbers from a SAS date value. curr_day = day(today()); CURR_DAY equals 12 if today is June 12, 2001.
hour / minute / second	Extracts the hour, minute, and second from a SAS time value. curr_hr = hour(onset); CURR_HR equals 7 if ONSET is 7:04.

Category/Name	Description and example of usage
intck	Returns the number of intervals between two dates, times, or date-times. qtr=intck('qtr','01jan2001'd,'01dec2001'd); QTR equals 3
intnx	Advances a date, time, or date-time by a specified interval. yr=intnx('year','15mar99'd,2); YR equals 14976 (01JAN01 if formatted)
mdy	Creates a SAS date value from user-specified month, day, and year. sas_date = mdy(1, 1, 2001); SAS_DATE is 14976

time /
today
	Return the current time and date. Note that these functions do not require parameters. The parentheses are necessary for SAS to make the distinction between the function call to DATE and TIME and variables of the same name.
	curr_time = time();
	curr_date = date();
	CURR_TIME is 63666.44 (17:41 if formatted)
	CURR_DATE is 15138 (June 12, 2001 if formatted)

descriptive statistics	

all functions	See Appendix A for details.
	The function names correspond to statistics available in the MEANS procedure.
	General form of usage is:
	function_name(arg1, arg2, …)
	function_name(OF base1-baseN)
	For example:
	tot = sum(ne, se, nw, sc, pc, mw);
	tot = sum(of r1-r5);
	tot = sum(of r1-r5, intl);

macro	
symput	CALL routine if eof then call symput('goodones', put(_n,3.)); Macro variable GOODONES contains the value of variable _N.
symget	Retrieves the value of a macro variable. status = symget('stat'); DATA step variable STATUS equals the value of macro variable STAT.
math	

abs	Returns the absolute value of a numeric variable.
	old = -3;
	new = abs(old);
	NEW equals 3

fact	Returns the factorial of an integer. fact = fact(5); FACT equals 120
log / log10 / log2	Return the natural, base 10, and base 2 logarithms of a positive number.

mod	Returns the remainder of a division.
	old = 100;
	new1 = mod(old, 10);
	new2 = mod(old, 8);
	NEW1 equals 0 (no remainder)
	NEW2 equals 4 (4 left over when 100 is divided by 8)

random number	
rannor / ranuni	As CALL routines, they return random variates from normal and uniform distributions. call ranuni(-1);
rannor / ranuni	As functions, they return random variates from normal and uniform distributions. The CALL routines greater control over seed values. call rannor(-1);
special	
system	CALL routine, it submits a host operating system command for execution.

Category/Name	Description and example of usage
	call system("dir p:\qc\meas*.txt /s > c:\temp\dir.txt"); Creates text file c:\temp\dir.txt, which contains the output from a directory command.
input	Allows a character variable to be read using standard SAS informats. This is a way to convert character values to numeric. char_date = '2001/08/19'; num_date = input(char_date, yymmdd10.); NUM_DATE has a numeric data type, with a value of 15206.
put	The reverse of INPUT, it writes formatted character or numeric variables to a target character variable. sales_c = put(sales, dollar8.) || ' - ' || put(sales, salefmt.); User-written format SALEFMT might result in a SALES_C value such as "120,300 – Time to buy a Lexus!"
system	Issues an operating system command and captures its return code. rc = system('cd t:\prod\rpts\graphs');
state and ZIP code	

FIPNAME / FIPSTATE /	Converts FIPS codes to state names and postal codes.
	fip_state = 37;
	name = fipname(fip_state);
	postal = fipstate(fip_state);
	NAME equals "NORTH CAROLINA"
	POSTAL equals "NC"

STFIPS / STNAME	Converts state postal codes to FIPS codes and state names.
	postal = 'NC';
	name = stname(postal);
	postal = stfips(postal);
	NAME equals "NORTH CAROLINA"
	POSTAL equals "NC"

ZIPFIPS / ZIPNAME / ZIPSTATE	Converts ZIP codes to FIPS codes, state names, and state postal codes.
	zip = '27516';
	name = zipname(zip);
	postal = zipfips(zip);
	fips = zipfips(zip);
	NAME equals "NORTH CAROLINA"
	POSTAL equals "NC"
	FIPS = 37;

truncation	

ceil	Rounds up to the nearest integer.
	old = 2.3;
	new = ceil(old);
	NEW equals 3.

floor	Rounds down to the nearest integer.
	old = 2.3;
	new = ceil(old);
	NEW equals 2.

int	Returns the integer portion of a number.
	old = -8.9;
	new = int(old);
	NEW equals –8.

round	Rounds to the nearest rounding unit (default rounding unit is 1).
	old = 100.53;
	new1 = round(old);
	new2 = round(old, .1);
	NEW1 equals 101
	NEW2 equals 100.5

variable control	
label	CALL routine, it returns the value of a variable's label. label old = "Old's nondescript label"; call label(old, label_value); LABEL_VALUE equals "Old's nondescript label"

vname	CALL routine, it returns the name of a variable.
	length name $32;
	array temp(*) st-block;
	do i = 1 to dim(temp);
	call vname(temp(i), name);
	put name=;
	end;
	This code writes the name of each element in TEMP.

Questions? Comments?

Your feedback is always welcome. Contact the author at fcd1@mindspring.com.

Appendix A: Functions and CALL Routines by Category

The table in this appendix is taken directly from the Version 8.0 SAS Online Doc. It gives an idea of the power and versatility of the functions and CALL routines that come with the SAS System. For details, of course, refer to the specific help file or other SAS documentation.

Array	
DIM	Returns the number of elements in an array
HBOUND	Returns the upper bound of an array
LBOUND	Returns the lower bound of an array

Bitwise Logical Operations	
BAND	Returns the bitwise logical AND of two arguments
BLSHIFT	Returns the bitwise logical left shift of two arguments
BNOT	Returns the bitwise logical NOT of an argument
BOR	Returns the bitwise logical OR of two arguments
BRSHIFT	Returns the bitwise logical right shift of two arguments
BXOR	Returns the bitwise logical EXCLUSIVE OR of two arguments

Character String Matching	
CALL RXCHANGE	Changes one or more substrings that match a pattern
CALL RXFREE	Frees memory allocated by other regular expression (RX) functions and CALL routines
CALL RXSUBSTR	Finds the position, length, and score of a substring that matches a pattern
RXMATCH	Finds the beginning of a substring that matches a pattern and returns a value
RXPARSE	Parses a pattern and returns a value

Character	
BYTE	Returns one character in the ASCII or the EBCDIC collating sequence
COLLATE	Returns an ASCII or EBCDIC collating sequence character string
COMPBL	Removes multiple blanks from a character string
COMPRESS	Removes specific characters from a character string
DEQUOTE	Removes quotation marks from a character value
INDEX	Searches a character expression for a string of characters
INDEXC	Searches a character expression for specific characters
INDEXW	Searches a character expression for a specified string as a word
LEFT	Left aligns a SAS character expression
LENGTH	Returns the length of an argument
LOWCASE	Converts all letters in an argument to lowercase
MISSING	Returns a numeric result that indicates whether the argument contains a missing value
QUOTE	Adds double quotation marks to a character value
RANK	Returns the position of a character in the ASCII or EBCDIC collating sequence
REPEAT	Repeats a character expression
REVERSE	Reverses a character expression
RIGHT	Right aligns a character expression
SCAN	Selects a given word from a character expression
SOUNDEX	Encodes a string to facilitate searching
SPEDIS	Determines the likelihood of two words matching, expressed as the asymmetric spelling distance between the two words
SUBSTR (left of =)	Replaces character value contents
SUBSTR	Extracts a substring from an argument
TRANSLATE	Replaces specific characters in a character expression

Character

TRANWRD	Replaces or removes all occurrences of a word in a character string
TRIM	Removes trailing blanks from character expressions and returns one blank if the expression is missing
TRIMN	Removes trailing blanks from character expressions and returns a null string (zero blanks) if the expression is missing
UPCASE	Converts all letters in an argument to uppercase
VERIFY	Returns the position of the first character that is unique to an expression

Double-Byte Character Set (DBCS)

KCOMPARE	Returns the result of a comparison of character strings
KCOMPRESS	Removes specific characters from a character string
KCOUNT	Returns the number of double-byte characters in a string
KINDEX	Searches a character expression for a string of characters
KINDEXC	Searches a character expression for specific characters
KLEFT	Left aligns a SAS character expression by removing unnecessary leading DBCS blanks and SO/SI
KLENGTH	Returns the length of an argument
KLOWCASE	Converts all letters in an argument to lowercase
KREVERSE	Reverses a character expression
KRIGHT	Right aligns a character expression by trimming trailing DBCS blanks and SO/SI
KSCAN	Selects a given word from a character expression
KSTRCAT	Concatenates two or more character strings
KSUBSTR	Extracts a substring from an argument
KSUBSTRB	Extracts a substring from an argument based on byte position
KTRANSLATE	Replaces specific characters in a character expression
KTRIM	Removes trailing DBCS blanks and SO/SI from character expressions
KTRUNCATE	Truncates a numeric value to a specified length
KUPCASE	Converts all single-byte letters in an argument to uppercase
KUPDATE	Inserts, deletes, and replaces character value contents
KUPDATEB	Inserts, deletes, and replaces character value contents based on byte unit
KVERIFY	Returns the position of the first character that is unique to an expression

Date and Time

DATDIF	Returns the number of days between two dates
DATE	Returns the current date as a SAS date value
DATEJUL	Converts a Julian date to a SAS date value
DATEPART	Extracts the date from a SAS datetime value
DATETIME	Returns the current date and time of day as a SAS datetime value
DAY	Returns the day of the month from a SAS date value
DHMS	Returns a SAS datetime value from date, hour, minute, and second
HMS	Returns a SAS time value from hour, minute, and second values
HOUR	Returns the hour from a SAS time or datetime value
INTCK	Returns the integer number of time intervals in a given time span
INTNX	Advances a date, time, or datetime value by a given interval, and returns a date, time, or datetime value
JULDATE	Returns the Julian date from a SAS date value
JULDATE7	Returns a seven-digit Julian date from a SAS date value
MDY	Returns a SAS date value from month, day, and year values
MINUTE	Returns the minute from a SAS time or datetime value
MONTH	Returns the month from a SAS date value
QTR	Returns the quarter of the year from a SAS date value
SECOND	Returns the second from a SAS time or datetime value
TIME	Returns the current time of day
TIMEPART	Extracts a time value from a SAS datetime value
TODAY	Returns the current date as a SAS date value
WEEKDAY	Returns the day of the week from a SAS date value
YEAR	Returns the year from a SAS date value
YRDIF	Returns the difference in years between two dates
YYQ	Returns a SAS date value from the year and quarter

Descriptive Statistics

CSS	Returns the corrected sum of squares
CV	Returns the coefficient of variation
KURTOSIS	Returns the kurtosis
MAX	Returns the largest value
MEAN	Returns the arithmetic mean (average)
MIN	Returns the smallest value
MISSING	Returns a numeric result that indicates whether the argument contains a missing value
N	Returns the number of nonmissing values
NMISS	Returns the number of missing values
ORDINAL	Returns any specified order statistic
RANGE	Returns the range of values
SKEWNESS	Returns the skewness
STD	Returns the standard deviation
STDERR	Returns the standard error of the mean
SUM	Returns the sum of the nonmissing arguments
USS	Returns the uncorrected sum of squares
VAR	Returns the variance

External Files

DCLOSE	Closes a directory that was opened by the DOPEN function and returns a value
DINFO	Returns information about a directory

External Files

DNUM	Returns the number of members in a directory
DOPEN	Opens a directory and returns a directory identifier value
DOPTNAME	Returns directory attribute information
DOPTNUM	Returns the number of information items that are available for a directory
DREAD	Returns the name of a directory member
DROPNOTE	Deletes a note marker from a SAS data set or an external file and returns a value
FAPPEND	Appends the current record to the end of an external file and returns a value
FCLOSE	Closes an external file, directory, or directory member, and returns a value
FCOL	Returns the current column position in the File Data Buffer (FDB)
FDELETE	Deletes an external file or an empty directory
FEXIST	Verifies the existence of an external file associated with a fileref and returns a value
FGET	Copies data from the File Data Buffer (FDB) into a variable and returns a value
FILEEXIST	Verifies the existence of an external file by its physical name and returns a value
FILENAME	Assigns or deassigns a fileref for an external file, directory, or output device and returns a value
FILEREF	Verifies that a fileref has been assigned for the current SAS session and returns a value
FINFO	Returns the value of a file information item
FNOTE	Identifies the last record that was read and returns a value that FPOINT can use
FOPEN	Opens an external file and returns a file identifier value
FOPTNAME	Returns the name of an item of information about a file
FOPTNUM	Returns the number of information items that are available for an external file
FPOINT	Positions the read pointer on the next record to be read and returns a value
FPOS	Sets the position of the column pointer in the File Data Buffer (FDB) and returns a value
FPUT	Moves data to the File Data Buffer (FDB) of an external file, starting at the FDB's current column position, and returns a value
FREAD	Reads a record from an external file into the File Data Buffer (FDB) and returns a value
FREWIND	Positions the file pointer to the start of the file and returns a value
FRLEN	Returns the size of the last record read, or, if the file is opened for output, returns the current record size
FSEP	Sets the token delimiters for the FGET function and returns a value
FWRITE	Writes a record to an external file and returns a value
MOPEN	Opens a file by directory id and member name, and returns the file identifier or a 0
PATHNAME	Returns the physical name of a SAS data library or of an external file, or returns a blank
SYSMSG	Returns the text of error messages or warning messages from the last data set or external file function execution
SYSRC	Returns a system error number

External Routines

CALL MODULE	Calls the external routine without any return code
CALL MODULEI	Calls the external routine without any return code (in IML environment only)
MODULEC	Calls an external routine and returns a character value
MODULEIC	Calls an external routine and returns a character value (in IML environment only)
MODULEIN	Calls an external routine and returns a numeric value (in IML environment only)
MODULEN	Calls an external routine and returns a numeric value

Financial

COMPOUND	Returns compound interest parameters
CONVX	Returns the convexity for an enumerated cashflow
CONVXP	Returns the convexity for a periodic cashflow stream, such as a bond
DACCDB	Returns the accumulated declining balance depreciation
DACCDBSL	Returns the accumulated declining balance with conversion to a straight-line depreciation
DACCSL	Returns the accumulated straight-line depreciation
DACCSYD	Returns the accumulated sum-of-years-digits depreciation
DACCTAB	Returns the accumulated depreciation from specified tables
DEPDB	Returns the declining balance depreciation
DEPDBSL	Returns the declining balance with conversion to a straight-line depreciation
DEPSL	Returns the straight-line depreciation
DEPSYD	Returns the sum-of-years-digits depreciation
DEPTAB	Returns the depreciation from specified tables
DUR	Returns the modified duration for an enumerated cashflow
DURP	Returns the modified duration for a periodic cashflow stream, such as a bond
INTRR	Returns the internal rate of return as a fraction
IRR	Returns the internal rate of return as a percentage
MORT	Returns amortization parameters
NETPV	Returns the net present value as a fraction
NPV	Returns the net present value with the rate expressed as a percentage
PVP	Returns the present value for a periodic cashflow stream, such as a bond
SAVING	Returns the future value of a periodic saving
YIELDP	Returns the yield-to-maturity for a periodic cashflow stream, such as a bond

Hyperbolic

COSH	Returns the hyperbolic cosine
SINH	Returns the hyperbolic sine
TANH	Returns the hyperbolic tangent

Macro

CALL EXECUTE	Resolves an argument and issues the resolved value for execution
CALL SYMPUT	Assigns DATA step information to a macro variable
RESOLVE	Returns the resolved value of an argument after it has been processed by the macro facility
SYMGET	Returns the value of a macro variable during DATA step execution

Mathematical

ABS	Returns the absolute value
AIRY	Returns the value of the airy function
CNONCT	Returns the noncentrality parameter from a chi-squared distribution
COMB	Computes the number of combinations of **n** elements taken **r** at a time and returns a value
CONSTANT	Computes some machine and mathematical constants and returns a value
DAIRY	Returns the derivative of the airy function
DEVIANCE	Computes the deviance and returns a value
DIGAMMA	Returns the value of the DIGAMMA function
ERF	Returns the value of the (normal) error function
ERFC	Returns the value of the complementary (normal) error function
EXP	Returns the value of the exponential function
FACT	Computes a factorial and returns a value
FNONCT	Returns the value of the noncentrality parameter of an F distribution
GAMMA	Returns the value of the Gamma function
IBESSEL	Returns the value of the modified □essel function
JBESSEL	Returns the value of the □essel function
LGAMMA	Returns the natural logarithm of the Gamma function
LOG	Returns the natural (base e) logarithm
LOG10	Returns the logarithm to the base 10
LOG2	Returns the logarithm to the base 2
MOD	Returns the remainder value
PERM	Computes the number of permutations of **n** items taken **r** at a time and returns a value
SIGN	Returns the sign of a value
SQRT	Returns the square root of a value
TNONCT	Returns the value of the noncentrality parameter from the student's **t** distribution
TRIGAMMA	Returns the value of the TRIGAMMA function

Probability

CDF	Computes cumulative distribution functions
LOGPDF	Computes the logarithm of a probability (mass) function
LOGSDF	Computes the logarithm of a survival function
PDF	Computes probability density (mass) functions
POISSON	Returns the probability from a Poisson distribution
PROBBETA	Returns the probability from a beta distribution
PROBBNML	Returns the probability from a binomial distribution
PROBBNRM	Computes a probability from the bivariate normal distribution and returns a value
PROBCHI	Returns the probability from a chi-squared distribution
PROBF	Returns the probability from an **F** distribution
PROBGAM	Returns the probability from a gamma distribution
PROBHYPR	Returns the probability from a hypergeometric distribution
PROBMC	Computes a probability or a quantile from various distributions for multiple comparisons of means, and returns a value
PROBNEGB	Returns the probability from a negative binomial distribution
PROBNORM	Returns the probability from the standard normal distribution
PROBT	Returns the probability from a **t** distribution
SDF	Computes a survival function

Quantile

BETAINV	Returns a quantile from the beta distribution
CINV	Returns a quantile from the chi-squared distribution
FINV	Returns a quantile from the **F** distribution
GAMINV	Returns a quantile from the gamma distribution
PROBIT	Returns a quantile from the standard normal distribution
TINV	Returns a quantile from the **t** distribution

Random Number

CALL RANBIN	Returns a random variate from a binomial distribution
CALL RANCAU	Returns a random variate from a Cauchy distribution
CALL RANEXP	Returns a random variate from an exponential distribution
CALL RANGAM	Returns a random variate from a gamma distribution
CALL RANNOR	Returns a random variate from a normal distribution
CALL RANPOI	Returns a random variate from a Poisson distribution
CALL RANTBL	Returns a random variate from a tabled probability distribution
CALL RANTRI	Returns a random variate from a triangular distribution
CALL RANUNI	Returns a random variate from a uniform distribution
NORMAL	Returns a random variate from a normal distribution
RANBIN	Returns a random variate from a binomial distribution
RANCAU	Returns a random variate from a Cauchy distribution

Random Number

RANEXP	Returns a random variate from an exponential distribution
RANGAM	Returns a random variate from a gamma distribution
RANNOR	Returns a random variate from a normal distribution
RANPOI	Returns a random variate from a Poisson distribution
RANTBL	Returns a random variate from a tabled probability
RANTRI	Random variate from a triangular distribution
RANUNI	Returns a random variate from a uniform distribution
UNIFORM	Random variate from a uniform distribution

SAS File I/O

ATTRC	Returns the value of a character attribute for a SAS data set
ATTRN	Returns the value of a numeric attribute for the specified SAS data set
CEXIST	Verifies the existence of a SAS catalog or SAS catalog entry and returns a value
CLOSE	Closes a SAS data set and returns a value
CUROBS	Returns the observation number of the current observation
DROPNOTE	Deletes a note marker from a SAS data set or an external file and returns a value
DSNAME	Returns the SAS data set name that is associated with a data set identifier
EXIST	Verifies the existence of a SAS data library member
FETCH	Reads the next nondeleted observation from a SAS data set into the Data Set Data Vector (DDV) and returns a value
FETCHOBS	Reads a specified observation from a SAS data set into the Data Set Data Vector (DDV) and returns a value
GETVARC	Returns the value of a SAS data set character variable
GETVARN	Returns the value of a SAS data set numeric variable
IORCMSG	Returns a formatted error message for _IORC_
LIBNAME	Assigns or deassigns a libref for a SAS data library and returns a value
LIBREF	Verifies that a libref has been assigned and returns a value
NOTE	Returns an observation ID for the current observation of a SAS data set
OPEN	Opens a SAS data set and returns a value
PATHNAME	Returns the physical name of a SAS data library or of an external file, or returns a blank
POINT	Locates an observation identified by the NOTE function and returns a value
REWIND	Positions the data set pointer at the beginning of a SAS data set and returns a value
SYSMSG	Returns the text of error messages or warning messages from the last data set or external file function execution
SYSRC	Returns a system error number
VARFMT	Returns the format assigned to a SAS data set variable
VARINFMT	Returns the informat assigned to a SAS data set variable
VARLABEL	Returns the label assigned to a SAS data set variable
VARLEN	Returns the length of a SAS data set variable
VARNAME	Returns the name of a SAS data set variable
VARNUM	Returns the number of a variable's position in a SAS data set
VARTYPE	Returns the data type of a SAS data set variable

Special

ADDR	Returns the memory address of a variable
CALL POKE	Writes a value directly into memory
CALL SYSTEM	Submits an operating environment command for execution
DIF	Returns differences between the argument and its **n**th lag
GETOPTION	Returns the value of a SAS system or graphics option
INPUT	Returns the value produced when a SAS expression that uses a specified informat expression is read
INPUTC	Enables you to specify a character informat at run time
INPUTN	Enables you to specify a numeric informat at run time
LAG	Returns values from a queue
PEEK	Stores the contents of a memory address into a numeric variable
PEEKC	Stores the contents of a memory address into a character variable
POKE	Writes a value directly into memory
PUT	Returns a value using a specified format
PUTC	Enables you to specify a character format at run time
PUTN	Enables you to specify a numeric format at run time
SYSGET	Returns the value of the specified operating environment variable
SYSPARM	Returns the system parameter string
SYSPROD	Determines if a product is licensed
SYSTEM	Issues an operating environment command during a SAS session

State Postal, FIPS, and ZIP Codes

FIPNAME	Converts FIPS codes to uppercase state names
FIPNAMEL	Converts FIPS codes to mixed case state names
FIPSTATE	Converts FIPS codes to two-character postal codes
STFIPS	Converts state postal codes to FIPS state codes
STNAME	Converts state postal codes to uppercase state names
STNAMEL	Converts state postal codes to mixed case state names
ZIPFIPS	Converts ZIP codes to FIPS state codes
ZIPNAME	Converts ZIP codes to uppercase state names
ZIPNAMEL	Converts ZIP codes to mixed case state names
ZIPSTATE	Converts ZIP codes to state postal codes

Trigonometric

ARCOS	Returns the arccosine
ARSIN	Returns the arcsine

Trigonometric	
ATAN	Returns the arctangent
COS	Returns the cosine
SIN	Returns the sine
TAN	Returns the tangent

Truncation	
CEIL	Returns the smallest integer that is greater than or equal to the argument
FLOOR	Returns the largest integer that is less than or equal to the argument
FUZZ	Returns the nearest integer if the argument is within 1E-12
INT	Returns the integer value
ROUND	Rounds to the nearest round-off unit
TRUNC	Truncates a numeric value to a specified length

Variable Control	
CALL LABEL	Assigns a variable label to a specified character variable
CALL SET	Links SAS data set variables to DATA step or macro variables that have the same name and data type
CALL VNAME	Assigns a variable name as the value of a specified variable

Variable Information	
VARRAY	Returns a value that indicates whether the specified name is an array
VARRAYX	Returns a value that indicates whether the value of the specified argument is an array
VFORMAT	Returns the format that is associated with the specified variable
VFORMATD	Returns the format decimal value that is associated with the specified variable
VFORMATDX	Returns the format decimal value that is associated with the value of the specified argument
VFORMATN	Returns the format name that is associated with the specified variable
VFORMATNX	Returns the format name that is associated with the value of the specified argument
VFORMATW	Returns the format width that is associated with the specified variable
VFORMATWX	Returns the format width that is associated with the value of the specified argument
VFORMATX	Returns the format that is associated with the value of the specified argument
VINARRAY	Returns a value that indicates whether the specified variable is a member of an array
VINARRAYX	Returns a value that indicates whether the value of the specified argument is a member of an array
VINFORMAT	Returns the informat that is associated with the specified variable
VINFORMATD	Returns the informat decimal value that is associated with the specified variable
VINFORMATDX	Returns the informat decimal value that is associated with the value of the specified argument
VINFORMATN	Returns the informat name that is associated with the specified variable
VINFORMATNX	Returns the informat name that is associated with the value of the specified argument
VINFORMATW	Returns the informat width that is associated with the specified variable
VINFORMATWX	Returns the informat width that is associated with the value of the specified argument
VINFORMATX	Returns the informat that is associated with the value of the specified argument
VLABEL	Returns the label that is associated with the specified variable
VLABELX	Returns the variable label for the value of a specified argument
VLENGTH	Returns the compile-time (allocated) size of the specified variable
VLENGTHX	Returns the compile-time (allocated) size for the value of the specified argument
VNAME	Returns the name of the specified variable
VNAMEX	Validates the value of the specified argument as a variable name
VTYPE	Returns the type (character or numeric) of the specified variable
VTYPEX	Returns the type (character or numeric) for the value of the specified argument

Web Tools	
HTMLDECODE	Decodes a string containing HTML numeric character references or HTML character entity references and returns the decoded string
HTMLENCODE	Encodes characters using HTML character entity references and returns the encoded string
URLDECODE	Returns a string that was decoded using the URL escape syntax
URLENCODE	Returns a string that was encoded using the URL escape syntax

Basic SAS PROCedures for Generating Quick Results

Kirk Paul Lafler, Software Intelligence Corporation

Abstract

As an IT professional, saving time is critical. So is delivering timely and quality looking reports to management, end users, and customers. The SAS System provides numerous *"canned"* PROCedures for generating quick results to take care of both... and more. Attendees will see how basic SAS PROCedures such as SORT, PRINT, SQL, and FORMS are used to create detail reports; how CHART, FREQ, MEANS, PLOT, and UNIVARIATE are used to summarize and create graphical, tabular, or statistical output; and several useful techniques including how to inform the SAS System which data set to use as input to a procedure, how to subset data using a WHERE statement (or WHERE= clause), and how to perform BY-group processing to separate data into groups of like information.

Introduction

Once data has been collected and stored in a SAS data set, results can be produced quickly using one or more procedures. The SAS System provides numerous ready-to-use procedures designed for data analysis and presentation. Procedures are designed to be simple to use, and are what differentiate SAS from other software products. SAS' built-in procedures offer users with a unique ability to generate quick results – requiring little, if any, programming skills. Using a procedure, or PROC, is similar to filling out a simple request form. By specifying the name of procedure and one or more options, you can produce results quickly and easily.

Procedures frequently write their results to the Output window (in SAS Display Manager), an output SAS data set, or an output file. When output is produced, it is often customized so it satisfies certain requirements such as automatic centering, printing or displaying dates and page numbers, and so on. Having the ability to customize the way output appears as well as the type of information that is produced is what makes procedures an indispensable tool for users everywhere.

SAS supports four categories of procedures: 1) reporting, 2) statistical, 3) scoring, and 4) utility. This paper investigates the use of several base-SAS procedures to enable the production of quick and useful reports, statistics, and tables of data, and will also look at procedures that can be used to perform simple data set management tasks.

The Anatomy of a PROC

Each procedure (or PROC) has unique characteristics and elements, but many common ones too. Each PROC consists of a keyword, one or more statements, and options – of which some are required and others are basically – optional. Although the statements and options vary from PROC to PROC, the basic anatomy of a PROC looks something like the following:

```
PROC procname DATA= _____________________;
  TITLE ___________________________________;
  FOOTNOTE ________________________________;
  BY ______________________________________;
  LABEL ___________________________________;
  FORMAT __________________________________;
RUN; <or> QUIT;
```

The PROC Statement

Every PROC begins with the keyword PROC and is a required element. This keyword signals to the SAS supervisor (the internal traffic cop that controls everything that goes on in the SAS System) that a *"canned"* procedure is being launched, and not a DATA step or MACRO program.

The name of the PROC follows the keyword PROC – our example above references the anatomy PROC. In its simplest and purest form this is all that is required by a few procedures to run. As you might expect though, the results may also tend to be basic and output will appear without any customizations.

Once the name of the PROC is specified, you may then specify one or more options available with the PROC, and in any order. In our basic skeletal example above, the DATA= option appears. This option, if specified, informs the SAS System what data set to use as input to the PROC. If omitted, it automatically defaults to the most recently created data set – which may not be the most data set most recently used. (Readers will see several PROCs and their various options on the following pages).

TITLE and FOOTNOTE Statements

TITLE and FOOTNOTE statements are considered to be global statements and can generally be used universally throughout the SAS System (e.g., PROCs and DATA steps). A maximum of ten TITLE and FOOTNOTE statements can be specified in any PROC. TITLE statements, when specified, appear at the top of each output page and FOOTNOTE statements, if present, produce output at the bottom of each page of output. Readers are cautioned to use care when specifying TITLE and FOOTNOTE

statements since they reduce the available space for printing detail lines.

BY Statement
A BY statement is optional in all PROCs except the SORT procedure. A BY statement in PROC SORT tells SAS what the order or arrangement should be for observations in a data set. A BY statement in any other PROC informs SAS to perform a separate analysis on the values in each BY group opposed to one large group. The data must have been sorted before it can be used in a reporting procedure.

LABEL Statement
A LABEL statement is also optional, and if present allows a more descriptive label to be assigned as variable (or column) headings. If omitted, SAS uses the variable names as column headers on output. When a LABEL statement is used in a PROC the assigned descriptive labels are only available for duration of the PROC step, and are not saved with the data set.

FORMAT Statement
A FORMAT statement is an optional statement that, when used, tells SAS to display information on output in a designated way. For example, you could have a date value displayed or written using a *mm/dd/yyyy* form such as 08/20/2001 or a *Month dd, yyyy* form such as August 20, 2001 to enhance readability. In the absence of a FORMAT statement, data is displayed using a internal date offset (the number of days from January 1, 1960) or a user-defined date format stored as part of the data set.

RUN or QUIT Statement
A RUN or QUIT statement tells SAS to terminate the PROC step before executing the next step in a program. A RUN statement is normally specified to designate an end to a non-interactive procedure like PROC PRINT, whereas a QUIT statement is specified to terminate an interactive procedure such as PROC SQL (more will be said about interactive procedures later). Although not required statements, specifying a RUN or QUIT statement can provide modest CPU improvements since the SAS Supervisor knows when one step ends and another begins.

A Print-Oriented World
Information is the lifeblood of virtually every organization in today's print-oriented world. And it appears that this need for even more information by decision makers is growing, according to Terence Mullin of Quest Software, a software services company located in Irvine, California. In the February 2001 Enterprise Systems Journal issue, "The Paper Chase", Mullin claims that "***Industry experts predict that digitally printed information will continue to grow 10 percent, annually, through 2004***". This translates into a huge and growing need for an organization's intellectual property.

Often an organization's intellectual property is collected and stored in multiple locations and not in a centralized information repository. With much of this information kept in scattered locations, from employees' PCs to network file servers and databases, information is not always available for a controlled reporting process. Although this can present huge problems for an organization's decision makers, it is sadly a fact of life.

Description of Data Used in Reports
The examples in this paper reference a data set containing a compilation of popular movies that I have watched over the years. This Movies data set consists of 22 observations, six variables, and contains the following data sorted in ascending order by the Movie Rating.

	Title	Length	Category	Year	Studio	Rating
1	The Wizard of Oz	101	Adventure	1939	MGM / UA	G
2	Casablanca	103	Drama	1942	MGM / UA	PG
3	Jaws	125	Action Adventure	1975	Universal Studios	PG
4	Rocky	120	Action Adventure	1976	MGM / UA	PG
5	Star Wars	124	Action Sci-Fi	1977	Lucas Film Ltd	PG
6	Poltergeist	115	Horror	1982	MGM / UA	PG
7	The Hunt for Red October	135	Action Adventure	1989	Paramount Pictures	PG
8	National Lampoon's Vacation	98	Comedy	1983	Warner Brothers	PG-13
9	Christmas Vacation	97	Comedy	1989	Warner Brothers	PG-13
10	Ghost	127	Drama Romance	1990	Paramount Pictures	PG-13
11	Jurassic Park	127	Action	1993	Universal Pictures	PG-13
12	Forrest Gump	142	Drama	1994	Paramount Pictures	PG-13
13	Michael	106	Drama	1997	Warner Brothers	PG-13
14	Titanic	194	Drama Romance	1997	Paramount Pictures	PG-13
15	Dressed to Kill	105	Drama Mysteries	1980	Filmways Pictures	R
16	Scarface	170	Action Cops & Robber	1983	Universal Studios	R
17	The Terminator	108	Action Sci-Fi	1984	Live Entertainment	R
18	Lethal Weapon	110	Action Cops & Robber	1987	Warner Brothers	R
19	Coming to America	116	Comedy	1988	Paramount Pictures	R
20	Silence of the Lambs	118	Drama Suspense	1991	Orion	R
21	Dracula	130	Horror	1993	Columbia TriStar	R
22	Brave Heart	177	Action Adventure	1995	Paramount Pictures	R

SAS Reporting Procedures
So you may be wondering how the SAS System and its many PROCs can help with your reporting needs. The answer is simple. SAS and its many "canned, ready-to-go" PROCs turn information into structured and meaningful reports. More than 3.5 million software customers in 115 countries, and 98% of the Fortune 100 companies and 90% of the Fortune 500 companies can attest to the fact that producing detailed reports with the SAS System has never been easier.

SAS and its many 'canned, ready-to-go"PROCs turn information into structured and meaningful reports.

Being able to put a powerful reporting tool in the hands of so many users, and not only programmers, is also an asset – not a liability. Having said this – it remains critical that an organization maintain a controlled environment to ensure information being dispensed in any report does not violate privacy and

security issues. It is also critical to maintain a high level of accuracy in all reported information.

SAS reporting PROCs consist of a broad range of easy-to-use report formats. Because output is frequently requested in a variety of formats to satisfy a vast number of requirements, SAS is ready and able to help by bundling base-SAS PROCs that produce reports in the following formats:

1) **Detail** – prints one or more observations without collapsing data that meet specific report criteria.
2) **Summary** – collapses and prints information.
3) **Tabular** – collapses and prints information with borders.
4) **Statistical** – computes and prints descriptive statistics.
5) **Graphical** – prints information as simple line-oriented bar and pie charts, and line plots.

An alphabetical list of several SAS reporting PROCs and their output formats is illustrated in table 1 below.

Report Output Style	PROC
Detail Output	FORMS
	PRINT
	REPORT
	SQL
Summary Output	CHART
	FREQ
	MEANS
	PLOT
	SQL
	UNIVARIATE
Tabular Output	FREQ
	TABULATE
Statistical Output	MEANS
	SQL
	UNIVARIATE
Graphical Output	CHART
	PLOT

Table 1. PROCs and Report Formats

PROCs for All That Detail

Sometimes a report must show all the detail it can. When this is the case, SAS provides the PRINT, REPORT, and SQL procedures to generate detail reports and the FORMS procedure to produce repetitive forms and labels. Although each PROC is relatively easy to use, they can also provide the level of support needed by even the most demanding programmer. By using one or more statements and options each PROC can produce simple to semi-custom reports.

Using PROC PRINT

The PRINT procedure is a popular reporting tool that is used by users everywhere. In its simplest form, PROC PRINT prints all variables for all observations in a data set. The SAS System writes a default title line at the top of each report page automatically.

Suppose you had to create a report containing all Movie data (all observations and variables) with the littlest amount of code possible. The PROC PRINT statement illustrated below is about as simple as it gets – would produce a detail-oriented report consisting of all observations and variables.

Procedure Code:

```
PROC PRINT DATA=SSU.MOVIES;
RUN;
```

When the PROC PRINT code is executed, the SAS System applies certain defaults in creating report output including the default title, the number of observations, and list of variables.

Results:

The SAS System

Obs	Title	Length	Category	Year	Studio	Rating
1	The Wizard of Oz	101	Adventure	1939	MGM / UA	G
2	Casablanca	103	Drama	1942	MGM / UA	PG
3	Jaws	125	Action Adventure	1975	Universal Studios	PG
4	Rocky	120	Action Adventure	1976	MGM / UA	PG
5	Star Wars	124	Action Sci-Fi	1977	Lucas Film Ltd	PG
6	Poltergeist	115	Horror	1982	MGM / UA	PG
7	The Hunt for Red October	135	Action Adventure	1989	Paramount Pictures	PG
8	National Lampoon's Vacation	98	Comedy	1983	Warner Brothers	PG-13
9	Christmas Vacation	97	Comedy	1989	Warner Brothers	PG-13
10	Ghost	127	Drama Romance	1990	Paramount Pictures	PG-13
11	Jurassic Park	127	Action	1993	Universal Pictures	PG-13
12	Forrest Gump	142	Drama	1994	Paramount Pictures	PG-13
13	Michael	106	Drama	1997	Warner Brothers	PG-13
14	Titanic	194	Drama Romance	1997	Paramount Pictures	PG-13
15	Dressed to Kill	105	Drama Mysteries	1980	Filmways Pictures	R
16	Scarface	170	Action Cops & Robber	1983	Universal Studios	R
17	The Terminator	108	Action Sci-Fi	1984	Live Entertainment	R
18	Lethal Weapon	110	Action Cops & Robber	1987	Warner Brothers	R
19	Coming to America	116	Comedy	1988	Paramount Pictures	R
20	Silence of the Lambs	118	Drama Suspense	1991	Orion	R
21	Dracula	130	Horror	1993	Columbia TriStar	R
22	Brave Heart	177	Action Adventure	1995	Paramount Pictures	R

Let's take it a step further. Suppose you did not want all the variables from the Movies data set – say you wanted only the Movie Title, Length, and Rating in that order. You could start with the PROC PRINT code from the previous example and insert a VAR statement between the PROC and RUN statements to tell SAS what variables to output along with their specific order on the report.

Procedure Code:

```
PROC PRINT DATA=SSU.MOVIES;
  VAR TITLE LENGTH RATING;
RUN;
```

Results:

The SAS System

Obs	Title	Length	Rating
1	The Wizard of Oz	101	G
2	Casablanca	103	PG
3	Jaws	125	PG
4	Rocky	120	PG
5	Star Wars	124	PG
6	Poltergeist	115	PG
7	The Hunt for Red October	135	PG
8	National Lampoon's Vacation	98	PG-13
9	Christmas Vacation	97	PG-13
10	Ghost	127	PG-13
11	Jurassic Park	127	PG-13
12	Forrest Gump	142	PG-13
13	Michael	106	PG-13
14	Titanic	194	PG-13
15	Dressed to Kill	105	R
16	Scarface	170	R
17	The Terminator	108	R
18	Lethal Weapon	110	R
19	Coming to America	116	R
20	Silence of the Lambs	118	R
21	Dracula	130	R
22	Brave Heart	177	R

Although the PROC PRINT code from the previous example printed the three variables you asked for, it also printed the Observation number as column one on the report. The observation number is automatically displayed as the first column on all PROC PRINT output – but can be suppressed by specifying the NOOBS option. Suppose you wanted to remove the observation column and change the current title appearing at the top of the report.

Procedure Code:

```
PROC PRINT DATA=SSU.MOVIES NOOBS;
  TITLE 'Movie Classics';
  VAR TITLE LENGTH RATING;
RUN;
```

Results:

Movie Classics

Title	Length	Rating
The Wizard of Oz	101	G
Casablanca	103	PG
Jaws	125	PG
Rocky	120	PG
Star Wars	124	PG
Poltergeist	115	PG
The Hunt for Red October	135	PG
National Lampoon's Vacation	98	PG-13
Christmas Vacation	97	PG-13
Ghost	127	PG-13
Jurassic Park	127	PG-13
Forrest Gump	142	PG-13
Michael	106	PG-13
Titanic	194	PG-13
Dressed to Kill	105	R
Scarface	170	R
The Terminator	108	R
Lethal Weapon	110	R
Coming to America	116	R
Silence of the Lambs	118	R
Dracula	130	R
Brave Heart	177	R

It is frequently necessary to subset the rows of data generated on output with a WHERE statement. Suppose you wanted to generate a report on PG and PG-13 rated movies.

Procedure Code:

```
PROC PRINT DATA=SSU.MOVIES NOOBS;
  TITLE 'Movie Classics';
  VAR TITLE LENGTH RATING;
  WHERE RATING IN ('PG','PG-13');
RUN;
```

Results:

Movie Classics

Title	Length	Rating
Casablanca	103	PG
Jaws	125	PG
Rocky	120	PG
Star Wars	124	PG
Poltergeist	115	PG
The Hunt for Red October	135	PG
National Lampoon's Vacation	98	PG-13
Christmas Vacation	97	PG-13
Ghost	127	PG-13
Jurassic Park	127	PG-13
Forrest Gump	142	PG-13
Michael	106	PG-13
Titanic	194	PG-13

Let's make one last change to the PROC PRINT code. Suppose you wanted to compute a total number of minutes for all movies in a rating group (e.g., G, PG, PG-13, and R). Since the Movies data set was originally sorted in ascending order by Rating, a BY statement can be specified with the Rating variable. A SUM statement is used to compute the total number of minutes for all movies in a By-group.

Procedure Code:

```
PROC PRINT DATA=SSU.MOVIES NOOBS;
  TITLE 'Movie Classics';
  BY RATING;
  VAR TITLE LENGTH RATING;
  SUM LENGTH;
RUN;
```

Results (Partial Output – Excludes R Ratings):

Movie Classics

Rating=G

Title	Length	Rating
The Wizard of Oz	101	G

Rating=PG

Title	Length	Rating
Casablanca	103	PG
Jaws	125	PG
Rocky	120	PG
Star Wars	124	PG
Poltergeist	115	PG
The Hunt for Red October	135	PG
Rating	722	

Rating=PG-13

Title	Length	Rating
National Lampoon's Vacation	98	PG-13
Christmas Vacation	97	PG-13
Ghost	127	PG-13
Jurassic Park	127	PG-13
Forrest Gump	142	PG-13
Michael	106	PG-13
Titanic	194	PG-13
Rating	891	

Using PROC SQL

The SQL procedure is known as the Structured (or Standard) Query Language and is a popular reporting tool among database users (e.g., SAS, Oracle, IBM,

etc.). In its simplest form, PROC SQL prints all variables (or columns) for all observations (or rows) in a data set (or table). As was illustrated with PROC PRINT, the SAS System writes a default title line at the top of each report page automatically. One or more TITLE statements can be specified to customize the title at the top of each page.

Besides using different statement syntax, an obvious distinction between PROC PRINT and PROC SQL is that a QUIT statement is specified for the latter, rather than a RUN statement, to terminate processing. The QUIT statement is used with interactive procedures.

Suppose you had to create a report containing all Movie data (all observations and variables) with the littlest amount of code possible. The PROC SQL statement illustrated below produces a detail-oriented report (similar to PROC PRINT) consisting of all observations and variables.

Procedure Code:

```
PROC SQL;
  TITLE 'Movie Classics';
  SELECT *
    FROM SSU.MOVIES;
QUIT;
```

When the PROC SQL code is executed, the SAS System applies certain defaults in creating report output including the default title, the number of observations, and list of variables.

Results:

Movie Classics

Title	Length	Category	Year	Studio	Rating
The Wizard of Oz	101	Adventure	1939	MGM / UA	G
Casablanca	103	Drama	1942	MGM / UA	PG
Jaws	125	Action Adventure	1975	Universal Studios	PG
Rocky	120	Action Adventure	1976	MGM / UA	PG
Star Wars	124	Action Sci-Fi	1977	Lucas Film Ltd	PG
Poltergeist	115	Horror	1982	MGM / UA	PG
The Hunt for Red October	135	Action Adventure	1989	Paramount Pictures	PG
National Lampoon's Vacation	98	Comedy	1983	Warner Brothers	PG-13
Christmas Vacation	97	Comedy	1989	Warner Brothers	PG-13
Ghost	127	Drama Romance	1990	Paramount Pictures	PG-13
Jurassic Park	127	Action	1993	Universal Pictures	PG-13
Forrest Gump	142	Drama	1994	Paramount Pictures	PG-13
Michael	106	Drama	1997	Warner Brothers	PG-13
Titanic	194	Drama Romance	1997	Paramount Pictures	PG-13
Dressed to Kill	105	Drama Mysteries	1980	Filmways Pictures	R
Scarface	170	Action Cops & Robber	1983	Universal Studios	R
The Terminator	108	Action Sci-Fi	1984	Live Entertainment	R
Lethal Weapon	110	Action Cops & Robber	1987	Warner Brothers	R
Coming to America	116	Comedy	1988	Paramount Pictures	R
Silence of the Lambs	118	Drama Suspense	1991	Orion	R
Dracula	130	Horror	1993	Columbia TriStar	R
Brave Heart	177	Action Adventure	1995	Paramount Pictures	R

PROC SQL allows for the subsetting of observations as was illustrated earlier with a WHERE statement in PROC PRINT. The only difference is that a WHERE clause is used in PROC SQL.

Procedure Code:

```
PROC SQL;
 TITLE 'Movie Classics';
 SELECT *
   FROM SSU.MOVIES
    WHERE RATING IN ('PG','PG-13');
QUIT;
```

Results:

Movie Classics

Title	Length	Category	Year	Studio	Rating
Casablanca	103	Drama	1942	MGM / UA	PG
Jaws	125	Action Adventure	1975	Universal Studios	PG
Rocky	120	Action Adventure	1976	MGM / UA	PG
Star Wars	124	Action Sci-Fi	1977	Lucas Film Ltd	PG
Poltergeist	115	Horror	1982	MGM / UA	PG
The Hunt for Red October	135	Action Adventure	1989	Paramount Pictures	PG
National Lampoon's Vacation	98	Comedy	1983	Warner Brothers	PG-13
Christmas Vacation	97	Comedy	1989	Warner Brothers	PG-13
Ghost	127	Drama Romance	1990	Paramount Pictures	PG-13
Jurassic Park	127	Action	1993	Universal Pictures	PG-13
Forrest Gump	142	Drama	1994	Paramount Pictures	PG-13
Michael	106	Drama	1997	Warner Brothers	PG-13
Titanic	194	Drama Romance	1997	Paramount Pictures	PG-13

Using PROC FORMS

The FORMS procedure provides a handy tool for printing label and form information. Generally the FORMS procedure is used when information is of a repetitive nature, such as mailing labels. In its simplest form, PROC FORMS prints just the information you specify using one or more LINE statements. Suppose you wanted to output the Movie Title, Category, and Rating variables for each observation in a single column.

Procedure Code:

```
TITLE;
PROC FORMS DATA=SSU.MOVIES;
  LINE 1 TITLE;
  LINE 2 CATEGORY;
  LINE 3 RATING;
RUN;
```

Results (Partial Output):

```
The Wizard of Oz
Adventure
G

Casablanca
Drama
PG

Jaws
Action Adventure
PG

Rocky
Action Adventure
PG

Star Wars
Action Sci-Fi
PG

Poltergeist
Horror
PG

The Hunt for Red October
Action Adventure
PG

National Lampoon's Vacation
Comedy
PG-13

Christmas Vacation
Comedy
PG-13

Ghost
Drama Romance
PG-13
```

The FORMS procedure statement has several options that can be specified to control the appearance of output. Table 2 below illustrates many important options and their descriptions.

Option	Description
DATA=	Identifies the input data set.
FILE=	Identifies an external output file.
LINES=	Number of lines in a form unit.
WIDTH=	Number of columns in a form unit.
ACROSS=	Number of form units across a page.
BETWEEN=	Number of spaces between form units.
DOWN=	Number of lines to skip before printing the first form unit.
ALIGN=	Number of dummy form units to print for alignment purposes.
COPIES=	Number of form units to print for each observation in data set.

Table 2. PROC FORMS options

Suppose you wanted to print the same information as the previous example, but instead of a single column of form units, you prefer to instruct PROC FORMS to construct two columns of form units. This is a popular format used with many of the leading printer-label products in use today (e.g., Avery).

Procedure Code:

```
TITLE;
PROC FORMS DATA=SSU.MOVIES
           ACROSS=2;
  LINE 1 TITLE;
  LINE 2 CATEGORY;
  LINE 3 RATING;
RUN;
```

Results:

```
The Wizard of Oz                Casablanca
Adventure                       Drama
G                               PG

Jaws                            Rocky
Action Adventure                Action Adventure
PG                              PG

Star Wars                       Poltergeist
Action Sci-Fi                   Horror
PG                              PG

The Hunt for Red October        National Lampoon's Vacation
Action Adventure                Comedy
PG                              PG-13

Christmas Vacation              Ghost
Comedy                          Drama Romance
PG-13                           PG-13

Jurassic Park                   Forrest Gump
Action                          Drama
PG-13                           PG-13

Michael                         Titanic
Drama                           Drama Romance
PG-13                           PG-13

Dressed to Kill                 Scarface
Drama Mysteries                 Action Cops & Robber
R                               R

The Terminator                  Lethal Weapon
Action Sci-Fi                   Action Cops & Robber
R                               R

Coming to America               Silence of the Lambs
Comedy                          Drama Suspense
R                               R

Dracula                         Brave Heart
Horror                          Action Adventure
R                               R
```

PROCs That Summarize

Detail reports are great in many situations – but sometimes contain so much information that it makes understanding their contents nearly impossible. When this is the case, a summary report may be in order. The purpose of a summary report is to collapse all the detail information in a report into easy-to-understand summary-level information. This helps to digest the enormous amounts of data frequently stored in a data set.

SAS provides the CHART, FREQ, MEANS, PLOT, SQL, TABULATE, and UNIVARIATE procedures to generate summary-level reports. For purposes of illustration, CHART, FREQ, MEANS, and UNIVARIATE will be presented.

As illustrated with the detail-level reporting procedures, each PROC is relatively easy to use, but each can also provide the level of support needed by even the most demanding programmer. By using one or more statements and options each PROC can produce simple to semi-custom reports.

Using PROC CHART

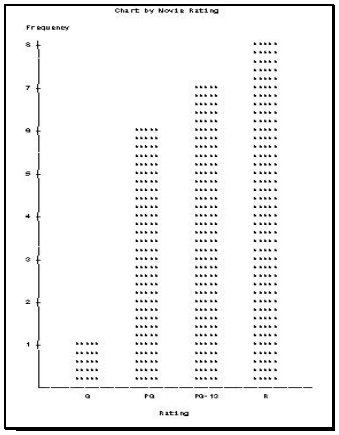
The CHART procedure is a line-oriented graphics tool that is used to print simple histograms (horizontal and vertical bar charts), block charts, and pie charts for numeric and character data. In its simplest form, PROC CHART summarizes the observations in a data set based on the variables listed in the type of chart being produced.

Suppose you wanted to display the number of movies grouped by their rating in a vertical bar chart.

Procedure Code:

```
PROC CHART DATA=SSU.MOVIES;
  TITLE 'Chart by Movie Rating';
  VBAR RATING;
RUN;
```

Results:

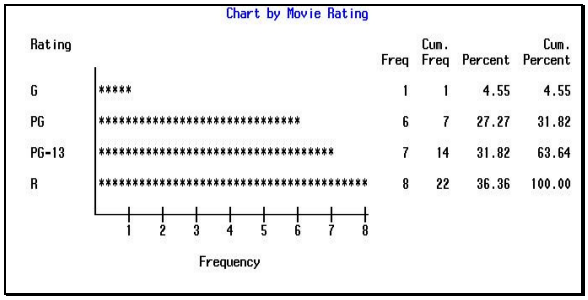


Suppose you wanted to see the number of movies grouped by their rating as a horizontal bar chart. Notice that basic-level statistics (e.g., frequency, cumulative frequency, percent, and cumulative percent) are automatically displayed with a horizontal bar chart.

Procedure Code:

```
PROC CHART DATA=SSU.MOVIES;
  TITLE 'Chart by Movie Rating';
  HBAR RATING;
RUN;
```

Results:



Using PROC FREQ

The FREQ procedure produces one-way to n-way frequency and cross-tabulation tables for numeric or character variables. In it simplest form, PROC FREQ produces a one-way frequency table.

Procedure Code:

```
PROC FREQ DATA=SSU.MOVIES;
  TITLE 'Frequency by Movie Rating';
  TABLES RATING;
RUN;
```

Results:

Frequency by Movie Rating

The FREQ Procedure

Rating	Frequency	Percent	Cumulative Frequency	Cumulative Percent
G	1	4.55	1	4.55
PG	6	27.27	7	31.82
PG-13	7	31.82	14	63.64
R	8	36.36	22	100.00

To see a cross-tabulation table for two variables, you will need to specify two variable names separated with an asterisk '*'.

Procedure Code:

```
PROC FREQ DATA=SSU.MOVIES;
  TITLE '2-Way Frequency Table';
  TABLES RATING * CATEGORY;
RUN;
```

Results (Partial Output):

2-Way Frequency Table

The FREQ Procedure

Table of Rating by Category

Rating Category

Frequency Percent Row Pct Col Pct	Action	Action Adventure	Action Cops & Robber	Action Sci-Fi	Adventure	Comedy	Total
G	0 0.00 0.00 0.00	0 0.00 0.00 0.00	0 0.00 0.00 0.00	0 0.00 0.00 0.00	1 4.55 100.00 100.00	0 0.00 0.00 0.00	1 4.55
PG	0 0.00 0.00 0.00	3 13.64 50.00 75.00	0 0.00 0.00 0.00	1 4.55 16.67 50.00	0 0.00 0.00 0.00	0 0.00 0.00 0.00	6 27.27
PG-13	1 4.55 14.29 100.00	0 0.00 0.00 0.00	0 0.00 0.00 0.00	0 0.00 0.00 0.00	0 0.00 0.00 0.00	2 9.09 28.57 66.67	7 31.82
R	0 0.00 0.00 0.00	1 4.55 12.50 25.00	2 9.09 25.00 100.00	1 4.55 12.50 50.00	0 0.00 0.00 0.00	1 4.55 12.50 33.33	8 36.36
Total	1 4.55	4 18.18	2 9.09	2 9.09	1 4.55	3 13.64	22 100.00

(Continued)

Using PROC MEANS

The MEANS procedure produces descriptive statistics for numeric variables only. In it simplest form, PROC MEANS will produce descriptive statistics for all numeric variables in a data set.

Procedure Code:

```
PROC MEANS DATA=SSU.MOVIES;
  TITLE 'Descriptive Statistics';
RUN;
```

Results:

Descriptive Statistics

The MEANS Procedure

Variable	N	Mean	Std Dev	Minimum	Maximum
Length	22	124.9090909	25.8344714	97.0000000	194.0000000
Year	22	1982.91	15.2124920	1939.00	1997.00

Analysis can be performed to help understand the data by using a numeric or character categorical variable in a CLASS statement. Using the CLASS statement will create descriptive statistics as subgroups. The next example illustrates descriptive statistics for each subgroup of movie by Movie Rating.

Procedure Code:

```
PROC MEANS DATA=SSU.MOVIES;
  TITLE 'Descriptive Statistics';
  CLASS RATING;
RUN;
```

Results:

Descriptive Statistics

The MEANS Procedure

Rating	N Obs	Variable	N	Mean	Std Dev	Minimum	Maximum
G	1	Length	1	101.0000000	.	101.0000000	101.0000000
		Year	1	1939.00	.	1939.00	1939.00
PG	6	Length	6	120.3333333	10.7641380	103.0000000	135.0000000
		Year	6	1973.50	16.2818918	1942.00	1989.00
PG-13	7	Length	7	127.2857143	33.9004144	97.0000000	194.0000000
		Year	7	1991.86	4.9809160	1983.00	1997.00
R	8	Length	8	129.2500000	28.4190580	105.0000000	177.0000000
		Year	8	1987.63	5.1806646	1980.00	1995.00

Using PROC UNIVARIATE

The UNIVARIATE procedure works similar to the MEANS procedure, except it provides a larger number of descriptive statistics. In its simplest form, PROC UNIVARIATE produces descriptive statistics for all numeric variables in a data set.

Procedure Code:

```
PROC UNIVARIATE DATA=SSU.MOVIES;
  TITLE 'Descriptive Statistics';
RUN;
```

Results (Partial Output):

Descriptive Statistics

The UNIVARIATE Procedure
Variable: Length

Moments

N	22	Sum Weights	22
Mean	124.909091	Sum Observations	2748
Std Deviation	25.8344714	Variance	667.419913
Skewness	1.45414494	Kurtosis	1.71453814
Uncorrected SS	357266	Corrected SS	14015.8182
Coeff Variation	20.682619	Std Error Mean	5.50792781

Basic Statistical Measures

Location		Variability	
Mean	124.9091	Std Deviation	25.83447
Median	119.0000	Variance	667.41991
Mode	127.0000	Range	97.00000
		Interquartile Range	24.00000

Tests for Location: Mu0=0

Test	-Statistic-		-----p Value------	
Student's t	t	22.67806	Pr > |t|	<.0001
Sign	M	11	Pr >= |M|	<.0001
Signed Rank	S	126.5	Pr >= |S|	<.0001

As with PROC MEANS, analysis can be performed to help understand the data by using a numeric or character categorical variable in a CLASS statement. Using the CLASS statement will create descriptive statistics as subgroups. The next example illustrates descriptive statistics for each subgroup of movie by Movie Rating.

Procedure Code:

```
PROC UNIVARIATE DATA=SSU.MOVIES;
  TITLE 'Descriptive Statistics';
  CLASS RATING;
RUN;
```

Results (Partial Output):

Descriptive Statistics

The UNIVARIATE Procedure
Variable: Length
Rating = G

Moments

N	1	Sum Weights	1
Mean	101	Sum Observations	101
Std Deviation	.	Variance	.
Skewness	.	Kurtosis	.
Uncorrected SS	10201	Corrected SS	0
Coeff Variation	.	Std Error Mean	.

Basic Statistical Measures

Location		Variability	
Mean	101.0000	Std Deviation	.
Median	101.0000	Variance	.
Mode	101.0000	Range	0
		Interquartile Range	0

Tests for Location: Mu0=0

Test	-Statistic-		-----p Value------	
Student's t	t	.	Pr > |t|	.
Sign	M	0.5	Pr >= |M|	1.0000
Signed Rank	S	0.5	Pr >= |S|	1.0000

PROCs for Data Management

Now we will turn our attention to a PROC that is used universally within the SAS user community to help with data management tasks: the DATASETS procedure. PROC DATASETS is a powerful PROC for any SAS user to know. It provides all the tools necessary to manage a SAS data library and the members within it. Table 3 below illustrates the various statements and tasks the DATASETS procedure can perform.

Since the DATASETS procedure is an interactive procedure (like the SQL procedure), it remains active even after a RUN statement is issued. To turn it off you would issue a QUIT statement. The general form looks something like the following:

```
PROC DATASETS LIBRARY=libref;
  Datasets-statement _____;
QUIT;
```

The beauty of the DATASETS procedure is that it copy, save, age, rename, and delete data sets. It can also produce a contents listing containing one or more members of a data library. Finally, it can be used to create a backup and recovery or data set aging process for important data sets.

Statement	Description
AGE	Create a backup and recovery process for important data sets.
APPEND	Concatenate one or more data set observations to the end of a “master” data set.
CONTENTS	Produce a detailed description of the members in a SAS data library.
COPY	Replicate one or more members in a SAS data library.
DELETE	Delete (remove) one or more data sets from a SAS data library.
MODIFY	Change attributes for one or more variables in a data set.
REPAIR	Restore damaged data sets or catalogs to a usable condition.
SAVE	Save specified members in a SAS data library and automatically deletes members not specified.

Table 3. PROC DATASETS Statements and Tasks

Describing Members of a Data Library

Suppose you were asked to produce a contents listing of an important SAS data library. By using PROC DATASETS, you could generate a detailed member listing of any SAS data library easily and quickly.

Procedure Code:

```
PROC DATASETS LIBRARY=SSU;
QUIT;
```

Results:

```
PROC DATASETS LIBRARY=SSU;
                          -----Directory-----

                Libref:       SSU
                Engine:       V8
                Physical Name: C:\SSU 2001 (New Orleans)\Papers
                File Name:    C:\SSU 2001 (New Orleans)\Papers
                                    File
                 #  Name    Memtype   Size  Last Modified

                 1  MOVIES  DATA      9216   01JAN2000:16:19:54
QUIT;
```

Aging for Backup and Recovery Purposes

Being able to create a backup and recovery process for important data sets can be critical should disaster strike. One or more related data sets can be assigned an aging number corresponding to when it was last updated. Suppose you wanted to safeguard the Movies data set by creating a safety net consisting of three versions of the data. The most recent version would be called MOVIES1, the next most recent version would be called MOVIES2, and the oldest and least recent version would be called MOVIES3.

Procedure Code:

```
PROC DATASETS LIBRARY=SSU;
  AGE MOVIES MOVIES1-MOVIES3;
QUIT;
```

Analysis:

Once this procedure is executed, each time the MOVIES data set is updated it will automatically rename the current data set to the first member name in the list, the next most recent data set in the member list will be renamed to the third name in the member list, and the oldest data set in the member list will be deleted. This aging process is especially useful for important data sets that are updated frequently

Repairing Damaged Data Sets

On rare instances where a system failure occurs (e.g., during a power brownout or electrical storm) during an update operation, a data set may become damaged an unusable. To fix a problem like this and restore a data set to a usable condition, it may be necessary to try to rebuild data set indexes. The PROC DATASETS REPAIR statement can be a life-saving statement (at least as it related to rescuing your damaged data set).

Suppose during an unexpected power outage our computer system experienced a problem that caused the Movies data set to become damaged. You could then use the REPAIR statement to try to restore the data set back to usability.

Procedure Code:

```
PROC DATASETS LIBRARY=SSU;
  REPAIR MOVIES;
QUIT;
```

Results:

```
58   PROC DATASETS LIBRARY=SSU;
                          -----Directory-----

                Libref:       SSU
                Engine:       V8
                Physical Name: C:\SSU 2001 (New Orleans)\Papers
                File Name:    C:\SSU 2001 (New Orleans)\Papers

                                    File
                 #  Name    Memtype   Size  Last Modified

                 1  MOVIES  DATA      9216   01JAN2000:16:19:54
59     REPAIR MOVIES;
60   QUIT;

NOTE: Repairing SSU.MOVIES (memtype=DATA).
NOTE: PROCEDURE DATASETS used:
      real time          0.04 seconds
```

Conclusion

Delivering timely and quality looking reports to management, end users, and customers is critical. With the SAS System's *"canned"* PROCedures for generating quick results, users around the world appreciate the benefits of using these "tried and proven" tools. Basic SAS PROCedures such as PRINT, SQL, and FORMS for detail reporting, and CHART, FREQ, MEANS, and UNIVARIATE for summary reporting are worth their weight in gold. So learn their syntax, use them wisely, and you will be happy you did. Happy computing!

Acknowledgments

The author would like to thank Tom Winn, Texas State Auditor's Office, Imelda Go, Richland County School District One, and Andrew T. Kuligowski, Nielsen Media Research for their support and encouragement in the creation of this paper, and for asking me to be a an invited speaker in the first place. I would also like to thank Deborah Babcock Buck, D. B. & P. Associates and S. David Riba, JADE Tech, Inc. for all their hard work and for doing a great job as SSU 2001 Conference C-Chairs. Thank you!

References

Delwiche, Lora D. and Susan J. Slaughter, "The Little SAS Book, A Primer (Second Edition)" SAS Institute Inc, 1998.

Lafler, Kirk Paul, SAS® Fundamentals, Version 8 Course Notes, Revised and Updated 2001, Software Intelligence Corporation, Spring Valley, CA, USA.

Trademark Citations

About the Author

Kirk is a SAS Quality Partner® and SAS Certified Professional® with 25 years of experience working with the SAS System. He has authored over one hundred articles on computing and technology and has presented at SAS Users Group International (SUGI) conferences, regional SAS User Groups, and local SAS User Groups since 1981. His popular SAS Tips column appears regularly in the SANDS and SESUG Newsletters and is read by thousands of SAS users. When Kirk isn't writing, teaching or consulting, he can be found enjoying all that San Diego has to offer with his wife and son.

Kirk can be reached at:

Kirk Paul Lafler
Author, Speaker, and Consultant
Software Intelligence Corporation
P.O. Box 1390
Spring Valley, California 91979-1390
E-mail: KirkLafler@cs.com
Website: http://www.software-intelligence.com

Formats, Informats and How to Program with Them

Ian Whitlock, Westat, Rockville, MD

Abstract

Formats tell how to display stored data and informats how to read them. In other words, they allow the separation of data from the values one works with. While most programming languages provide some sort of formatting capability, SAS® provides an extensive system that comes to dominate how one works with SAS.

This talk will review some of the supplied formats, show you how to make your own with code or data, and then supply programming examples for how to get the best use of this system.

Introduction

Formats allow a computer language to separate how data are stored from how the data are displayed. Informats allow the language to separate how data are entered from how the data are stored. In some sense formats and informats must play some role in almost all computer languages because numbers are rarely stored as they appear. The difference in SAS is the extent to which formats and informats are implemented. Thus they play a far more important role in SAS than they do in many other computer languages.

We use the term "formats" sometimes to refer to the class of both formats and informats, and sometimes to refer only to formats as distinguished from informats. It is left to the reader to distinguish based on context which is intended.

There are two types of formats - those automatically supplied by SAS, and those that you as a SAS programmer can create.

System Formats and Informats

There are formats to read and write numbers. Do you want them with commas? How many decimal places? Or do you prefer hexadecimal to decimal? The on-line documentation for SAS Version 8 lists 15 formats for character variables and 93 for numeric. The corresponding list of informats is not much shorter. Why are there so many? SAS has only two types of data values - character and real floating point. It falls to the system of informats to get data from the rich external world of many types into these two categories. Similarly the system of formats must display to the external world in many more types than just the two that we have. From this point of view it is not surprising that SAS had to have such a large system of formats and informats.

The two most basic formats are the F format for reading (or displaying) character digits and the $CHAR format for reading (or displaying) characters. Typically the $-sign is used to indicate character data.

Suppose that I have a number X somewhere between 3 and 4. How wide should the display field be? How many decimal places should there be? In general format names use an integer immediately following the name to indicate the total width (including any decimal point), followed by how many of those places should come after the decimal point. Thus F8.3 would say to display my number in 8 columns with exactly three rounded decimal places showing. The whole thing should be right justified.

So in the above case there would be exactly three leading blanks. In practice, one rarely gives the name F and only gives the 8.3. For example, I might have

```
data _null_ ;
   x = 12 / 3.7 ;
   put x 8.3 ;
run ;
```

This data step would write " 3.243" on the log.

On the otherhand, the informat F8.3 says something significantly different. Here a decimal point is usually displayed in the number. The 3 in F8.3 says if there is no decimal point shown then assume it is 3 places from the right hand end, i.e. divide by 1000. Consequently the decimal specifier in formats is important and a decimal specifier in the informat is usually a mistake that may look as if it is working, until it reads an integer value.

Perhaps the most important area for formats is that of dates. How should SAS store a date? Characters would rule out calculations like what month will it be in 82 days, so numeric is the obvious choice. But even having numbers does not mean that one can do arithmetic. Some computer systems store August 19, 2001 as 20010819. This is a number, but it is useless for doing date calculations.

About 400 years ago Rene Descartes tied geometry to numbers by choosing a 0 point and then a 1 point on a line. SAS does the same thing for the time line. It chose January 1, 1960 as the zero point and 1 to indicate one day. Thus January 2, 1960 is 1 and December 31, 1959 is -1. So August 19, 20001 is 15,206 days since January 1, 1960. Who can figure out their birthday that way? Aha! We said the system was good for calculations, we did not say it was a good thing to show your clients. Remember

formats is the topic and they should supply the solution. We need informats to read human recognizable dates into SAS and formats to display dates in human terms for our reports, but these human readable forms do not make a good way to store dates because they are hard to use in date calculations.

It should not surprise you by now to find there are 39 different formats for displaying dates listed in the on-line documentation for version 8. Here are a few examples using August 19, 2001.

DATE9. Displays army style (19Aug2001)

DAY2. Displays the day of the month (8)

Julian7. Displays Julian date (=2001231)

MMDDYY8. Displays American (08/19/01)

MONYY7. Displays (Aug2001)

WEEKDATE29. Displays
(Sunday, August 19, 2001)

WORDDATE18. Displays
(August 19, 2001)

The lesson should be clear, data should be stored for easy calculation. Formats and informats should be able to display and read the data in any reasonable fashion. A typical beginner's mistake is to fail to store dates as SAS dates, and consequently using elaborate and often incorrect routines to do simple date calculations provided by SAS functions. A description of the SAS date handling functions is beyond the scope of this paper, but coding a SAS date seems reasonable. To assign DATE the example value, use

```
date = "19aug2001"d ;
```

Note there are no spaces between the double quote and the letter, d.

Problem - calculate your age in days. Let's assume you were born on September 15, 1978.

```
data _null_ ;
  AgeInDays=today()-"15sep1978"d;
  put AgeInDays comma6. ;
run ;
```

SAS also provides a similar system for date time values as the number of seconds since midnight January 1, 1960. In addition there is a system to measure relative time in seconds since midnight.

Make Your Own Formats

With PROC FORMAT SAS provides the ability to make formats and informats. They are stored in a SAS catalog typically called FORMATS. (In general catalogs are used to hold system information created by the user. They differ from data in that the user cannot usually see inside a catalog entry.) By default the catalog is stored in the library, WORK.FORMATS. In this case the formats exist only for the life of the SAS session.

A common example is provided by storing gender information. For example, we typically store the number 1 for male and 2 for female. It would be embarrassing to hand a client a report from PROC PRINT with

ID	Gender	State
100001	1	LA
100002	2	LA
100003	.	AL

We need a format to dress up the report.

```
proc format ;
   value sex
      1 = "Male"
      2 = "Female"
      . = "Unknown"
   ;
run ;

proc print data = report ;
   var id gender state ;
   format gender sex7. ;
run ;
```

The VALUE statement provides a set of translations. On the left of the equal sign are the values stored. On the right are the character values to display. So now the report is.

ID	Gender	State
100001	Male	LA
100002	Female	LA
100003	Unknown	AL

This provides the simplest most basic use for formats.

Now let's consider an AGE variable. In a frequency report we might get any and all ages say between 5 and 85. But we would like to get age collapsed into a few groups. A format is the answer.

```
proc format ;
   value AgeGrp
      1 - 10 = "Child"
      11 - 20 = "Teenager"
      21 - 40 = "Adult"
      40 <-< 65 = "Middle Aged"
      65 - high = "Senior"
      other = "Error"
   ;
run ;
```

Note 40 is considered "Adult" not "Middle Aged" because the <-sign indicates that 40 is to be excluded. Similarly exactly 65 is excluded. "High" is a key word indicating the largest number. Note that key words are not quoted. Similarly there is a key word "low" for the lowest number (i.e. most negative number excluding missing values). "Other" is the key word for all values not accounted for.

It is not surprising that a print would show the labels instead of ages, but what about PROC FREQ? This procedure also respects formats for counting so that all people between 21 and 40 will be counted as adults. This provides a second important use of formats - grouping. In many languages one would have to create a new variable indicating the group in a process called recoding. Formats often make recoding unnecessary in SAS. However not all procedures respect formats so one must check each case individually. In general, formats are respected if it makes sense to do so.

How does the PROC FREQ know our variable, AGE, had a format? We could have told it in a FORMAT statement just as we did in the PRINT example, but it would be nice to tell the dataset instead, so that any procedure could check with the data to see if a format is present. This can be done by placing a FORMAT statement in the DATA step that makes the dataset. What if the data were made by a procedure or somebody else?

PROC DATASETS can modify the header of SAS dataset where knowledge of the format is stored.

```
proc datasets lib = survey ;
   modify agedata ;
   format age agegrp. ;
quit ;
```

Note that only the format name is stored in the header. Remember the format itself is stored in the WORK.FORMATS catalog.

After this we can produce the collapsed counts with:

```
proc freq data = survey.agedata ;
   table age ;
run ;
```

If, for some reason, we wanted to see the ages instead of groups we could remove the format association with a format statement.

```
format age ;
```

One might even use

```
format _all_ ;
```

to remove all format associations. This is often convenient when you do not have the associated formats or they are interfering with debugging and you need to see the values. Thus we see that formats may be applied, changed, and removed without ever modifying the data values.

Permanent Formats

PROC FORMAT has a LIBRARY parameter (abbreviated LIB) to specify where formats should be stored. Thus one might have

```
libname library "c:\myplace" ;
proc format lib = library ;
```

In this case the formats would be stored in LIBRARY.FORMATS. We use the libref LIBRARY because SAS by default looks in two places for formats

1. Work.Formats
2. Library.Formats

The concept of a library of formats in SAS is older than the notion of a libref and catalogs.

The system option FMTSEARCH allows you to specify which libraries and even which catalogs should be searched for formats. For example, the statement

```
options fmtsearch =
    ( mylib mylib.special ) ;
```

causes the system to look for formats in four places:

1. Work.Formats
2. Library.Formats
3. MyLib.Formats
4. MyLib.Special

If the format referenced is not found in any of the search catalogs, then the system issues an error message and may stop processing depending on the setting of the system option, FMTERR. By default it is set so that processing is stopped. To continue processing ignoring the request for the format, use

```
options nofmterr ;
```

One fear that often prevents beginners from making permanent formats is the fear that they cannot see the code or retrieve the formats when the code is lost. There is a PROC FORMAT option, FMTLIB which will report formats and informats. You can use a SELECT statement to obtain a report on specific formats or EXCLUDE statement to obtain all but a few formats.

Thus there is no good reason not to make permanent formats and in fact there are good reasons to do so. Formats can standardize and document how a project looks at its data values. Formats should become part of the standard environment in which project code is written rather than a part of the code. If there are many formats, it can take a significant amount of time to create the formats and they can take up a significant portion of the code in a program. Thus it makes a great deal of sense for a project to build a library of formats, just as it should have libraries of permanent SAS data.

Using SAS Datasets to Specify Formats

Just as the FMTLIB option can make a report on stored formats, the CNTLOUT= option names a SAS dataset that holds information capable of making the formats. Again SELECT or EXCLUDE statements can control which formats are represented in the dataset.

So far we have made formats with code, but now we can go directly from data to formats. This can be a very powerful idea because formats can be data dependent. Another thing this hints at is that formats can be quite large, say with a thousand or even a hundred thousand entries. Here it would be painful to have to write out such code.

Armed with an appropriate dataset, say, FMTDATA. The code is

```
proc format cntlin = fmtdata ;
run ;
```

So what properties must FMTDATA have. At a minimum only three variables are required.

1. FMTNAME to supply the name of the format.
2. START to supply the left hand side of an assignment.
3. LABEL to supply the right hand side of an assignment.

Suppose I am dealing with an educational survey and have a SAS data set with ID, an eight digit school identifier and SCHOOLNAME, a 40 byte character variable holding the name of the corresponding school. Then I might choose to store only the identifier on other data sets and use a format, $SCHLFMT. to display the names of the schools. The code to make the format might go like this.

```
data fmtdata ;
   retain fmtname "$schlfmt" ;
   set schools
     (keep = id schoolname
      rename = ( id = start
            SchoolName = Label )
     ) ;
run ;

proc format cntlin = fmtdata ;
run ;
```

Or perhaps you would like to show both the identifier and the school name. Then use

```
data fmtdata
    (keep = fmtname start label);
   retain fmtname "$schlfmt" ;
   set schools
     (keep = id schoolname
      rename = ( id = start )
     ) ;
   Label = id||", "||SchoolName
run ;

proc format cntlin = fmtdata ;
run ;
```

When you have learned the full power of DATA step character handling functions, you will find many opportunities for many variations of this idea.

So, just how large can a format be? Well typical of SAS, there is no prescribed limit. A format must be held in memory during its use so there is a practical limit. I consider 100,000 entries my limit, but even this can depend on how long the labels are.

Why do we need the format name as a variable? Couldn't it be a parameter on PROC FORMAT? No, one can actually specify many formats with one dataset, so it is better the way it has been designed.

What are the optional values and what kind of values must they have? The version 8 documentation leaves much to be desired on this point. The simple answer is: make a small format with code having the features you wish to know about. Then use the CNTLOUT= option to make the corresponding dataset and study it. For example, consider the key words and ideas we introduced earlier in a format.

```
proc format cntlout=fmtdata ;
   value datachk
      low -< 0 = "Negative"
      0 = "Zero"
      0 <- high = "Positive"
      other = "Missing"
   ;
   select datachk ;
run ;
```

You should quickly find that the relevant variables appear to be:

Start, End, Label, SExcl, EExcl, and HLO

In a table

Start	End	Label	SSexcl	EExcl	HLO
Low	0	Neg	N	Y	L
0	0	Zero	N	N	
0	High	Pos	Y	N	H
Other	Other	Missing	N	N	O

Armed with this information we can add an other condition to our $SCHLFMT.

```
data fmtdata ;
   retain fmtname "$schlfmt" ;
   if eof then
   do ;
```

```
        hlo = "O" ;
        label = "Error" ;
        output ;
    end ;
    set schools
      (keep = id SchoolName
       rename = ( id = start
             SchoolName = Label )
      ) end = eof ;
    output ;
run ;

proc format cntlin = fmtdata ;
run ;
```

Although it is not clear from the above, the value of START is irrelevant when HLO = "O". The format used to learn abut HLO is interesting in its own way. The following code provides a quick check for missing and negative values.

```
proc freq data = anyset ;
   tables _numeric_ / missing ;
   format _numeric_ datachk. ;
run ;
```

Remember PROC FREQ respects the grouping indicated by a format. Similarly

```
proc format ;
   value $datachk
      " " = "Blank"
      other = "present"
   ;
run ;
```

provides a quick check for character variables.

Informats

Up to this point we have concentrated on formats. It is now time to consider informats. They are made with an INVALUE statement. They come in two varieties - those that make numeric values and those that make character values.

Suppose you are reading a four digit column of numbers, but some of the entries are "XXXX" to indicate the value is missing. If you use

```
input ... number 4. ... ;
```

then there will be an invalid data message on the log. You could use

```
input ... number ?? 4. ... ;
```

to suppress the invalid data message, but then you also miss the message when it as appropriate. The answer is to make in informat.

```
proc format ;
   invalue numchk
   "XXXX" = . ;
run ;
```

Now the value "XXXX" is quietly read and converted to missing. What about other values? When a format or informat does not specify how to treat a value then the default treatment is used. Thus

```
input ... number numchk4. ... ;
```

will turn all other values over to BEST4. for reading as numbers. If a value cannot be read as a number then there will be a message to alert you to the problem.

As an example of a character informat we might return to our gender example. This time we want to read 1 or 2 but store "Male" or "Female".

```
proc format ;
   invalue gender
      "1" = "Male"
      "2" = "Female"
   ;
run ;

data _null_ ;
   length sex $ 6 ;
   input sex $gender1. ;
   put sex= ;
cards ;
1
2
;
```

Note that the LENGTH statement is needed to get the proper length for the variable, SEX. Unfortunately there is a bug in versions 8 and 8.1 so that only the first character is stored. In version 8.2 the expected behavior returns.

Key Idea

Informats and formats have been useful for reading from and writing to a buffer, but there is more. SAS also supplies the INPUT function to read from a character variable according to an informat and the PUT function to write to a character variable according to a format.

Suppose CHARX is a character variable holding 3 digits. Then we can convert to a numeric variable NUMX using

```
numx = input ( charx , 3. ) ;
```

Similarly we might convert in the other direction with

```
charx = put ( numx , 3. ) ;
```

Note that in both cases the format is numeric. In the first case it is numeric because the resulting value, NUMX, is numeric. In the second case it is a numeric informat because we are writing a numeric value. Beginners often get confused here and want to add a $-sign, but it is

wrong. Just remember INPUT always reads character stuff and PUT always writes character stuff.

Suppose we have a numeric id and we want a character id with leading zeros. There is a system format, Z for writing leading zeroes. Hence

```
charID = put ( numID , z8. ) ;
```

would convert to an 8-digit character identifier with leading zeros.

Often one wants to keep the same names. This gets a little tricky because it involves several ideas at once. Suppose we have a dataset W with a variable X that is character (always digits). We want to end up with a dataset W and a numeric variable X corresponding to the original X. We have to free up the name X on the input set so that we have it available for use on the output dataset. Here is the code.

```
data w ( drop = temp ) ;
   set w ( rename = (x=temp) ) ;
   x = input ( temp, best12. ) ;
run ;
```

The code is simple, but I suspect very few programmers discover and put together all the features needed for this problem. Most of them learn it from somebody who already knows the answer. It is my favorite problem for illustrating the inadequacy of SAS documentation since every feature is mentioned, but nowhere is the problem discussed. It is left to a "faq" sheet at http://www.sas.com.

Recoding and Look-up

In considering grouping formats, I pointed out that recoding is often not needed in SAS. However, sometimes one must recode because say a procedure like REG requires it. Consider the AGE variable we used before. Now we make a new format

```
proc format ;
   value AgeGrp
      1 - 10 = "1 Child"
      11 - 20 = "2 Teenager"
      21 - 40 = "3 Adult"
      40 <-< 65= "4 Middle Aged"
      65 - high = "5 Senior"
      other = ". Error"
   ;
run ;
```

and use it in a DATA step

```
data recoded ( drop = age ) ;
   set agedata ;
   agegrp =
     input(put(age,agegrp1.),1.);
run ;
```

to recode the AGE variable.

Now consider a slight change in point of view and the format $SCHLFMT that we made from a SAS dataset. Here the line

```
SchoolName = put(id, $schlfmt40.);
```

can be thought of as a look-up function - given the ID value for a school look up the name of the school.

Often one solves this type of look-up problem with a sort and merge by ID where one file has the ID and needed data while the other file has ID and SchoolName. There are advantages to both.

The primary advantage of the format method is that the files do not need sorting. This can be very important when you need to look up values based on several different variables. The format method provides flexibility.

On the other hand, the merge solution is probably better when one wants to look up many different variables where the look-up information is all in one file. The merge is more efficient when the files happen to already be in their required order. As the size of the look-up file grows, so does the importance of this efficiency.

SAS arrays are numerically indexed. Sometimes it would be very convenient if one could have an array indexed by character values. Suppose we wanted to count how many times certain words were used in some text. For a list of say 100 words we might set up an array with

```
array count (100) ;
```

The problem is which word goes with which array element. In this case it doesn't matter as long as it is fixed in some specific order. We might use an informat, SPECIAL, to recode the special words into the numbers from 1 to 100. We could then send all other words to 0. Now for any value of WORD we could use

```
x = input ( word , special3. ) ;
if x > 0 then
     count ( x ) + 1 ;
```

subsetting with wanted format

Eliminating IFs

All of the problems discussed in the previous section have lot in common, and they could have been solved with IF statements. IF statements provide a powerful method for making programs flexible. However they should be used with great care because they also make a program more difficult to follow. A reasonable goal is to eliminate as many IFs as possible.

Consider the recoding of age. We could have written

```
data recoded ( drop = age ) ;
   set agedata ;
   if 1 <= age <= 10 then
      AgeGrp = 1 ;
```

```
    else
    if 11 <= age <= 20 then
       AgeGrp = 2 ;
    else
    if 21 <= age <= 40 then
       AgeGrp = 3 ;
    else
    if 40 < age < 65 then
       AgeGrp = 4 ;
    else
       AgeGroup = 5 ;
run ;
```

Note that this step is longer and more tedious with poorer documentation. After comparing it with our original recoding, it is clear that what the format did was to give us the ability to lift the tedious IF/ELSE chain out of the DATA step and move it to a more appropriate place. But then if it were a stored format it would not even be in the code at all. In any case it allows us to treat the IF/ELSE chain as a black box.

For the example the IF/ELSE chain was not too bad because it was relatively short, but the school name look-up in the second example might have gone on for thousands of lines. In the school name case, remember we actually started with a dataset and made the format from data. It should now be clear that we have three different methods of storing information:

1. Store it in a dataset
2. Store it in a format
3. Store it in SAS code

So formats provide another method of storing and using information. In considering the three methods, code is usually the hardest to modify and the most tedious to change. It is also the most likely to conceal a mistake. Where PROC FORMAT will display an error messaage when overlapping ranges are specified, the compiler will find nothing wrong with the corresponding IF/ELSE chain and yet it is likely that the code is wrong.

I call the above DATA step "wall paper" code. It all looks the same with a simple pattern to it. In general it is a good idea to remove wall paper code and formats provide one important tool for doing this. Arrays provide another most important tool in this area. The third tool, SAS macro, provides the most sophisticated tool for this purpose. However, it is a good idea for the beginner to get a sound grasp of the principles involved with formats and arrays before turning to macro code.

Picture Formats

The VALUE statement provides you with the ability to specify a single display value for a collection of stored values. Sometimes it would be nice to simply add some extra symbols to the number. For example, display the phone number, 1234567890, as (123)456-7890. Note that this cannot be done in general with a VALUE statement. Or perhaps you would like to display money values in thousands of dollars, e.g. 143,265 dollars would display as $143K. For this type of problem we need a new statement, the PICTURE statement. The idea originally came from COBOL which used picture formats.

For the telephone example, one might try

```
proc format ;
   picture phone
      0 - 9999999999 =
         "(999)999-9999"
   ;
run ;
```

However, this does not work; 1234567890 is shown as

```
123)456-7890
```

with the leading left parenthesis missing. When the leading characters are not numbers one has to specify them in a special option prefix.

```
proc format ;
   picture phone
      0 - 9999999999 =
         "*999)999-9999"
         (prefix="(")
   ;
run ;
```

Note that in the picture itself I placed an asterisk first. It doesn't matter what this symbol is, but space must be reserved for the prefix symbols. This makes for what I call the "Mother May I" type of syntax after the children's game "giant steps". It is a case where the developer took short - cuts at the expense of the user.

For the money in thousands example, we could use

```
proc format ;
   picture money
      500 - 999499 = "*009K"
         (prefix="$" mult=.001)
      other = "error" ;
run ;
```

In this example we added another picture option, MULTIPLIER= (or MULT=). This tells SAS to divide the number by 1000 before placing into the picture. Again there is a funny thing. If we consider the number, 1899, is displayed as

```
$1K
```

instead of the expected, $2K. In general, formats will round numbers to the indicated number of decimal places; however picture formats truncate instead of round. Fortunately there is a ROUND option, but it is a general format option instead of one specific to the PICTURE statement. Hence, it cannot be placed in the parentheses

with the PREFIX= option as one might expect. The correct form of the statement is

```
picture money (round)
    500 - 999499 = "*009K"
        (prefix="$" mult=.001)
    other = "error" ;
```

Finally suppose we want to display money values in thousands, but allow one decimal place. Then one might think that

```
picture money (round)
    500 - 999499 = "*009.0K"
        (prefix="$" mult=.001)
    other = "error" ;
```

would work. However, the decimal point in the picture indicates that SAS is to divide by 10 so that the correct statement is

```
picture money (round)
    50 - 999949 = "*009.0K"
        (prefix="$" mult=.01)
    other = "error" ;
```

Note that I have extended the range of valid values because we can now display a wider range of values because of the extra decimal place. Of course we do not have to make all other values and error. We could have left them to display in an unformatted form or we could have added more range pictures to include more values.

Why are picture formats so funny to work with? I suspect it is the COBOL heritage showing in SAS.

Nested Formats

Formats and informats can be nested. That is you can use a format in specifying a range for another format. For example, suppose we want to display dates for this year using the DATE9. format, but all other dates should display just the year. Then the code might be

```
proc format ;
    value thisyr
        "1jan2001"d - "31dec2001"d
            = [date9.]
        other = [year.]
    ;
run ;
```

Note that the specified format in the label does not and cannot appear in quotes. On some computer systems you have to use the combination "(|" for "[" and "|)" for "]".

The above example used system formats, but the same principle can be handy with user formats. Suppose you have project format PROJECT, but the codes for 1 and 2 need to be changed. Then instead of rewriting the whole format you could use

```
proc format ;
    value myproj
        1 = "First change"
        2 = "Second change"
        other = [project16.]
    ;
run ;
```

Although the first example format was called THISYR, the name is really only good for 2001. How could one write the code so that the format would always be appropriate to this year? For this simple example one could use some macro code, but let's solve it with simpler DATA step techniques. Remember we can use a SAS dataset to specify a format. In this case HLO is used to also specify format instructions.

```
data fmtdata ;
    fmtname = "thisyr" ;
    start = intnx ( "year",
                    today(),
                    0 ) ;
    end = intnx ( "year" ,
                  today(),
                  0,
                  "end" ) ;
    label = "date9." ;
    hlo = "F " ;
    output ;
    label = "year4." ;
    hlo = "FO" ;
    output ;
run ;

proc format cntlin = fmtdata ;
run ;
```

Note that there are no brackets around the name of the nested format. The variable HLO tells SAS that the label is a format name instead of the brackets used with format code.

Multi-label Formats

Version 8 has introduced a new option for formats, the MULTILABEL option. In general, it doesn't make much sense to have one value display is more than one way. However, remember that PROC SUMMARY and PROC TABULATE respect formats in grouping statistics. For these two procedures it does make sense to allow one value to go to several labels. In this case we want several groups to include the corresponding statistics. The same reasoning might apply to PROC FREQ but the multilabel option has not been implemented for it.

For example, suppose we have a variable RATE with values 1 to 10. Perhaps 1 - 5 are considered low and 6 - 10 high. Now we would like to add an overlapping middle group 4 - 7.

```
proc format ;
```

```
    value rate (multilabel)
        1-5 = "Low"
        6-10 = "High"
        4 - 7 = "Middle"
    ;
run ;

proc tabulate data = look ;
    class rate / mlf ;
    format rate rate. ;
    table rate * n ;
run ;
```

Note that the CLASS statement uses the option MLF to stand for multilabel. In the "Mother May I" fashion that SAS has been developing since the change to a C based language, you cannot spell it out the CLASS statement, while you must spell it out in the FORMAT statement.

Conclusion

We have not discussed all the options and things one can do with formats and informats. However, I hope that we have covered enough for you to see that it is an intrinsic part of SAS that you must know if you are going to be a SAS programmer.

Contact Information

The author may be contacted via mail using the address

Ian Whitlock
Westat
1650 Research Boulevard
Rockville, MD 20850

or perhaps better via e-mail at

IanWhitlock@westat.com

What's Next?

Thomas J. Winn, Jr.
Texas State Auditor's Office, Austin, Texas

Abstract

Intended for attendees who have learned the material that was covered in the earlier presentations in the "Introduction to SAS" Section, this short paper concludes the Section by describing some recommendations for the next developmental steps which could be taken by beginning SAS programmers. The paper includes highlights of the following topics: writing customized reports (PROC PRINT, DATA _NULL_, PROC TABULATE, & PROC REPORT), the SAS macro language, and PROC SQL.

Recommendations for your next developmental steps

Let us assume that one has learned the material which was covered in the earlier presentations in the SSU 2001 "Introduction to SAS" Section. Now, I would like to conclude this Section by describing some recommendations regarding the next developmental steps that could be taken by beginning SAS programmers.

Specifically, I will include highlights of the following topics:

- Output Delivery System,
- customized report writing, using PROC PRINT, PROC REPORT, PROC TABULATE, and DATA _NULL_,
- the SAS macro language, and
- PROC SQL.

A working knowledge of these topics should be included in the basic toolkit of every SAS programmer.

Here are a few ways to take those next steps:

- Take a training class,
- Attend tutorial presentations at SAS users conferences,
- Consult with an expert,
- Read the documentation or other appropriate reference materials.

Most of the topics are covered in papers presented in the SSU 2001 Tutorials section!

Output Delivery System

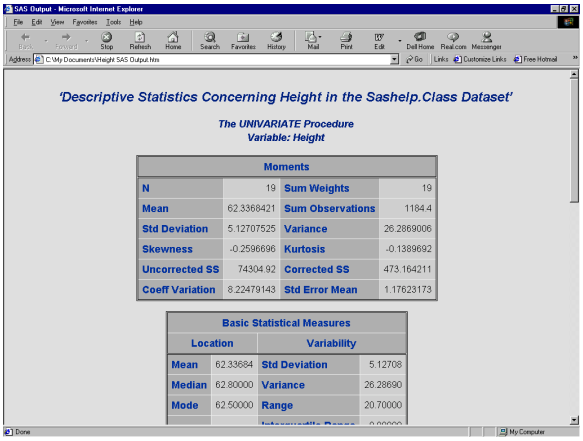
Beginning with Version 7, the SAS System provided an ability to deliver procedure output in a flexible variety of file types and formats, through the SAS Output Delivery System (ODS). ODS combines raw data with table definitions to produce output objects, which can be sent to one or more ODS destinations. Currently, the available ODS destinations include: the Listing destination, the Output destination (SAS Data Set), the HTML destination, the Printer destination (PostScript, PDF, PCL), and the RTF destination.

ODS destinations can be either open or closed. When a destination is open, ODS can send output objects to it. And whenever an ODS destination is closed, output objects cannot be sent to it. ODS statements are used to control different features of the Output Delivery System. For example, they could be used to open, close, or manage an ODS destination. Moreover, they can be used anywhere in a SAS program.

Here is an example of the use of ODS statements to direct some SAS output to HTML:

```
ods html file='odshtml-body.htm'
          contents='odshtml-contents.htm'
          page='odshtml-page.htm'
          frame='odshtml-frame.htm';
proc univariate data=sashelp.class;
     var height;
     id name;
     title 'Descriptive Statistics Concerning Height
in the Sashelp.Class Dataset';
run;
ods html close;
```

And here is a portion of the HTML output that was generated:



'Descriptive Statistics Concerning Height in the Sashelp.Class Dataset'

The UNIVARIATE Procedure
Variable: Height

Moments			
N	19	Sum Weights	19
Mean	62.3368421	Sum Observations	1184.4
Std Deviation	5.12707525	Variance	26.2869006
Skewness	-0.2596696	Kurtosis	-0.1389692
Uncorrected SS	74304.92	Corrected SS	473.164211
Coeff Variation	8.22479143	Std Error Mean	1.17623173

Basic Statistical Measures			
Location		Variability	
Mean	62.33684	Std Deviation	5.12708
Median	62.80000	Variance	26.28690
Mode	62.50000	Range	20.70000

In most cases, the default output style will be adequate. However, the programmer may customize ODS output by specifying style definitions, which affect the colors, font, size, etc.

Customized Reports with PROC PRINT

PROC PRINT has several features and options which can be used to customize the appearance of detail reports printed with this procedure.

- DATA=, DOUBLE, HEADING=, LABEL, N, NOOBS, OBS=, ROUND, SPLIT=, and UNIFORM,
- VAR, ID, BY, SUM, PAGEBY, SUMBY, FORMAT, TITLE, and FOOTNOTE.

PROC TABULATE

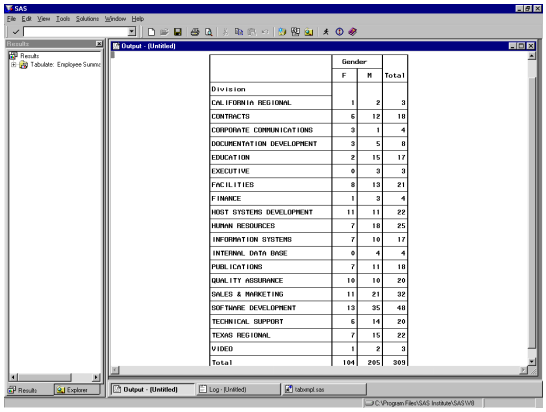
PROC TABULATE is used to build tabular reports containing descriptive statistical information, including

hierarchical relationships among variables. PROC TABULATE is particularly well-suited for preparing summary reports (where each row represents multiple observations). However, it is less well-suited for producing detail reports (single row for each observation).

Here is some SAS code for a simple tabular report:

```
libname company 'C:\Program Files\SAS
    Institute\SAS\V8\core\sample' ;
proc tabulate  data=company.empinfo
    format=5.0;
    class division gender;
    keylabel n='  '
             all='Total';
    label division = 'Division'
          gender = 'Gender';
    table division all, gender all / rts=30
    misstext='0';
    title 'Summary of Employee Information';
run;
```

Here is the associated report:



Division	Gender		
	F	M	Total
CALIFORNIA REGIONAL	1	2	3
CONTRACTS	6	12	18
CORPORATE COMMUNICATIONS	3	1	4
DOCUMENTATION DEVELOPMENT	3	5	8
EDUCATION	2	15	17
EXECUTIVE	0	3	3
FACILITIES	8	13	21
FINANCE	1	3	4
HOST SYSTEMS DEVELOPMENT	11	11	22
HUMAN RESOURCES	7	18	25
INFORMATION SYSTEMS	7	10	17
INTERNAL DATA BASE	0	4	4
PUBLICATIONS	7	11	18
QUALITY ASSURANCE	10	10	20
SALES & MARKETING	11	21	32
SOFTWARE DEVELOPMENT	13	35	48
TECHNICAL SUPPORT	6	14	20
TEXAS REGIONAL	7	15	22
VIDEO	1	2	3
Total	104	205	309

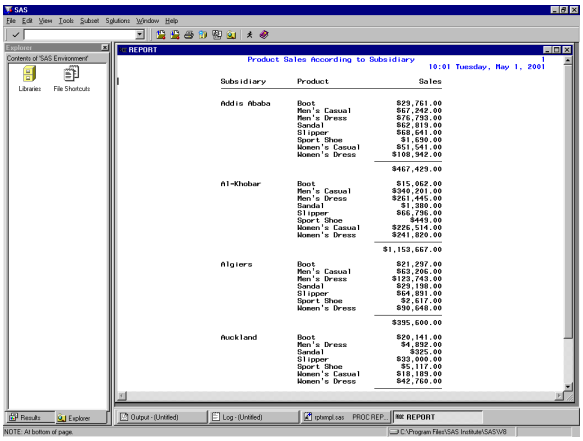
PROC REPORT

PROC REPORT is a flexible procedure that can produce a variety of reports. PROC REPORT offers more control and customization than PROC PRINT. PROC REPORT can be used to produce both detail and summary reports; however, it is not as good as PROC TABULATE for producing hierarchical tables.

Here is some SAS code for a simple PROC REPORT step:

```
proc report  data=sashelp.shoes  headskip
    headline missing;
  column subsidiary product sales;
  define subsidiary / order          'Subsidiary'
    format=$15.;
  define product     / order         'Product'
    format=$15.;
  define sales       / analysis sum  'Sales'
    format=dollar15.2;
  break after subsidiary / ol summarize skip
    suppress;
  title 'Product Sales According to Subsidiary';
run;
```

Here is the report which was produced:



Product Sales According to Subsidiary

10:01 Tuesday, May 1, 2001

Subsidiary	Product	Sales
Addis Ababa	Boot	$29,761.00
	Men's Casual	$67,242.00
	Men's Dress	$76,793.00
	Sandal	$62,819.00
	Slipper	$68,641.00
	Sport Shoe	$1,630.00
	Women's Casual	$51,541.00
	Women's Dress	$108,942.00
		$467,429.00
Al-Khobar	Boot	$15,062.00
	Men's Casual	$340,291.00
	Men's Dress	$261,445.00
	Sandal	$1,380.00
	Slipper	$66,756.00
	Sport Shoe	$449.00
	Women's Casual	$226,514.00
	Women's Dress	$241,820.00
		$1,153,667.00
Algiers	Boot	$21,297.00
	Men's Casual	$63,206.00
	Men's Dress	$123,743.00
	Sandal	$29,198.00
	Slipper	$64,891.00
	Sport Shoe	$2,617.00
	Women's Dress	$90,648.00
		$395,600.00
Auckland	Boot	$20,141.00
	Men's Dress	$4,032.00
	Sandal	$325.00
	Slipper	$33,000.00
	Sport Shoe	$5,117.00
	Women's Casual	$18,189.00
	Women's Dress	$42,760.00

Writing Customized Reports

FILE and PUT statements can be used in a DATA _NULL_ step to write special reports according to detailed specifications. The disadvantage is that pointer controls must be used to specify the placement of every item in the report.

The SAS Macro Language

The SAS Macro Language is a powerful programming tool ...

- for simplifying repetitive coding,
- for communicating information between program steps,
- for generating data-dependent SAS statements,
- for permitting conditional execution of SAS code, and
- for dynamically importing certain information from the SAS Supervisor.

Macros are stored text that contain entire blocks of SAS code, and which are identified by a name. The stored text can include SAS statements, literals, numbers, macro variables, macro functions, macro expressions, or calls to other macros.

Macro variables are used to facilitate symbolic substitution of strings of text, whereas macros can be used to manipulate SAS source statements. Macro information can be inserted at any point in a SAS program simply by referring to the macro entity by name, preceded by a special character, which distinguishes macro statements from ordinary SAS code.

PROC SQL

Structured Query Language (SQL) is a language that talks to a relational database management system. PROC SQL processes SQL statements that read and update tables. Besides being useful for queries, it also is a powerful tool for data manipulation.

PROC SQL uses Structured Query Language to . . .

- retrieve and manipulate SAS data sets,
- create and delete data sets,

- add or modify data values in a data set,
- add, modify, or drop columns in a data set,
- create and delete indexes on columns in a data set.

Components of the SAS System

Do you remember this display from the very first "Introduction to SAS" presentation? It is a listing of many of the components of the SAS System.

- Base SAS	- SAS/LAB
- SAS/ACCESS	- SAS/MDDB Server
- SAS/AF	- SAS/OR
- SAS/ASSIST	- SAS/QC
- SAS/CONNECT	- SAS/SHARE
- SAS/EIS	- SAS/SPECTRAVIEW
- SAS/ETS	- SAS/STAT
- SAS/FSP	- SAS/TOOLKIT
- SAS/GIS	- SAS/AppDev Studio
- SAS/GRAPH	- SAS/Enterprise Guide
- SAS/IML	- SAS/Enterprise Miner
- SAS/INSIGHT	- SAS Universal ODBC Driver
- SAS/IntrNet	- SAS/Warehouse Administrator

After becoming familiar with Base SAS, novice SAS programmers might begin learning other pertinent components of the SAS System.

"What's Next?" Summary

Build on the foundation of knowledge you already have acquired. Your next developmental steps might include the following suggested topics:

- Output Delivery System,
- customized report writing, using PROC PRINT, PROC REPORT, PROC TABULATE, and DATA _NULL_,
- the SAS macro language,
- PROC SQL,
- other components of the SAS System.

Here are some ways by which you might take those next developmental steps:

- Take a training class.
- Attend tutorial presentations at SAS users conferences (Be sure to review the many excellent papers in the SSU 2001 Tutorials Section!).
- Consult with an expert.
- Read the documentation or other appropriate reference materials.

The next step is up to you.

Suggested References:

SAS Institute Inc., SAS OnlineDoc, Version 8
SAS Institute Inc., Getting Started With the SAS System, Version 8
SAS Institute Inc., SAS Language Reference: Concepts, Version 8
SAS Institute Inc., SAS Language Reference: Dictionary, Version 8, Volumes 1 and 2
SAS Institute Inc., SAS Procedures Guide, Version 8, Volumes 1 and 2

Author Information

Tom Winn
Texas State Auditor's Office
P.O. Box 12067
Austin, TX 78711-2067

Telephone: 512 / 936-9735
E-mail: twinn@sao.state.tx.us

POSTERS

SECTION CHAIRS

Philip d'Almada
EDS

Eric Brinsfield
Meridian Software, Inc

POSTERS

Cubes on the Cheap

By Jimmy DeFoor
Brierley and Partners

Cubes are hot items in the OnLine Analytical Processing (OLAP) environment. The functionality is usually quite costly to acquire, but equivalent content with reasonably good navigational capabilities can be created using linked HTMLs generated by SAS Data Steps and simple Put _ODS_ statements. This paper explains how to create such 'cubes' and provides examples of the code that would create them.

A cube is a multi-level view of the dimensions of a data set. A multi-level view is one in which statistical measures exist for each combination of the dimensions of the data. For example, if the dimensions of sales data for a grocery store chain were Year, Month, Store, and Household, a cube would have statistical measures such as sums, averages, and medians for sales at every combination of those dimensions.

The different levels of a cube are created by collapsing one or more dimensions and then calculating statistical measures on the other dimensions. For example, Year is collapsed and statistical measures are calculated on sales for each store, household and month for all years. Year and Month are then both collapsed and statistics are calculated on sales at each store and household for all years and all months. This process is repeated until all combinations have been evaluated.

The number of levels in a cube is calculated using the exponential function: 2**D, where D is the number of dimensions. For example, four dimensions create 16 levels; three dimensions, 8 levels; two, 4; one, 2.

Proc Summary generates these multi-level views in one pass of the input data set. By default, it generates sums for every level, but other measures can be requested, including maxima, minima, P10, P90, etc. Each level is given a value generated by the _Type_ field. This value is based upon the order of the dimensions in the Class statement and the rule that a zero (0) is assigned to a field when it is collapsed and a one (1) when it is not.

If the order of the dimensions were Year, Month, Store, and Household, Proc Summary would assign a value of 0000 to the level that had all dimensions collapsed and a value of 1111 to the dimension which had no fields collapsed. It would assign 0011 to the dimension that represented a collapse of Year and Month and a 1000 to the dimension that represented a collapse of Month, Store, and Household.

When a field is collapsed, Proc Summary sets that field to blank. Creating a user format and adding it to the Proc Summary will change the blanks to 'Collapsed' in the output data set.

```
Proc Format;
    Value $Blank
      '  '  =  'Collapsed'
    ;
Run;
*;
Proc Summary data = DataforCube;
  Class Year Month Store Household;   /* Dimension fields */
  Var   Sales Visits;                 /* Measure fields   */
  Output out= DataforCubeStats(drop=_freq_) /* drop record counts */
    sum    = Sales    Visits
    mean   = AvgSales AvgVisits
    median = P50Sales P50Visits
    ;
  Format Year Month Store Household $Blank.0; /* Change Blank to Collapsed */
Run;
```

Proc Summary doesn't write out the _Type_ field in the binary format. Instead, it converts it to a decimal value. A value of 1111 becomes 8 + 4 + 2 + 1 or 15. A value of 0011 becomes 0 + 0 + 2 + 1 or 3. A value of 1011 becomes 8 + 0 + 2 + 1 or 11. This calculation can be done in a Do loop:

```
TypeValue  =   0;
Pos        =   0;
Dimension  =   4;
Start      =  Dimension - 1;
Do J =  Start to 0 by -1;
  Pos  =  Pos + 1;
  PosValue  = Input(Substr(_Type_,Pos,1),3.0);
  TypeValue = TypeValue + 2**(J*PosValue);
End;
_Type_ = Put(TypeValue,4.0);
```

Knowing this, a programmer can reconstruct the original value of _Type_ and place it into a new field, BinaryType, which can be used to separate the output of Proc Summary into a data set for each level.

```
Array  Binary    (0:3) p0 p1 p2 p3;
Array  Exponent  (0:3) e0 e1 e2 e3
                      (0  1  2  3); /* values of fields in array */
Remainder  = _Type_;
/* Start with highest exponent and evaluate downward. */
Do J = 3 to 0 by -1;
  BinaryValue = 2**Exponent(J);
  If Remainder ge BinaryValue then
    do;
       Binary(J) = '1';
       Remainder = Remainder - BinaryValue;
    end;
  else
       Binary(J) = '0';
End;
/* Create Binary Type field by concatenating array values */
Binarytype = P3||P2||P1||P0;
```

After separate data sets are built for each BinaryType, an HTML is created for each using the ODS Html statement and Put_ODS_. The cube is built by linking the HTMLs to a Table of Contents and to each other.

The links are constructed using the anchor HTML tag, <a, and the hyperlink tag, href.

```
<a href=Type0001.html> "0001" </a>
```

The text inside the quotes is underlined in the HTML. When the text is clicked, the browser opens the file referenced by the href.

Here is an example of a Table of Contents with links to each report. Not all levels (_Types_) are shown. The underlined values contain the links.

Household Sales and Visits Data of Ajax Supermarkets
Table of Contents
Click Report to See That Level of the Cube

Report	Level	Year	Month	Store	Household
0000	0	Collapsed	Collapsed	Collapsed	Collapsed
0001	1	Collapsed	Collapsed	Collapsed	Expanded
0010	2	Collapsed	Collapsed	Expanded	Collapsed
0011	3	Collapsed	Collapsed	Expanded	Expanded
0100	4	Collapsed	Expanded	Collapsed	Collapsed
1111	15	Expanded	Expanded	Expanded	Expanded

Notice the entries under each dimension. They are either Collapsed or Expanded to indicate the summary level of that column. They are created by sub-stringing the Report field for the value of the dimension and writing 'Collapsed' if it is a zero (0) and 'Expanded' if it is a one (1). The Level field is created from _Type_ field that was output by Proc Summary. The Report field is created using the BinaryType field generated from the _Type_ field.

Each linked HTML will have the data for the level listed in that Report. For example, the 0100 Report would have monthly sales data over all years, all stores, and all households. In this example, it shows total, average, and P50 (median) sales for each month.

Household Sales and Visits Data of Ajax Supermarkets
Report 0100
Go to Table of Contents

Year	Month	Store	Household	TotalSales	AvgSales	P50Sales
Collapsed	January	Collapsed	Collapsed	10,000,000	2,000,000	1,800,000
Collapsed	February	Collapsed	Collapsed	11,000,000	2,200,000	1,900,000
Collapsed	March	Collapsed	Collapsed	12,000,000	2,400,000	2,200,000
Collapsed	April	Collapsed	Collapsed	11,000,000	2,200,000	2,000,000
Collapsed	May	Collapsed	Collapsed	12,000,000	2,400,000	2,100,000
Collapsed	June	Collapsed	Collapsed	13,000,000	2,600,000	2,300,000

A link back to the Table of Contents is placed in the header and footer of the report so that the user can easily move to any level of the cube. Though needed, that approach is insufficient. It does not give the user an intuitive method of expanding and collapsing dimensions. The 'Expand, Collapse' method is created by adding a table at the top and bottom of each report that takes the user to the 'opposite' view of each dimension. This is the table that would be used in the 0100 Report.

Year	Month	Store	Household
Expand	Collapse	Expand	Expand

The links for each column of the header and footer table are created by 'flipping' each digit one at a time, leaving the other digits unchanged. For example, the first digit of 0100 is for the Year dimension. Flipping the first digit to 1 creates a BinaryType of 1100. It shows expanded sales values for both Year and Month. Similarly, flipping only the second digit creates a BinaryType of 0000, which is the top level of all dimensions meaning all four dimensions have been collapsed.

Creating these links is done with the following code. It flips each digit and builds a ReportType of either expand or collapse. It then creates a value for each variable that contains the link and the ReportType. The resulting variables are written to a SAS data set that is used to create the header and footer for each report.

```
Array Fields (4) Field1 - Field4;
Start = 1;
Do J = 1 to 4; /* Evaluate each position of BinaryType */
  Linkref = Binarytype;  /* copy BinaryType*/
  Binary = Substr(BinaryType,Start,1); /* examine one position of field */
  If Binary eq '0' then
    do;
       substr(LinkRef,Start,1) = '1';  /* Flip to a 1 if originally a 0 */
       ReportType = 'Expand  ';
    end;
  else
    do;
       substr(LinkRef,Start,1) = '0';  /* Else Flip to a 0 */
       ReportType = 'Collapse';
    end;
  Field(J)  = '<a href=Type'||LinkRef||'.html> "'||ReportType||'" </a>';
  Start = Start + 1;
End;
```

Opening the data set generated in this step will show this content for each column.

```
'<a href=Type0100.html>"Expand" </a>';
```

As stated earlier, the browser opens the file named in the 'href=' after the double-quoted item is clicked, which is Expand in this example.

The data step that writes the header and footer values is very simple. It uses labels to create column names that match the columns of the report. It then tells ODS which variables will be in the table and writes the variables to the table using the default Put _ODS_ statement. That statement writes all variables at once, in the order defined in the File Print ODS statement.

```
Data _null_;
  Set HeaderFooter end=eof;
  Label Field1 = Year Field2 = Month Field3 = Store Field4 = Household;
  File Print ODS=(Variables=(Field1 - Field4));
  Put _ODS_;
run;
```

Each report HTML is constructed with three data steps. The first step writes the header table; the second, the report table; the third, the footer table. Writing to one HTML file three times requires the use of the 'mod' parameter on the file statement so that the tables are appended instead of overwritten. It also necessitates the use of the No_Top_Matter and No_Bottom_Matter options on the Body parameter of the ODS HTML statement to prevent HTML commands from being written to the file that would cause the browser to display separation lines between the tables.

```
Filename body "D:\Proj\ZDevelopment\Reports\Type0100.html" mod;
ODS HTML body=body (No_Top_Matter No_Bottom_Matter);
```

The ODS HTML statement specifies that HTML commands will be written to the filename referenced in the Body= parameter. The ODS HTML statement is also used to open and close the file, set the style template that will be used for the fonts in the file, and indicate whether frame or page files will be associated with the body file being written.

The three data steps that create the report HTMLs are placed into a macro that is executed once for every data set created for a unique BinaryType. In the example used in this paper, there are 16 such data sets.

The macro and the data to create the 'cube' are available from the author via email: thefoor@hotmail.com. That macro is capable of creating a cube with 1 to 10 dimensions.

Below are examples of the Table of Contents and one report created by the author's macro.

Household Sales and Visits Data of Ajax SuperMarkets

Table of Contents

Click Report to See That Level of the Cube

Report	Level	Year	Month	Store	Household
0000000000	0	Collapsed	Collapsed	Collapsed	Collapsed
0000000001	1	Collapsed	Collapsed	Collapsed	Expanded
0000000010	2	Collapsed	Collapsed	Expanded	Collapsed
0000000011	3	Collapsed	Collapsed	Expanded	Expanded
0000000100	4	Collapsed	Expanded	Collapsed	Collapsed
0000000101	5	Collapsed	Expanded	Collapsed	Expanded
0000000110	6	Collapsed	Expanded	Expanded	Collapsed
0000000111	7	Collapsed	Expanded	Expanded	Expanded
0000001000	8	Expanded	Collapsed	Collapsed	Collapsed
0000001001	9	Expanded	Collapsed	Collapsed	Expanded
0000001010	10	Expanded	Collapsed	Expanded	Collapsed
0000001011	11	Expanded	Collapsed	Expanded	Expanded
0000001100	12	Expanded	Expanded	Collapsed	Collapsed
0000001101	13	Expanded	Expanded	Collapsed	Expanded
0000001110	14	Expanded	Expanded	Expanded	Collapsed
0000001111	15	Expanded	Expanded	Expanded	Expanded

Household Sales and Visits Data of Ajax SuperMarkets

Report 0000001010

"Go to Table of Contents"

Year	Month	Store	Household
"Collapse"	"Expand "	"Collapse"	"Expand "

Level	Year	Month	Store	Household	Sales	Visits	AvgSales	AvgVisits	P50Sales	P50Visits
10	1999	Collapsed	7926	Collapsed	51,706	1,140	97.19	2.14	50	1
10	1999	Collapsed	7931	Collapsed	29,402	605	113.08	2.33	45	1
10	1999	Collapsed	7964	Collapsed	95,899	1,484	168.84	2.61	114	2
10	2000	Collapsed	7926	Collapsed	50,840	1,114	93.97	2.06	50	1
10	2000	Collapsed	7931	Collapsed	39,625	826	103.19	2.15	48	1
10	2000	Collapsed	7964	Collapsed	100,187	1,489	160.81	2.39	97	2

Year	Month	Store	Household
"Collapse"	"Expand "	"Collapse"	"Expand "

"Go to Table of Contents"

More information on ODS and on creating HTMLs with the SAS Data Step can be found in the Online Documentation and in these three papers available on the SAS website.

Heffner, W. F. (1998), "ODS: The Data Step Knows," in the *Proceedings for the Twenty-Third Annual SAS Users Group International Conference,* Cary, NC: SAS Institute Inc.

Olinger, C. R. (1999), "Twisty Little Passages, All Alike – ODS Templates Exposed" in the *Proceedings for the Twenty-Forth Annual SAS Users Group International Conference,* Cary, NC: SAS Institute Inc.

Olinger, C. R. (2000), "ODS for Dummies" in the *Proceedings for the Twenty-Fifth Annual SAS Users Group International Conference,* Cary, NC: SAS Institute Inc.

A nice primer on HTML commands, now available in many half-price bookstores, is

Evans, Tim, Sams Teach Yourself HTML 4 in 10 minutes, Sams Corporation, Indianapolis, Ind., 1998, 230 pages. Http://www.samspublishing.com/.

TRADEMARKS

Transforming Single-Record Spreadsheet Data into Multiple Observations

Glenda Garner, Wake Forest University

ABSTRACT

Generating SAS datasets from ASCII files is a simple task. Generating datasets from spreadsheet style data is also easily accomplished in SAS. However, the task becomes much more difficult when dealing with a mixed format of data variables with a varying number of input lines per record. One such example was the conversion process of biomechanical gait data into SAS datasets in the ADAPT study(Arthritis, Diet, and Activity Promotion Trial).
This poster will address implementing advanced features of the INPUTstatement.

RAW DATA EXAMPLE

TABLE 1

60030r1w1.xls		
Avg Step Width (cm)	10.81	
R_Velocity	102.007	
R_Stride_Len	122.392	
R_Cadence	98.63	
L_Velocity		
L_Stride_Len		
L_Cadence		
R_Support_Time	64.384	
L_Support_Time		
R_Non_Support	35.616	
L_Non_Support		
R_Step_Len	57.84	
L_Step_Len	64.552	
R_Dbl_Support	16.438	
L_Dbl_Support		
RHS 58	131	
LHS 94		
RTO 105	5	
LTO 70		
RHS FP 59		
LHS FP		
R_HIP Rot ANG	R_HIP Abd ANG	R_HIP Flex ANG
6.608	2.316	62.296
6.748	2.578	60.988

Table 1 shows a portion of the Biomechanical gait data for the ADAPT study. Each participant's data was contained in a separate spreadsheet. Within each spreadsheet, each record consisted of 23 rows of temporal/spatial data on the participant. The next group of rows consist of 87 columns of data with each row representing a time period in the gait cycle. Processing included exporting each spreadsheet from Excel into a CVS file (ASCII comma- delimited format) and then concatenating all the records into one file.

SAMPLE CODE

INPUT Line Features Used:

- Line pointer control, #, moves the pointer to the line number specified.
- Column pointer control, @, moves the pointer to the column specified.
- Line hold specifier, trailing @@, keeps the pointer on the current input line.
- Three input statements are executed:
  1. The first INPUT statement reads the first 23 lines
  2. The second INPUT statement tests for end of record
  3. The third INPUT statement is within the DO UNTIL loop, which creates an output line for each iteration. The loop is executed until end of record or end of file is reached.

Infile Options Used:

- lrecl the logical record length must be given or a default of 80 is used.
- dlm the delimiting character is the comma

TABLE 2

```
libname x '/home/pepper/adapt/gait/datasets/baseline';
filename raw '/home/pepper/adapt/gait/rawdata/baseline/gait.out';
options ls=80;

data one;
length id $ 11;
quit='n';
obnumber=0;
infile raw lrecl=600 dlm="," ;

input  id $       /* The first INPUT statement      */
#2 @20 avg_step
#3 @11 r_vel
#4 @13 r_slen
#5 @10 r_cad
#6 @11 l_vel
#7 @13 l_slen
#8 @11 l_cad
#9 @15 r_time
#10 @15 l_time
#11 @14 r_nsup
#12 @14 l_nsup
#13 @11 r_step
#14 @11 l_step
#15 @14 r_dbl
#16 @14 l_dbl
#17 @4 rhs @7 rhs2
#18 @4 lhs @7 lhs2
#19 @4 rto @7 rto2
#20 @4 lto @7 lto2
#21 @7 rhs_fp1
```

```
#22 @7 lhs_fp
#23  headers $ ;

do until (quit='y');
 input var1 @@;            /*The second  INPUT statement    */

if var1 ne . then do;
    input var2-var87;         /*The third INPUT statement       */
    obnumber = obnumber + 1;
    end;
else
   quit='y';
  output;
end;
```

Table 2 shows the code used in making each of these rows a record with the temporal/spatial data included in the observation.

SAMPLE OUTPUT DATA

TABLE 3

Obs	ID	AVG_STEP	R_VEL	R_SLEN
1	60030r1w1.x	10.81	102.007	122.392
2	60030r1w1.x	10.81	102.007	122.392
3	60030r1w1.x	10.81	102.007	122.392
4	60030r1w1.x	10.81	102.007	122.392

Obs	RTO2	LTO	Var1	Var2	Var3	Var4
1	5	70	6.608	2.316	62.296	-21.889
2	5	70	6.748	2.578	60.988	-21.312
3	5	70	6.556	2.632	59.652	-20.490
4	5	70	5.953	2.371	58.303	-19.358

Table 3 shows a portion of the first four records of the output. The temporal/spatial data is repeated for each observation. All variables are kept in the final dataset. The ID variable identifies the participant ID, the visit number, more affected side, and the trial number. These data will be analyzed to determine any significant differences or changes between or within the intervention groups.

CONCLUSION

This paper demonstrates how to write a SAS program to read ASCII data with a non-fixed format and a varying number of lines per record. Additionally shown is how to create output with multiple observations from one single record.

References
SAS Institute Inc(1990) SAS Language, Cary, NC: SAS Institute Inc, 420-421 pp.

Generating Matched Case Data Using PROC SQL

Imelda C. Go, Lexington County School District One, Lexington, SC

ABSTRACT

Some statistical methods compare data matched on certain variables. For example, pairs of subjects with the same gender are to be compared. Whenever data from males/females are to be matched with data from other males/females, a many-to-many match is involved. SAS users are discouraged from using the DATA step to perform many-to-many matches. PROC SQL can adequately handle such types of matches. The paper is an introduction on how to use PROC SQL for the said purpose whether data are to be matched on one or more variables/criteria.

DATA STEP

Consider two data sets, `group1` and `group2`. In the examples, SS stands for scaled score.

group1 data set

GRADE	NAME	SS	LUNCH	SEX
4	Garbo, Greta	434	R	F
3	Davis, Betty	380	F	F
3	Monroe, Marilyn	324	F	F
5	Gabor, Eva	567	R	F
2	Taylor, Liz	245	N	F
6	Farrow, Mia	655	N	F

group2 data set

GRADE	NAME	SS	LUNCH
3	Midler, Bette	354	R
3	Gabor, Zsazsa	381	R
6	Seymour, Jane	656	F
4	Field, Sally	434	F
5	Loren, Sophia	577	F
2	Ryan, Meg	234	N

A DATA step can be used to merge `group1` and `group2` by `grade`. When data are merged this way, the two data sets need to be sorted by `grade` (or the BY-variables). When contributing data sets have variables with the same name, the variables need to be renamed in order to prevent the values of one data set from overwriting the values of the other data set during the merge.

The DATA step can handle one-to-one, one-to-many, and many-to-one matches but not many-to-many matches. For true many-to-many matches, the result should be a cross product. For example, if two records from one data set match two records from another data set, the merged results should have $2 \times 2 = 4$ records.

The results shown next are problematic because of the many-to-many situation and because the variables with the same name were not renamed or dropped from either data set.

```
proc sort data=group1; by grade;
proc sort data=group2; by grade;

data test;
     merge group1 group2;
     by grade;

proc print;
```

GRADE	NAME	SS	LUNCH	SEX
2	Ryan, Meg	234	N	F
3	Midler, Bette	354	R	F
3	Gabor, Zsazsa	381	R	F
4	Field, Sally	434	F	F
5	Loren, Sophia	577	F	F
6	Seymour, Jane	656	F	F

PROC SQL

Many-to-Many Matches

PROC SQL can correctly handle many-to-many matches. Use it to create a data set that involves all possible matches. PROC SQL can also perform, under one procedure, what could be done using a combination of DATA step statements, and the PRINT, SORT, and SUMMARY procedures. The procedure is powerful, but only a few of its features are mentioned in this paper.

- The SELECT statement allows users to specify columns for the query, create column aliases (name/rename variables), and compute arithmetic expressions.
- The FROM statement specifies the sources of the variables listed in the SELECT statement.
- The WHERE statement specifies subsetting criteria.
- The ORDER BY specifies the sort order.
- *Tables* or data sets do not need to be sorted to use PROC SQL. The RUN statement does not need to be used with PROC SQL.

The example below matches data sets `group1` and `group2` by `grade` using PROC SQL. Cross products resulted for many-to-many match situations.

```
proc sql;
 select group1.grade, group1.name as name1,
        group2.name as name2, group1.ss as ss1,
        group2.ss as ss2, sex
   from group1, group2
  where group1.grade=group2.grade;
```

GRADE	NAME1	NAME2	SS1	SS2	SEX
4	Garbo, Greta	Field, Sally	434	434	F
3	Davis, Betty	Midler, Bette	380	354	F
3	Davis, Betty	Gabor, Zsazsa	380	381	F
3	Monroe, Marilyn	Midler, Bette	324	354	F
3	Monroe, Marilyn	Gabor, Zsazsa	324	381	F
5	Gabor, Eva	Loren, Sophia	567	577	F
2	Taylor, Liz	Ryan, Meg	245	234	F
6	Farrow, Mia	Seymour, Jane	655	656	F

Names repeat in the `name1` and `name2` fields. In producing the final matched case sample, care must be given to make sure that there are no unwanted repetitions in the matches. This can be achieved with more SAS programming statements but will not be shown in this paper.

Syntax Review

The PROC SQL example above is repeated below. Some syntax notes are provided.

```
proc sql;
 select group1.grade, group1.name as name1,
        group2.name as name2, group1.ss as ss1,
        group2.ss as ss2, sex
   from group1, group2
  where group1.grade=group2.grade;
```

- Variables in the SELECT statement are separated by commas and can be specified using one-level or two-level names. Two-level names are of the form:
  `<data set name>.<variable name>`
  and can be used any time. One-level names are acceptable when there are no ambiguous references. `Group1.grade` refers to the `grade` variable from the `group1` data set. `Sex` uses a one-level name because it is in `group1` but not in `group2`.

- Variables can be renamed in the SELECT statement. `Group1.name` is renamed as `name1`.

- The FROM statement lists source data sets separated by commas. Data sets `group1` and `group2` contain the variables listed in the SELECT statement.

- The WHERE statement can contain subsetting criteria. The condition `group1.grade=group2.grade` restricts the results to data *joined* or matched with equal grade values. The results will not contain records that differ in grade values.

- `Grade` occurs as a variable in both data sets. The following PROC SQL example produces the error: "`Ambiguous reference, column GRADE is in more than one table.`"

```
proc sql;
 select grade, group1.name, group2.name
   from group1, group2
  where group1.grade=group2.grade;
```

- To include all variables from a table, use `SELECT *` instead of a variable list after the SELECT key word.

```
proc sql;
 select *
   from group1
  where grade=3;
```

```
GRADE  NAME                 SS  LUNCH  SEX
-----------------------------------------------
    3  Davis, Betty        380  F      F
    3  Monroe, Marilyn     324  F      F
```

More Matching Criteria #1

Building on the first PROC SQL example, suppose that the matches are to be restricted to those where the scaled scores differ by no more than two scaled score points. An additional criteria is added to the WHERE statement to produce the desired result. The `0<=abs(group1.ss-group2.ss)<=2` condition checks to see if the absolute difference of the scaled scores is not a missing value and is at most two. (For the contrived data, the results below are very convenient because there are no repetitions of students. There are only distinct names from `group1` and `group2`.)

```
proc sql;
 select group1.grade, group1.name as name1,
        group2.name as name2, group1.ss as ss1,
        group2.ss as ss2, sex
   from group1, group2
  where group1.grade=group2.grade and
        0<=abs(group1.ss-group2.ss)<=2;
```

```
GRADE  NAME1           NAME2                SS1       SS2  SEX
-----------------------------------------------------------------
    4  Garbo, Greta    Field, Sally         434       434  F
    3  Davis, Betty    Gabor, Zsazsa        380       381  F
    6  Farrow, Mia     Seymour, Jane        655       656  F
```

Creating a Table or an Output Data Set

The next example uses the CREATE TABLE statement to store the results of the query into a *table*. An ORDER BY statement was also added to produce the results in increasing `ss2` values. The log shows the following message after the SQL statement, "`NOTE: Table WORK.POOL created, with 3 rows and 6 columns.`"

```
proc sql;
 create table pool as
 select group1.grade, group1.name as name1,
        group2.name as name2, group1.ss as ss1,
        group2.ss as ss2, sex
   from group1, group2
  where group1.grade=group2.grade and
        abs(group1.ss-group2.ss)<=2
  order by ss2;

proc print data=pool;
```

```
OBS   GRADE     NAME1          NAME2       SS1  SS2  SEX

 1      3     Davis, Betty  Gabor, Zsazsa  380  381   F
 2      4     Garbo, Greta  Field, Sally   434  434   F
 3      6     Farrow, Mia   Seymour, Jane  655  656   F
```

The above example also shows that when PROC PRINT is used for the `pool` table, the output shows the query results.

More Matching Criteria #2

Suppose the `lunch` status needs to be part of the matching criteria. The condition can be added to the WHERE statement. The `lunch` variables were used in the WHERE statement even if they were not among the variables in the SELECT statement below. The `lunch` variable also does not appear in the results.

```
proc sql;
 select group1.grade, group1.name as name1,
        group2.name as name2, group1.ss as ss1,
        group2.ss as ss2, sex
   from group1, group2
  where group1.grade=group2.grade and
        group1.lunch=group2.lunch;
```

```
GRADE  NAME1          NAME2              SS1       SS2  SEX
-------------------------------------------------------------
    2  Taylor, Liz     Ryan, Meg          245       234  F
```

To include the `lunch` variable in the output, add the `lunch` variable to the variables listed in the SELECT statement.

```
proc sql;
 select group1.grade, group1.name as name1,
        group2.name as name2, group1.ss as ss1,
        group2.ss as ss2, sex, group1.lunch
   from group1, group2
  where group1.grade=group2.grade and
        group1.lunch=group2.lunch;
```

```
GRADE  NAME1          NAME2              SS1     SS2  SEX  LUNCH
-------------------------------------------------------------
    2  Taylor, Liz    Ryan, Meg          245     234  F    N
```

Arithmetic Expression

Suppose the difference of the `ss` variables needs to be added to the query. Add the necessary code to the SELECT statement. The difference of `group1.ss-group2.ss` is included in the SELECT statement below. The difference is given in the last column without a column heading.

```
proc sql;
 select group1.grade,group1.name as name1,
        group2.name as name2, group1.ss as ss1,
        group2.ss as ss2, sex,
        group1.ss-group2.ss
   from group1, group2
  where group1.grade=group2.grade and
        group1.lunch=group2.lunch;
```

```
GRADE  NAME1          NAME2          SS1   SS2  SEX
-------------------------------------------------------------
    2  Taylor, Liz     Ryan, Meg      245   234  F          11
```

Column Alias

A name can be given to the difference. For example, adding `as ssdiff` immediately after `group1.ss-group2.ss` in the previous SQL statement will name the difference as `ssdiff`.

```
proc sql;
 select group1.grade, group1.name as name1,
        group2.name as name2, group1.ss as ss1,
        group2.ss as ss2, sex,
        group1.ss-group2.ss as ssdiff
   from group1, group2
  where group1.grade=group2.grade and
        group1.lunch=group2.lunch;
```

```
GRADE  NAME1          NAME2          SS1   SS2  SEX    SSDIFF
-------------------------------------------------------------
    2  Taylor, Liz     Ryan, Meg      245   234  F          11
```

A Final Note

It is also possible to have several statements under the same procedure declaration.

```
proc sql;
  create …;
proc sql;
  select …;
proc sql;
  select …;
```

can be replaced by

```
proc sql;
  create …;
  select …;
  select …;
```

REFERENCES

SAS Institute Inc., *SAS® Language Reference, Version 8*, Cary, NC: SAS Institute Inc., 1999. 1256 pp.
SAS Institute Inc., *SAS OnLineDoc®, Version 8*, Cary, NC: SAS Institute Inc., 1999.

TRADEMARK NOTICE

Imelda C. Go (icgo@juno.com)
Lexington County School District One
Lexington, SC

Overcoming the Challenges of Longitudinal Data Collection

Imelda C. Go, Lexington County School District One, Lexington, SC

ABSTRACT

This paper discusses problems encountered when data are collected longitudinally. Longitudinal data collection implies data collected over time are to be matched on key variables (e.g., social security number, last name, first name). Data for the key variables are not always consistent due to a variety of reasons (e.g., human error, name changes). This lack of consistency creates problems during the matching process. The paper also discusses matching data using the DATA step versus PROC SQL.

INTRODUCTION

Using SAS to produce longitudinally matched data involves understanding several aspects of how SAS processes data. There are also various SAS programming solutions that produce the same results. The paper does not go into great detail regarding all these aspects. However, it provides an overview of things that need to be considered in producing longitudinal data. Most examples are actual situations encountered with school test data.

Some of the things that need to be considered include:

- Determining the quality of the data

  The data quality affects the data processing and the programming required to produce the longitudinal data.

- Determining the matching algorithm

  Can the data be matched on social security number (SSN)? The answer will depend on how reliable the SSN data are and what the project's tolerance for error is. What other sets of variables can the data be matched on? Should the leftover data that did not match stay unmatched? Should they be matched using other criteria or even manually inspected to see if more matches can be found? For certain studies, matched case data may already be hard to find so a manual inspection may be worth the trouble.

- Determining the project's tolerance for error

  The tolerance for error is usually determined by the consequences of an incorrect match. The SSN may have incorrect digits and may accidentally match another SSN. If the SSN is expected to be problematic, consider matching with the SSN and other variables. If the tolerance for error is on the high end, perhaps less stringent or even fuzzy matching methods can be used.

The rest of the paper is divided into four parts:

- Data quality
- Matching on key variables
- DATA step
- PROC SQL

DATA QUALITY

Input data can come from a variety of sources and it is not always possible to control how they are created and their quality. High quality data naturally require less troubleshooting and programming. Some strategies for preventing data processing problems are:

- Establish data entry standards. For example, how should name suffixes (e.g., Jr., II, III) be entered as data? Should there be another field specifically for name suffixes? Consistent data facilitates programming.

- Prevent invalid values from being entered as data. For example, use lookup tables that restrict values of variables to valid values only.

- Validate the data prior to processing. The quality of the data can be gauged by validating the data. For example, does the data set contain what it is expected to contain? If it is supposed to have data for 5-year old students, but it contains data for 10-year old students, there is something obviously wrong.

- Plan the data's structure to be compatible with the uses of the data. For data used in applications that rely heavily on street address data (e.g., transportation routing, mapping, geographic information systems), it might be useful to decompose a street address into a number of variables (street number, street name, street type, and apartment number) instead of using one variable to represent it.

- Know the data. Be aware of the sources of data, sources of error, and the quality of the data. Poor data quality often interferes with data processing and is a major cause of processing delays. It not only creates problems during processing, but it casts doubt on the results.

MATCHING ON KEY VARIABLES

Matching can be done on one or more key variables. When only one variable is used, a non-match implies the values on that one variable were not the same. If two variables are used for the match, a non-match implies the values differed on one or both variables. As more variables are used, the matching criteria become more stringent. Matches must be equal on more variables and differences between those variables are more likely to occur than if fewer variables were used.

When a unique identifier, such as the SSN, is used to match, make sure the SSN data type is the same. It is either numeric or character across all data sets that need to be matched on SSN.

Would there even be a situation where the SSN should be a character type? The answer depends on the nature of the data. If one or more digits of the SSN are missing and a space is used to indicate the missing digit(s), the character type would preserve the information. If the SSN of '251112222' is bubbled on a scan form and some of the bubble marks were not scanned properly, the data file might have '25?12222' where ? is a digit that did not scan properly. If the data type is numeric, '251 12 22' (4th and 7th digits are missing) and '25?12222' are invalid numeric values and are set to missing.

When data are matched on character variables (e.g., last name, first name), make sure the character values are as consistent as possible. Reducing inconsistencies in character values can help maximize the number of successful matches. However, people can also change their names over time.

Comparing strings is case-sensitive. That is, `'Edward'` is not the same as `'EDWARD'`. Typing is subject to human error. For example, `'EDWARD'` may have been typed as `'EDWard'`. One solution is to use the UPCASE or LOWCASE functions.

sample statement	value of test
test=upcase('Edward');	'EDWARD'
test=lowcase('EDWard');	'edward'

`'Mary Ann'` might be entered as `'Mary  Ann'` where there are two spaces between `Mary` and `Ann`. Use the COMPBL function to convert two or more consecutive blanks into one blank.

```
test=compbl(name);
```

value of name	value of test
'Mary Ann'	'Mary Ann'
'Mary Ann'	'Mary Ann'

`'Mary Ann'` might be entered as `' Mary Ann'` where there is a leading space before `Mary`. Use the LEFT function to left align a character expression.

```
test=left(name);
```

value of name	value of test
' Mary Ann'	'Mary Ann'
' Mary Ann'	'Mary Ann'

SAS ignores trailing spaces when character expressions are compared (e.g., `'Mary Ann  '` is equivalent to `'Mary Ann')`. If there are any unwanted spaces in the first name, use the COMPRESS function to remove specified characters from a string. If no characters for removal are specified, the function removes spaces by default.

```
test=compress(name);
```

value of name	value of test
'Edw ard'	'Edward'
' Edward'	'Edward'

Is it possible to have a first name with two words? Whatever the answer is, consistency is the key word when matching. If all the spaces are removed from all the variables used to match, then the only issue is whether the original name value needs to be retained. If there is any such concern, keep two name fields. The first one would be the original name, the second one would be the edited name used for matching.

Other characters may also need to be removed. List the characters to be removed in single quotes.

```
test=compress(name,'?-');
```

value of name	value of test
'Kidman-Cruise?'	'KidmanCruise'

There may be an attempt to type accent marks into the value of a name. For example, *André* might be typed in as `Andre'` and this is really a data entry standard issue. When the typist encounters an accent mark, should they attempt to type the accent mark? That particular accent mark lends itself well to an apostrophe. There are of course other foreign accent marks (e.g., ö) that would not lend themselves well to plain text characters.

What if the apostrophe itself needs to be eliminated? Use the following syntax.

```
test=compress(name,'''');
```

value of name	value of test
Andre'	Andre

Should non-letter characters be eliminated? The answer depends on the situation. An actual example encountered is the accidental or intentional typing of non-letter characters in names on student databases. Student names are sent to a testing company that creates bar-coded labels for test documents. In the process of creating the labels, the testing company eliminates all non-letter characters from the student names. Not all students will have documents with bar-coded labels during testing. Such students indicate their names by bubbling a scan form. Scan forms often only provide bubbles for letter characters in name fields. If data on the student database need to be matched to data received from the testing company later, original names need to be matched with the names from the testing company. The first set of names has non-letter characters while the second set of names has no such characters. The number of matches would not be maximized if the non-letter characters are not edited out of the original names prior to matching.

There are many data sources and data entry standards. Errors may also occur during the import and export of data between various people and computer systems. Files can get corrupted. A programmer might accidentally leave out the last digit of the SSN being written out to a raw data file, etc.

DATA STEP

Consider two data sets, `time1` and `time2`. In the examples, SS stands for scaled score.

time1 data set

GRADE	LAST	FIRST	SS	LUNCH	SSN
4	Garbo	Greta	434	R	111111111
3	Davis	Betty	380	R	222222222
2	Taylor	Liz	245	R	333333333
9	Kidman	Nicole	333	R	444444444

time2 data set

GRADE	LAST	FIRST	SS	LUNCH	SSN
5	Garbo	Greta	533	F	111111111
4	Davis	Betty	493	F	222222222
3	Taylor	Liz	399	F	333333333
8	Loren	Sophia	723	F	555555555

Using a DATA step to merge `time1` and `time2` by `ssn` has disadvantages. When data are merged this way, the two data sets need to be sorted by `ssn` (or the BY-variables). When contributing data sets have variables with the same name, the variables need to be renamed in order to prevent the values of one data set from overwriting the values of the other data set during the merge.

The DATA step can handle one-to-one, one-to-many, and many-to-one matching but not many-to-many matching. For true many-to-many matches, the result should be a cross product. For example, if there are two records that match from one contributing data set to two records from the other, the result should have $2 \times 2 = 4$ records in the merged data set.

The merge results shown below are problematic because of the many-to-many situation and because the variables with the same name were not renamed or dropped from either data set.

```
proc sort data=time1; by ssn;
proc sort data=time2; by ssn;

data test; merge time1 time2; by ssn;

proc print;
```

OBS	GRADE	LAST	FIRST	SS	LUNCH	SSN
1	5	Garbo	Greta	533	F	111111111
2	4	Davis	Betty	493	F	222222222
3	3	Taylor	Liz	399	F	333333333
4	9	Kidman	Nicole	333	R	444444444
5	8	Loren	Sophia	723	F	555555555

The following data step shows some variables being dropped and renamed prior to merging. The resulting data set has correct values.

```
data test;
merge time1 (drop=grade lunch rename=(ss=ss1))
      time2 (drop=grade lunch rename=(ss=ss2));
by ssn;
```

OBS	LAST	FIRST	SS1	SSN	SS2
1	Garbo	Greta	434	111111111	533
2	Davis	Betty	380	222222222	493
3	Taylor	Liz	245	333333333	399
4	Kidman	Nicole	333	444444444	.
5	Loren	Sophia	.	555555555	723

What happens if the BY statement is *accidentally* omitted from the previous example? No error message will be given because it is valid SAS syntax and SAS does merges without BY statements. The records are merged in the order in which they occur on the data set and without regard to any other criteria. The resulting data are invalid and shown below.

```
data test;
merge time1 (drop=grade lunch rename=(ss=ss1))
      time2 (drop=grade lunch rename=(ss=ss2));
```

OBS	LAST	FIRST	SS1	SSN	SS2
1	Garbo	Greta	434	111111111	533
2	Davis	Betty	380	222222222	493
3	Taylor	Liz	245	333333333	399
4	Loren	Sophia	333	555555555	723

The DATA step can handle one-to-many and many-to-one matches properly. If there is 1 record from the first source data set and there are 2 records from the second source data set for the same `ssn`, then the resulting data set will have 2 records. If the ultimate goal is to have only one record per `ssn`, then make sure there is only one record per `ssn` prior to the merge, or consolidate the multiple records into one record after the merge. If multiple records per `ssn` are invalid at any time, then that needs to be addressed during data validation.

Is there a way to determine which data set contributed to the resulting merged record? Use the IN= data set option. The `in1` variable is a 0/1 indicator of whether the `time1` data set contributed to the current observation (`in1=0` if it did not and `in1=1` if it did). The `in2` variable is the 0/1 indicator with respect to the `time2` data set.

```
data test;
merge time1 (in=in1 drop=grade lunch
               rename=(ss=ss1))
      time2 (in=in2 drop=grade lunch
               rename=(ss=ss2)); by ssn;
      if in1 and in2 then source='both ';
         else if in1 then source='time1';
         else if in2 then source='time2';
```

OBS	LAST	FIRST	SS1	SSN	SS2	SOURCE
1	Garbo	Greta	434	111111111	533	both
2	Davis	Betty	380	222222222	493	both
3	Taylor	Liz	245	333333333	399	both
4	Kidman	Nicole	333	444444444	.	time1
5	Loren	Sophia	.	555555555	723	time2

PROC SQL

Consider another two data sets, `time1` and `time2` below.

time1 data set

GRADE	LAST	FIRST	SS	LUNCH	SSN
4	Garbo	Greta	434	R	111111111
2	Taylor	Liz	245	R	333333333
2	Taylor	Liza	232	R	333333333
9	Kidman	Nicole	333	R	444444444

time2 data set

GRADE	LAST	FIRST	SS	LUNCH	SSN
5	Garbo	Greta	533	F	111111111
3	Taylor	Liz	399	F	333333333
3	Monroe	Marilyn	387	F	333333333

The following PROC SQL example will produce the correct results when many-to-many matches are involved. PROC SQL would produce all the possible matches while the DATA step would not. PROC SQL is a powerful procedure and can also perform, under one procedure, what could be done using a combination of DATA step statements, and the PRINT, SORT, and SUMMARY procedures.

```
proc sql;
   select time1.ss as ss1,
          time2.ss as ss2,
          time1.last as last1,
          time2.last as last2,
          time1.first as first1,
          time2.first as first2
     from time1,time2
    where time1.ssn=time2.ssn
 order by ss1, ss2;
```

SS1	SS2	LAST1	LAST2	FIRST1	FIRST2
232	387	Taylor	Monroe	Liza	Marilyn
232	399	Taylor	Taylor	Liza	Liz
245	387	Taylor	Monroe	Liz	Marilyn
245	399	Taylor	Taylor	Liz	Liz
434	533	Garbo	Garbo	Greta	Greta

Suppose there is a need to limit the results to those matches that show a greater than 150-point gain in scaled score points from *time 1* to *time 2*. The WHERE statement is expanded to include the criterion `time2.ss-time1.ss>150`. This difference is also computed as `ssdiff` and included in the query by adding `time2.ss-time1.ss as ssdiff` to the SELECT statement.

```
proc sql;
 select time1.ss as ss1,
        time2.ss as ss2,
        time2.ss-time1.ss as ssdiff,
        time1.last as last1,
        time2.last as last2,
        time1.first as first1,
        time2.first as first2
   from time1,time2
  where time1.ssn=time2.ssn and
        time2.ss-time1.ss>150;
```

SS1	SS2	SSDIFF	LAST1	LAST2	FIRST1	FIRST2
245	399	154	Taylor	Taylor	Liz	Liz
232	399	167	Taylor	Taylor	Liza	Liz
232	387	155	Taylor	Monroe	Liza	Marilyn

The data above were matched on `ssn`. Examining names for records with the same `ssn` value from both data sets show differences (`Liza Taylor` vs. `Liz Taylor`, and `Liza Taylor` vs. `Marilyn Monroe`). If the `ssn` values are correct and unique, then perhaps the name values are incorrect, or legal name changes were involved, or aliases were used. If the `ssn` values were incorrect, then the example illustrates a disadvantage of using only one variable to match data. If the `ssn` values are not considered very reliable, using more variables than `ssn` alone to verify an individual's identity can prevent incorrect matches.

Regarding the plausibility of two individuals with practically the same name having almost the same SSN, examination of large databases with student biographical data revealed identical twins with the same sex, the same birth date, almost identical names (e.g., Sierra & Tierra), and almost the same SSN (e.g., consecutive numbers -- 222-22-2212 and 222-22-2213).

CONCLUSION

An understanding of data quality issues, sources of error, how SAS processes data, and how things can go wrong during the matching process are essential to anticipating problems that need to be addressed during longitudinal data collection. Being unaware of how problems can occur could potentially allow such problems to occur undetected and eventually invalidate the data.

REFERENCES

SAS Institute Inc., *SAS® Language Reference, Version 8*, Cary, NC: SAS Institute Inc., 1999. 1256 pp.
SAS Institute Inc., *SAS OnLineDoc®, Version 8*, Cary, NC: SAS Institute Inc., 1999.

TRADEMARK NOTICE

Imelda C. Go (icgo@juno.com)
Lexington County School District One
Lexington, SC

Defining Test Data Using Population Analysis

Clarence Wm. Jackson, CQA - City of Dallas CIS

Abstract

Defining test data that provides complete test case coverage requires the tester to accumulate data that will satisfy all of the test cases in the requirements. Having access to production data eases the accumulation somewhat, but then any smart tester knows that testing should not be done in the production data. The next best situation is to completely copy production data, but this can be expensive depending on the size of the files and the amount of testing being committed. Another option would be to build test data based on the requirements, which may or may not match the varied data found in production data. This paper proposes using population analysis of production data as a tool to defining test data coverage. The tools available to provide statistical analysis, such as frequency of data element values and percentages to the total population will provide measurable verification of the test data to the production data, ensuring better results of testing scenarios.

Introduction

Test data is very important in ensuring that software performs as required. The testers using software specifications usually create test data, which usually means that the software will meet the specifications. This approach may not test the total requirements of the software or how the software will be used in production. If the tester has no knowledge of the data and how it's used, vital requirements can be missed in testing. Since the test data is an important link in the software testing cycle, ways to ensure that the test data is adequate should be explored. Population Analysis of production data will reduce the risk of missing critical data types, and increase the reliability of the software when moved to production.

Population Analysis

Population Analysis is simply creating reports that describe the type, frequency, and characteristics of data used by the application being tested. These reports will help in locating data and in defining the ranges and limits of data elements, lengths of fields, and the number of data values within each data element. The benefits of using Population Analysis are many, but the major benefit is to the tester in the following ways:

- Identification of codes and values being used in production which were not indicated in the specification or requirement document
- Unusual data conditions, like special codes, trigger values
- Provides a model for use in creating test transactions and test cases
- Provides a model for the type and frequency of transactions that should be created
- Helps identify incorrect transactions for testing error processing
- Documentation for use later in the project life cycle

There are three types of population analysis that can be performed to gather identifying information for testing. All the information gathered from the analysis should be documented. The three types of population analysis are:

1. File/data stores - to identify files and other input/output used by the software being tested
2. Screen - to identify screen displays that will be used or generated by the software
3. Field/data element - to identify characteristics and frequencies of fields/data elements

Usually the file/data stores are known, and the screens are straightforward. Of the three types, the Field/data element analysis is the most complex and time-consuming part of population analysis. That's where SAS comes in. Population analysis is best performed using software, and SAS can deal with any type of data encountered in any shop. SAS is designed for population analysis, and can compute all the statistics needed. There are so many PROCedures that will give you the information that it is a natural tool for population analysis. Other software that will provide analysis information are Audit Software, DataBase utilities, and CASE tools. But for field/data element analysis, SAS is the choice.

The steps for population analysis as presented in this paper are based on the Deming/Shewhart PDCA Quality Circle for continuous process improvement, and the Quality Assurance Institute's (QAI) process "WORKBENCH" model. The steps are:

1. **Define the inputs for analysis (Plan)**
- Existing production files being used "as is"
- Existing production files for which there will be minor changes to the file
- Production files that contain the same or similar fields/data elements that will be included
- Existing manual files, such as paper invoices, forms, etc.
- Files from other systems that contain the same or similar data elements

2. **Implement procedures (Do)**
- Identify file/data stores location, medium, organization of data, and other details
- Identify the record types on the file/data stores, number of records, and sizes
- Document the file/data stores characteristics
- Identify each data element/field in the file/data stores
- Document the characteristics of each data element
- Analyze the population of data recorded in this data element area
- Identify and document abnormal conditions

3. **Check procedures (Check)**
4. **Produce deliverables (Act)**

Before beginning, define what information (deliverable) is required from the analysis, create forms or other documentation templates. The documentation requirements for the three types of population analysis should be established in the planning stage. The documentation will assist you in evaluating the completeness of your analysis, and allow you to measure your progress.

Implement Procedures

The major task is to identify the files that will be used for your analysis. Files and data stores must be located and defined. Flowcharts, data flow diagrams, system charts, and other documents can be used to identify files within the scope of the testing project.

Field/data element analysis is the most complex of the analysis of population data. The analysis of the data is the most important, and time spent in this analysis will have a direct payback in improving the test data. The objective of this type of analysis is to describe the type, frequency, and characteristics of the date elements. It is this analysis that will be used to produce the test data.

The analysis of the data should include the frequency, maximum, minimum, percentage of the population, and total values for each data element in each file/data store. This information should be documented.

Check Procedures

You should have mostly everything needed at this point, but it's a good time to review. Check your progress using your documentation template. Are you getting the information needed to perform a good test, and create good test data? Review abnormal conditions with those involved with the data for verification. All abnormal conditions should be tracked and resolved.

Produce Deliverables

The deliverables are the documentation which describe the population of the files, screens, and data elements. With the correct analysis, a question concerning the percentage of records having certain values is known and documented. The number of values within a given data element is known and documented. You now know what data values are required, and a much better set of tests can be performed. With this information, the test project should proceed with more assurance that the correct items will be tested. The documentation can be used for any of the following purposes:

- Supplement the requirements and specification documentation
- Development of the test plan
- Creation of test cases and scripts
- Creation of the test data
- Stress testing

How SAS Can Be Utilized To Perform Population Analysis

SAS can be used in many ways to provide for all types of population analysis reports through the many tools that are a part of the SAS System. SAS can be used for every type, or just for the more labor-intensive analysis. In some cases, part of the file/data stores, screen, and data element analysis may be known, and provided to the tester. However, the analysis of the data element values should still be performed, and using SAS to do it will save many hours of manual checklisting. Also, SAS can be used to store the results of each type of analysis and report it.

For data element analysis, SAS will be able to read the data, and using your favorite PROC, such as FREQ, MEANS, TABULATE, etc, you can produce a listing of the frequencies of any and all data elements that are of concern. SAS can also load the test data stores using the values from the analysis, with a little coding using the DATA step programming statements.

Screen analysis can also be done using SAS, especially if it is in some format that can be read directly. Screens are such that tools may only be required to store the data from analysis.

The files and data stores can be analyzed either manually or using SAS. On the mainframe, PROC SOURCE, PDS, and other methods of getting file information from the operating system will give you plenty of information about the files. If you are working with SAS data, PROC DATASETS, CONTENT, and other SAS PROCedures will provide everything from file information to data element metadata. For instance, PROC CONTENT will provide you with most of the identification information from SAS data files regarding each data element.

Conclusion

The use of population analysis for testing is a valuable tool in ensuring that software will function properly when moved to production. The best source of data for testing is production data. Why? It is data that is being used in the live process. To analyze the production data for the purpose of building test data is a sound method to verify software. Population analysis is a method to avoid the risk of missing obvious transaction types during the testing phase. The benefits of using the method are many.

SAS offers many tools that can be used to perform population analysis of production data. SAS can readily compute mean, median, mode, standard deviation, and other statistics by using a few BASE SAS PROCedures for field/data element population analysis.

References

Jackson, Clarence Wm., *Using SAS To Help Manage Data* Proceedings of the Sixth Annual South Central Regional SAS Users Group Conference, 1996

Perry, William E., *A Structured Approach to Software Testing,* Q.E.D Information Sciences, Inc., Wellesley, MA, 1983

Quality Assurance Inst., *QA/QC Solutions – Population Analysis: Creating Test Data From Production Data,* The Process Warehouse, Quality Assurance Inst (QAI), Orlando, FL, 1997

Beizer, Boris, *Software Testing and Quality Assurance,* Van Nostrand Reinhold, NY, NY, 1984

The sample SAS program code used to support this paper is located in the Posters Area. The author can be contacted via email as follows:

Clarence Wm. Jackson, CQA
QA Change Manager
City of Dallas
Communication and Information Services
QA Change Management
1500 Marilla 4DS
Dallas, TX 75201
cljacks@ci.dallas.tx.us (City of Dallas)
CJac@compuserve.com (home)
http://ourworld.compserve.com/homepages/CJac

SAMPLE PROJECT

Test Project involves Salary and Jobcode changes for Company A

SAMPLE WORKSHEET #1
FILE/SUBSCHEMA POPULATION ANALYSIS

1. FILE/SUBSCHEMA IDENTIFIER:
2. FILE/SUBSCHEMA NAME:
3. RECORD TYPE(S)
4. IDENTIFIER FIELD
   a. Yes No
   b. NAME
5. VOLUME OF RECORDS (COUNT)
6. DESCRIPTION
7. VOLUME DENSITY OF RECORDS ON FILE:
   a. Average: Maximum: Minimum:
8. FILE MEDIA
9. FILE ORGANIZATION
10. FILE OWNER
11. SECURITY CLASSIFICATION
12. DATE LAST CREATED/MODIFIED
13. DAYS SAVED
14. STORAGE LOCATION

SAMPLE WORKSHEET #2
FIELD/DATA ELEMENT POPULATION ANALYSIS

1. FIELD IDENTIFIER
2. FIELD NAME
3. FIELD TYPE
   a. Numeric
   b. Alphabetic
   c. Alphanumeric
   d. Special Symbols
   e. Other (please specify)
4. FIELD DATA DICTIONARY DESCRIPTION ATTACHED? Yes No
5. FIELD OWNER

APPLICABLE	FIELD POPULATION ANALYSIS
Yes No_____	1. VOLUME/FREQUENCY OF USE
Yes No_____	2. NUMBER ALL BLANKS IN FIELD
Yes No_____	3. NUMBER ALL ZEROS IN FIELD
Yes No_____	4. HIGH VALUE
Yes No_____	5. LOW VALUE
Yes No_____	6. MEAN VALUE
Yes No_____	7. MEDIAN VALUE
Yes No_____	8. MODE VALUE
Yes No_____	9. STANDARD DEVIATION
Yes No_____	10. CODES FREQUENCY
Yes No_____	11. MAXIMUM NUMBER NONBLANK ZEROS FOUND IN FIELD
Yes No_____	**12.** MINIMUM NUMBER NONBLANK ZEROS FOUND IN FIELD

Data to complete the worksheets are in **BOLD** in the following SAS log and output.

SAS LOG

```
NOTE: Copyright (c) 1989-1996 by SAS Institute
Inc., Cary, NC, USA.
NOTE: SAS (r) Proprietary Software Release 6.12
TS020
      Licensed to CITY OF DALLAS, Site 0001457002.

NOTE: AUTOEXEC processing beginning; file is
C:\SAS\AUTOEXEC.SAS.
NOTE: AUTOEXEC processing completed.

1
2    * Sample program to get basic Population
Analysis data.
3      The data used is the SAS supplied sample
data "SALARY".
4
5      The Questions for analysis relates to
6            (1) retention rates for employees
7            (2) pay rates
8                a. by job type
9                b. by rate groups
10
11     Clarence Wm. Jackson, CQA
12   ;
13
14   title1 "SSU 2001, A Joint Conference of SCSUG
and SESUG";
15   title2 "Sample Set of Reports to Support
Population Analysis";
16
17   proc format;
18    value salrange
19      0 - 25000       = "$25,000 or Less   "
20      25001 - 50000   = "$25,001 - 50,000  "
21      50001 - 80000   = "$50,001 - 80,000  "
22      80001 - 110000  = "$80,001 - 110,000 "
23      110001 - 200000 = "$110,001 - 200,000"
24      200001 - 250000 = "$200,001 - 250,000"
25      250001 - HIGH   = "$250,001 Up       "
26      other           = "Not Paid - Error  ";
NOTE: Format SALRANGE has been output.
27   run;
NOTE: The PROCEDURE FORMAT used 0.44 seconds.
28
29   title4 "The SAS LOG Will Provides Information
About The File";
30
31   libname samples "C:\SAS\CORE\SAMPLE\";
NOTE: Libref SAMPLES was successfully assigned as
follows:
      Engine:        V612
      Physical Name: C:\SAS\CORE\SAMPLE
32
33   title4 "The PROC CONTENTS Provides Information
About The Data Set";
34
35   proc contents data=samples.salary;
36   run;
NOTE: The PROCEDURE CONTENTS used 1.32 seconds.
```

```
37
38   title4 "The PROC MEANS Provides Information
About Variables In Question";
39   title5 "Output data used in PROC FORMAT for
RANGES in SALARY variable";
40
41   proc means data=samples.salary MAXDEC=2;
42    var salary;
43
44    output out = salnums;
45   run;
NOTE: The data set WORK.SALNUMS has 5 observations
and 4 variables.
NOTE: The PROCEDURE MEANS used 0.82 seconds.
46
47   data sumrec (keep= maxsal minsal smean stdev
ucl lcl);
48    set _last_ end=alldone;
49    if _STAT_='MAX' then do;
50           maxsal=salary;
51           retain maxsal;
52    end;
53    if _STAT_='MIN' then do;
54           minsal=salary;
55           retain minsal;
56    end;
57    if _STAT_='MEAN' then do;
58           smean=salary;
59           retain smean;
60    end;
61    if _STAT_='STD' then do;
62           stdev=salary;
63           retain stdev;
64    end;
65    if alldone then do;
66           ucl=stdev+smean;
67           lcl=stdev-smean;
68           output;
69    end;
70    else do;
71           delete;
72           return;
73    end;
74   run;
NOTE: The data set WORK.SUMREC has 1 observations
and 6 variables.
NOTE: The DATA statement used 1.47 seconds.
75
76   proc print;
77   run;
NOTE: The PROCEDURE PRINT used 0.22 seconds.
78
79   title4 "At This Point, Some Reordering is
Needed on Variables to Make Sense";
80
81   data popanly;
82    if not allsum then do;
83        set sumrec end=allsum;
84        retain maxsal minsal smean stdev ucl lcl;
85        delete;
86        return;
87    end;
88    else do;
89       set samples.SALARY;
90        if salary gt ucl then pay="Above UCL ";
91        if salary lt lcl then pay="Below LCL ";
92        if salary gt smean and salary lt ucl then
pay="Above Avg ";
93        if salary lt smean and salary gt lcl then
pay="Below Avg ";
94        if enddate=. then current="Yes";
95           else current="No ";
96        jobtype=substr(jobcode,1,3);
97        if jobcode=" " then put _all_;
98    end;
99   run;
NOTE: The data set WORK.POPANLY has 319
observations and 14 variables.
NOTE: The DATA statement used 0.44 seconds.
100
101  proc sort;
102   by salary;
103  run;
NOTE: The data set WORK.POPANLY has 319
observations and 14 variables.
NOTE: The PROCEDURE SORT used 0.28 seconds.
104
105  proc freq data=_last_;
106   format salary salrange.;
107   tables jobcode;
108   tables jobtype;
109   tables salary;
110   tables current;
111   tables pay;
112   tables jobtype*salary;
113   tables jobtype*pay;
114  run;
NOTE: For table location in print file, see
      page 4 for JOBCODE
      page 9 for JOBTYPE
      page 9 for SALARY
      page 10 for CURRENT
      page 10 for PAY
      page 11 for JOBTYPE*SALARY
      page 16 for JOBTYPE*PAY
NOTE: The PROCEDURE FREQ used 0.7 seconds.
```

CONTENTS PROCEDURE OUTPUT

Data Set Name: SAMPLES.SALARY
Observations: 319
Member Type: DATA
Variables: 5
Engine: V612
Indexes: 0
Created: 17:08 Thursday, April 25, 1996
Observation Length: 40
Last Modified: 17:08 Thursday, April 25, 1996
Deleted Observations: 0
Protection:
Compressed: NO
Data Set Type:
Sorted: NO
Label:

-----Engine/Host Dependent Information-----

Data Set Page Size: 8192
Number of Data Set Pages: 2
File Format: 607
First Data Page: 1
Max Obs per Page: 203
Obs in First Data Page: 180

----Alphabetic List of Variables and Attributes-----

#	Variable	Type	Len	Pos	Format	Informat	Label
3	BEGDATE	Num	8	16	DATE7.	DATE7.	
4	ENDDATE	Num	8	24	DATE7.	DATE7.	
1	IDNUM	Num	8	0	SSN11.	F11.	Identification Num
5	JOBCODE	Char	8	32			
2	SALARY	Num	8	8	DOLLAR12.	DOLLAR12.	Salary

MEANS PROCEDURE OUTPUT

Analysis Variable : SALARY Salary

N	Mean	Std Dev	Minimum	Maximum
319	46936.83	39929.42	12000.00	500000.00

PRINT PROCEDURE OUTPUT

OBS	MAXSAL	MINSAL	SMEAN	STDEV	UCL	LCL
1	500000	12000	46936.83	39929.42	86866.25	-7007.42

FREQUENCY PROCEDURE OUTPUT

CURRENT	Frequency	Percent	Cumulative Frequency	Cumulative Percent
No	10	3.1	10	3.1
Yes	309	96.9	319	100.0

JOBCODE	Frequency	Percent	Cum Frequency	Cum Percent
5IS004	1	0.3	1	0.3
ACT001	1	0.3	2	0.6
APP001	1	0.3	3	0.9
APP002	6	1.9	9	2.8
……*(stuff in the middle deleted)*……				
VID002	2	0.6	318	99.7
VID003	1	0.3	319	100.0

SALARY	Frequency	Percent	Cum Freq	Cum Percent
$25,000 or Less	53	16.6	53	16.6
$25,001 - 50,000	179	56.1	232	72.7
$50,001 - 80,000	64	20.1	296	92.8
$80,001 - 110,000	7	2.2	303	95.0
$110,001 - 200,000	12	3.8	315	98.7
$200,001 - 250,000	3	0.9	318	99.7
$250,001 Up	1	0.3	319	100.0

Web-Application Bar Charts without SAS/GRAPH®

Steve James, Centers for Disease Control and Prevention, Atlanta, GA

ABSTRACT

A picture is worth 1000 words it's said. And while that may be true, producing a picture instead of words can sometimes be 1000 times more difficult. That's particularly true if you're not familiar with SAS/Graph and Annotate. The Internet is a great medium for displaying graphics, but producing them can be troublesome. Anything that would make the task more simple would be welcomed.

With creative uses of HTML tables and images, you can create a bar chart to display information without using SAS/Graph or Annotate. This paper describes how.

Note: This paper assumes the reader has a working knowledge of HTML and SAS/IntrNet™.

INTRODUCTION

In developing a web application displaying Injury Mortality Statistics for the Centers for Disease Control and Prevention, I came across a problem in that I wanted to display the breakdown of certain Injury categories in a meaningful and understandable way. A tabular format (see Figure 1) would provide the necessary format, but lacked the impact of a graphical display.

1997 US Homicide Deaths, Ages 25-34

Cause of Death	Number of Deaths	Percentage
Firearm	3,764	74.8%
Cut/Piercing	575	11.3%
etc.		

Figure 1 - Tabular Format

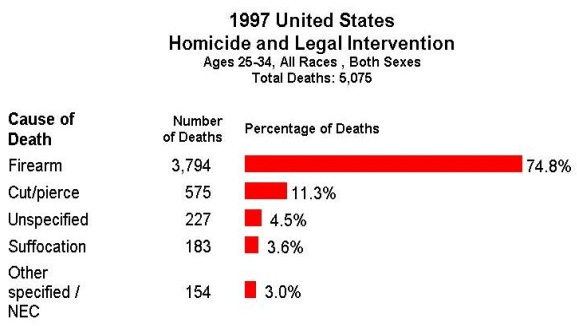


Figure 2 - Graphical Format

USING SAS/GRAPH AND ANNOTATE

A SUGI 24 paper from LeRoy Besseler entitled *Show Them What's Important: Solutions for a Finite Workday in an Era of Information Overload* contained a graph in the format I wanted and contained the code to create such a graph using SAS/Graph and Annotate. A graph for my use would look something like figure 2.

This graph has a much better visual impact than the simple tabular chart. Unfortunately it wasn't easy to produce because some of the labels were too long to display properly without a lot of extra work. Was there another way?

HTML SOLUTION

I knew I could construct a table using standard HTML commands to hold the textual information, but the big questions was how would I get the horizontal bars to generate.

It turned out that using the WIDTH= and HEIGHT= parameters of the HTML image tag would give me what I needed. All I needed was to produce a one-pixel .GIF file for the color I wanted, and then specify the width and height that I wanted it displayed. I could generate the HTML dynamically and thus vary the chart based on the input criteria I needed. In it's simplest form, the HTML code would look like this:

<IMG border=0 height=*h* src="red.gif" width=*w*>

where *h* and *w* were parameters that I could change to vary the size of the bar.

ONE SOLUTION, TWO TECHNIQUES

What's the best way to produce this HTML code? As it turns out, you can do it one of two ways. Using standard SAS® code provides the most flexibility and customization, but requires a fair amount of code. ODS provides a very concise solution, but customization is difficult. Choose the one that most meets your needs

Note: The gif file used in this presentation is available at http://www.cdc.gov/ncipc/images/red.gif

SAS Code Technique

Appendix A contains the SAS code used to produce the graphs above. It uses standard DATA STEP coding. While reading SAS code that produces HTML code may be challenging, the user might benefit from first generating the graph using an HTML editor like FrontPage® and looking at the resulting HTML code.

ODS Technique

Since implementation of the code mentioned above, I was told of another solution that used ODS (see Appendix B). While I've not actually put the code in production it appears to be a viable alternative to the DATA STEP programming. It is included for your benefit, though there are likely things that you would change

before you would actually use this in production.

BENEFITS

There were several advantages to this approach:

The graph could be put in a multi-column table which would let the bar be in one column and the labels be in another. This would allow the HTML table to control how the text of the label wrapped and ensured that it stayed with the horizontal bar. The application didn't have to control that, the browser did.

It didn't require any knowledge of SAS/Graph or Annotate, and used standard SAS/IntrNet techniques.

The resulting HTML file was small which would make downloads quicker. Instead of downloading a big graphic to your browser you end up downloading only a one-pixel .GIF file which speeds response time.

CONCLUSION

Adding graphs to your web pages can make them have a much bigger impact on the user. Using the techniques discussed in this paper can make the job easier.

REFERENCES

ACKNOWLEDGMENTS

I want to acknowledge that Steve Bloom told me about both of these techniques and has made my life much better by easing the process of adding graphics to my web application. I present them here only because he wasn't going to and I thought them too good not to share.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Steve James
Centers for Disease Control and Prevention
4770 Buford Highway, MS-K59
Atlanta, GA 30341
Work Phone: (770) 488-4269
Fax: (770) 488-1665
Email: sjames@cdc.gov

APPENDIX A

```
*-------------------------------------------------------------;
* Prepare data for creating graphics.  The output of this step ;
* is a SAS data set with three variables: cause, deaths, and   ;
* percent.                                                     ;
*-------------------------------------------------------------;
proc freq data=temp ;
  tables cause / noprint out=totals(rename=count=deaths) ;
  weight deaths ;
  label cause='Cause of Injury' deaths='Number of Deaths' ;
  run ;

*-----------------------------------------------------------------------;
* Now produce the chart as a table with three columns: one with the cause;
* of death, one with the number of deaths, and one with the percentage of;
* deaths including the bar chart.                                       ;
* The bar chart is drawn using a .GIF that is one pixel in size, and uses;
* the WIDTH= and HEIGHT= options on the IMG tag to make it the right     ;
* size.  Note: &DEATHS is a macro variable passed in that contains the   ;
* total number of deaths for all categories and is used in the title.    ;
*-----------------------------------------------------------------------;

data _null_ ;
  file _webout  ;
  set work.totals end=lastob;
  if _n_ = 1
  then do ;
       put '<table border="0" cellpadding="2" width="771">' //
            '<tr>' /
           '<td width="132" height="27"><font face="Arial">'
           '<b>Cause of Death</b></font></td>'
           '<td nowrap><p align="right"><font size="2" face="Arial">'
           '<b>Number<br> of Deaths</b></font></td>'
           '<td width="18" height="27"></td>'

           '<td width="536" height="27">'
           '<font face="Arial" size="2">'
           '<b>Percentage of Deaths</b></font></td>'
           '</tr>'
            ;
        end ;
   * create quoted strings for height and width.  Want them;
   * to look like height="20" and width="44.5%" ;
   length height width $8 total 5 ;
   if percent < 1.0 then width=quote("1.0%") ; * set minimum width of bars at 1% ;
   else width = compress(quote(put(percent,5.2) !! '%')) ;
   height = quote("20") ;
```

```
  total = &deaths ;
  put /* print out the cause of death */
      '<tr>'
      '<td width="132" height="30"><font face="Arial">'
      cause
      '</font></td>'

      /* print out the number of deaths */
      '<td width="61" height="30" align="right">'
      '<font face="Arial">'
      deaths comma10.
      '</font></td>'

       /* put spaces in column to hold it open */
      '<td width="10" align="right"><font color="#FFFFFF" size="2" '
      'face="Arial">'
      '<img src="http://www.cdc.gov/navimages/spacer.gif" width="5">'
      '</font></td>'

      /* print out the percentage of deaths and the bar */
      '<td width="538" height="30" align="left">'
      '<font face="Arial">'
      '<img border="0" '
      'src="http://www.cdc.gov/ncipc/images/red.gif"'
      'width=' width 'height=' height '>'
      '<img border="0" '
      'src="http://www.cdc.gov/navimages/spacer.gif" '
      'width="5">'
      percent 4.1 '%</font></td>'
      '</tr>'
      ;
    if lastob
    then do ;

        put '<tr>'
            '<td width="132" height="30"><font face="Arial" size="2">'
            '<b>Total Deaths </b></font></td>'
            '<td width="85" height="30" colspan="2"><font size="2">'
            '<p align="right"></font><font size="2" face="Arial"><b>'
            total comma10.
            '</b></font><font face="Arial"> </font></td>'
            '<td width="536" height="30"></td>'
            '</tr>'
            ;

         put '</table>' ;
         end ;
    run ;
```

APPENDIX B

```
proc template;
       edit Base.Freq.OneWayFreqs;
         COLUMN Line Variable ListVariable Frequency
         TestFrequency bar Percent  /*
        TestPercent CumFrequency CumPercent */;
         define bar;
             HEADER=" ";
               format=120.;
```

```
                just=l;
         compute as Percent;
        TRANSLATE
                _val_>0 into
                          "<a><img
 SRC='http://www.cdc.gov/ncipc/images/red.gif' width="
                                      || put(_val_ * 10 ,4.)
 || " height=10 ALT='' BORDER='0'>" || put(_val_/100,percent8.1) !! "</a>";
                               ;
           end;
         end;
      run;

ods listing close;
ods HTML file = "c:\temp\freq.html"; * set up file to hold HTML code ;


proc freq data=temp order=freq;
        weight deaths;
        table cause/ nocum;
run;

*-------------------------------------------------------------;
* Remove what you have done so it will not affect later freqs.;
*-------------------------------------------------------------;
Proc template;
Delete Base.Freq.OneWayFreqs;
Run;

ods HTML close;
```

Paper P407

Bootstrapping a Multidimensional Preference Analysis

E. Barry Moser and Xiaoming Liang, Department of Experimental Statistics,
Louisiana State University Agricultural Center, Baton Rouge, LA 70803-5606

ABSTRACT

Preference data are frequently collected to help decision makers prioritize items, or are measured on subjects, such as animals, and used to help infer preference for or avoidance of resources. Multidimensional preference analysis may be used to construct a biplot of the items and subjects in the space of a few components as an aid to interpretation and presentation of results. The variability associated with the objects in the plot, however, is frequently not shown. The SAS/STAT® survey sampling procedure SURVEYSELECT is used to select bootstrap samples from a preference data set, then multidimensional preference analysis is performed with the SAS/STAT® PRINQUAL procedure to produce estimates of the variabilities of objects. Next the SAS/GRAPH® GPLOT procedure is used to produce a variety of graphics using the bootstrap information that can then show the precision with which objects are estimated in the components space.

INTRODUCTION

Preference data arise in a variety of situations where items are rank-ordered according to some preference (or avoidance) of the items. In food science, a set of food preparations may be rank-ordered by judges based upon flavor characteristics. In marketing, product presentations may be ranked by a sample of potential consumers, while in ecology, habitat types may be “rank-ordered” by animals based upon their frequency of usage of habitats as determined from radio telemetry data. In an agricultural economics project of Professor Kenneth Paxton, Louisiana State University Agricultural Center, cotton farmers were asked to rank-order a dozen research thrusts that the Agricultural Center might pursue with respect to cotton production. Upon consultation with Dr. Paxton, one of the analyses that we suggested was a multidimensional preference analysis.

Multidimensional preference analysis (MDPREF) is one of a number of perceptual mapping techniques for graphically displaying product preferences in a low dimensional space of principal components (Carroll, 1972; see Kuhfeld, 1992). In the data matrix, the columns or variables correspond to the judges, while the rows or observations correspond with the items or products. A “biplot” can be produced where the items are given as points in the display, while the judges can each be plotted as vectors emanating from the origin. The longer the vector, the more of that judge's information that is contained in the display. The items are projected onto the vectors to interpret which items are most and least preferred by a judge. In a trained panel, such as in food science, it is often hoped that the judges are homogeneous and so their vectors would be coincident in a perfect world. In product presentation studies, variations among consumers, as indicated by different directions of their vectors, may be related to demographic or other characteristics of the consumers. In some cases, these characteristics can also be superimposed on the biplot.

Unlike regression analysis, these vector and item points do not come with a standard error or confidence interval formula. Thus although the data in the study may be a sample from a much larger population of interest, the MDPREF results are presented as if the sample is the population of interest. Since we are dealing with rank-ordered responses that have special mathematical properties, we decided to use a Monte Carlo method to derive measures of variability and confidence regions for the MDPREF analysis. In particular, bootstrap samples (see Manly, 1997) were constructed and the replications used to build confidence regions. Monteleone et al. (1998) used a similar approach to form preference maps for starchy food consumption.

JOURNAL PREFERENCES EXAMPLE

E. Roskam (as taken from Gifi, 1990) reported on journal preferences in a psychological research area as determined by 39 scientists. Each scientist rank-ordered the list of 10 journals as to their importance to the field. Note that for this type of data, not only do we have positive integer values, but also the sum of the ranks for each scientist is a constant. Thus multivariate normality is not likely plausible, and the dimensionality of the problem is at least 1 less than the number of items being ranked. A SAS® program was used to construct the MDPREF analysis of these data, then to bootstrap the data set, and to construct indicators as to the variability of the original display.

First, the original data set is constructed. In most settings, the observations will consist of the responses from each judge, while the columns will correspond with the items being ranked. Notice that this data organization is the transpose of the way in which we will need to have the data for analysis.

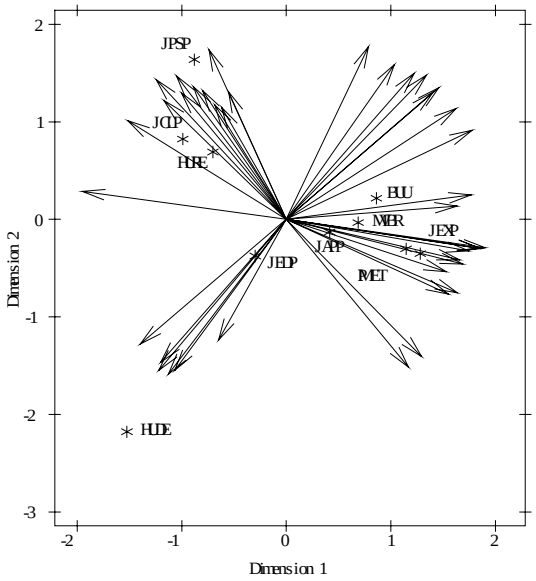
```
Data Roskam;
 Input Judge JEXP JAPP JPSP MVBR JCLP
       JEDP PMET HURE BUU HUDE;
Datalines;
1. 7 4 1 8 10 9 5 2 3 6
2. 7 6 2 9 3 8 10 1 4 5
3. 10 5 1 7 4 6 8 2 3 9
---- Data Omitted -----
38. 2 3 6 5 7 8 4 9 1 10
39. 2 6 7 3 10 8 4 9 1 5
;
```

Next the data matrix is transposed to get it into the proper form (PROC TRANSPOSE), the items are reverse-ranked so that the most important items have the highest rank (PROC RANK), and then the MDPREF analysis is performed using the PRINQUAL procedure. The %PLOTIT macro of SAS/STAT® is used to display the results from the PRINQUAL procedure. The basic code used is:

```
Proc Transpose Data=Roskam Out=Roskam2
               Prefix=Judge Name=Journal;
 Var JEXP--HUDE;
 Id Judge;
Run;

Proc Rank Data=Roskam2 Out=Roskam3
          Descending;
 Var Judge1-Judge39;
Run;

Proc Prinqual Data=Roskam3 Out=PQResults
          N=2 Replace MDPREF MaxIter=150;
 Id Journal;
 Transform Monotone(Judge1-Judge39);
Run;

Proc Print Data=PQResults;
Run;

%Plotit(Data=PQResults,
        Datatype=MDPREF 2);
```

The results from this analysis are shown in the following figure. There, it is observed that there appear to be different

groups of scientists in terms of what journals they prefer. For example, the journal HUDE (*Human Development*) is highly preferred by about 6 judges, 5 of which happen to be from Developmental and Educational Psychology departments.



BOOTSTRAP ANALYSIS

Let n be the size of the original data set. The basic bootstrap process takes a simple random sample with replacement of size n from the original data set. The SURVEYSELECT procedure using the METHOD=URS with n=39 provides such a sample. In fact, by setting REP=M, where M is some positive integer, we can generate M such bootstrap samples in a single call to SURVEYSELECT. Note that the original data set (though of reverse-ordered scores) is bootstrapped, as we believe that the scientists (judges) are the random sampling units, not the journals.

```
/*
 * Rearrange like original data but with
 * reverse-ranked values
 */
Proc Transpose Data=Roskam3
          Out=Roskam4(Drop=_LABEL_) Name=Judge;
 Var Judge1-Judge39;
 Id Journal;
Run;

Proc SurveySelect Data=Roskam4 Method=URS
                  n=39 Rep=1000 Out=Boots;
Run;
```

Once the bootstrap samples are selected, the data are again transposed, but BY REPLICATE, and the PRINQUAL procedure is again called to perform the analysis BY REPLICATE.

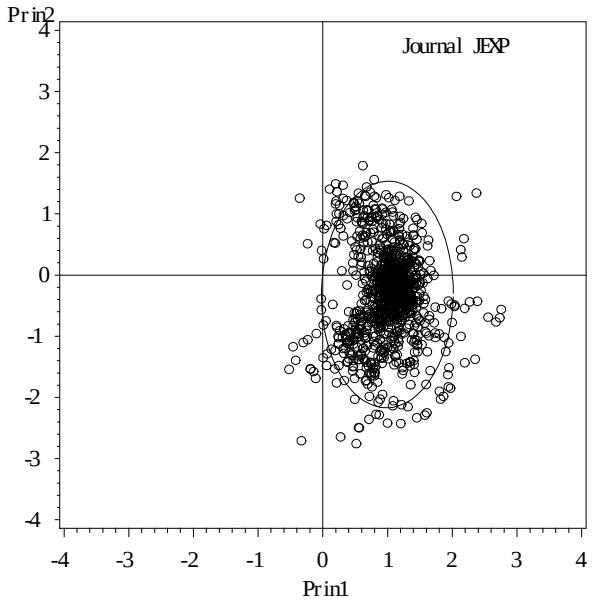
```
Proc Transpose Data=Boots Out=Trans
               Prefix=Judge Name=Journal;
 Var JEXP--HUDE;
 By Replicate;
Run;

/*
 * Perform separate MDPREF analyses on each rep
 */
Proc Prinqual Data=Trans Out=PQResults2 N=2
           Replace MDPREF NoPrint MaxIter=150;
  Id Journal;
  Transform Monotone(Judge1-Judge39);
  BY Replicate;
RUN;
```

It is important to note that the direction of a principal component is unimportant. Thus, it is possible that one result is a reflection of one or more axes of the original solution, and so each solution from each replicate must be registered relative to the original solution. We did this by measuring the sum of squared errors from each item point in the bootstrap solution to the item points in the original solution and selecting appropriate reflections to minimize this error. Thus we assumed that the appropriate registration for a solution is the one that "looks" most like the original solution. The code for this involved a couple of data steps to first compute the errors and find the best solution, then a second step to make the necessary conversions to the components in the solutions.

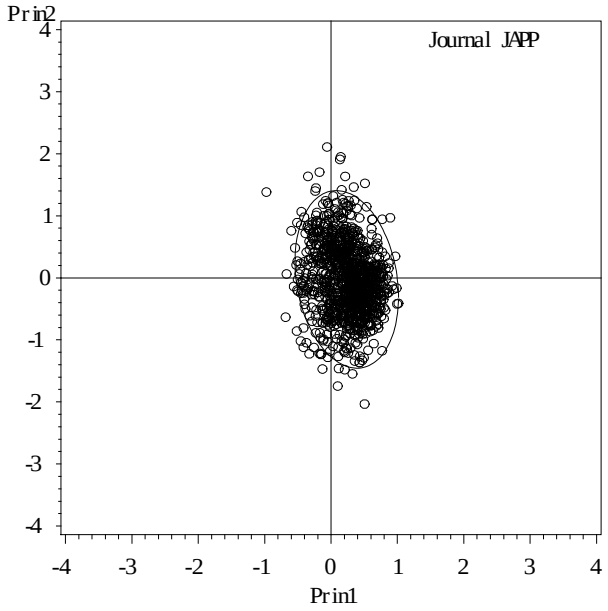
Next the %CONELIP macro obtained from the SAS Institute support web site was used to construct 95% confidence ellipses for each journal using the bootstrap replicates. A simple SAS macro was used to process the results for each journal separately, then to build a composite data set containing all of the ellipses.

```
%Macro Ellipses;
GOptions Reset=Symbol Reset=Axis;
Proc Datasets Library=Work NoList NoWarn;
 Delete All;
Run;
Quit;
%Let Journals=JEXP JAPP JPSP MVBR JCLP
             JEDP PMET HURE BUU HUDE;
%Do J=1 %To 10;
 %Let Journal=%Scan(&Journals,&J);
 %Conelip(Data=Scores(Where=(_NAME_=
        "&Journal")), Out=&Journal,
   x1=Prin1, x2=Prin2, mean=no, conf=.95);
 Data &Journal;
  Length Journal $4.;
  Retain Journal "&Journal";
  Set &Journal;
  Prin1=X1; Prin2=X2;
 Run;
 Proc Append Data=&Journal Base=All;
 Run;
 Title1 "Journal &Journal";
 Proc GPlot Data=&Journal;
  Plot Prin2*Prin1=Id / NoLegend
   VAxis=Axis1 HAxis=Axis1 VRef=0 HRef=0;
  Axis1 Length=4in Order=(-4 To 4 By 1);
  Symbol1 C=Black V=CIRCLE H=1 I=None;
  Symbol2 C=Black V=None I=Join L=1;
 Run;
 Quit;
%End;
%Mend Ellipses;
%Ellipses;
```

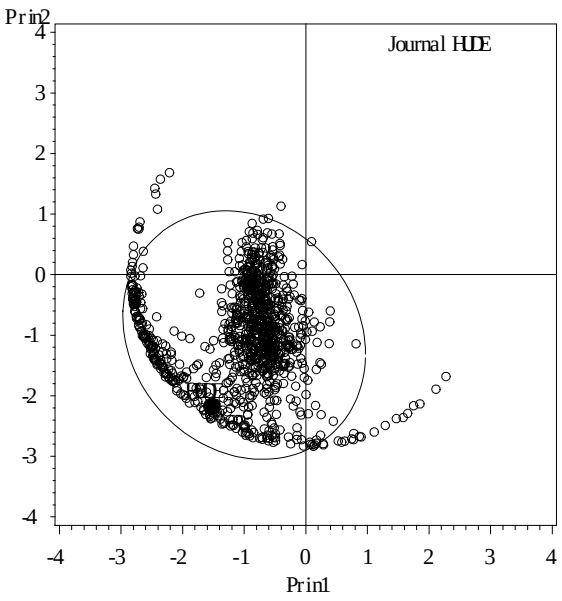
The replicates and superimposed ellipse for the journal JEXP are given below.



Notice that there is quite a bit of variability in the points and that the ellipse doesn't capture the data very well. On the other hand, the points tend to be in the southeast to east direction from the origin.

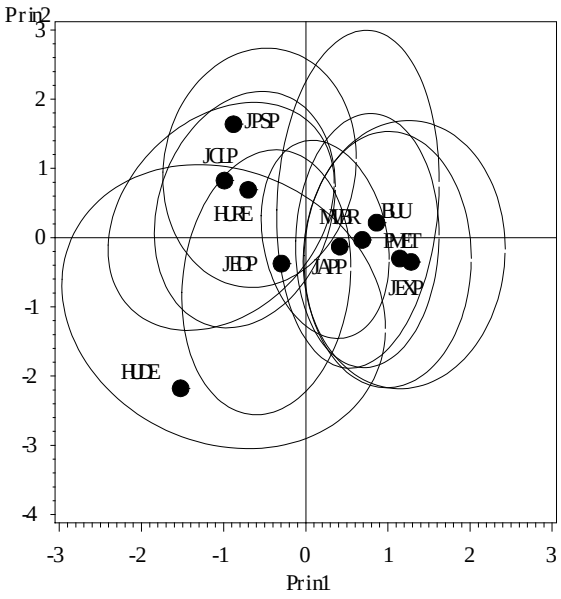


For the JAPP journal shown above, the points congregate a lot near the origin suggesting that this journal is not well represented in this presentation, or is generally preferred in the middle ranks by most everyone. A third journal, HUDE, demonstrates the wide variability that an item might exhibit. For this journal, a number of scientists ranked it as their number 1 journal, while several others ranked it at the very bottom of their lists.



We constructed these plots for each journal, and then the set of ellipses was plotted in a single figure using the SAS/GRAPH code:

```
GOptions Reset=Symbol Reset=Axis;
Proc GPlot Data=ALL(Where=(Id=2))
           Annotate=Centers;
 Plot Prin2*Prin1=Journal / VAxis=Axis1
      HAxis=Axis1 NoLegend VRef=0 HRef=0;
 Axis1 Length=4in;
 Symbol1 C=Black V=None I=Join L=1 R=10;
Run;
Quit;
```



The magnitudes of the ellipses are related to the variability that they exhibit through the selection of sampling units from the population.

CONCLUSION

The bootstrapping technique provides considerable insight into the variability in results that could occur simply as a consequence of the individuals selected into the sample assuming that the original sample is a simple random sample. For example, the journal Human Development (HUDE), shows a considerable range over which its placement could occur and is likely a result of the groups of individuals that ranked the journal either as 1 or 10, and so as these group sizes change as a consequence of sampling, so does the placement of the point. Note that here the sample size of 39 is small and so a few observations can have a profound impact on the results. For larger data sets, this technique should produce much more stable results.

Though confidence ellipses based upon the bivariate normal distribution were used to approximate the region occupied by the journal vectors, the bivariate normal model does not appear to be a good model for several of the journal points. Some nonparametric method such as hull peels (Green, 1981) may be more useful in these situations.

Finally there are a few points corresponding with a few bootstrap replications that appear to be miss-registered as viewed from the plots. Thus a more elegant algorithm for insuring correct registration may be required.

REFERENCES

Carroll, J.D. 1972. Individual differences and multidimensional scaling. Pages 105-155 in R.N. Shephard, A.K. Romney, and S.B. Nerlove, eds. *Multidimensional Scaling: Theory and Applications in the Behavioral Sciences*, Vol. 1., Seminar Press, NY.

Gifi, A. 1990. *Nonlinear Multivariate Analysis*. John Wiley & Sons, New York, p. 183.

Green, P.J. 1981. Peeling bivariate data. Pages 3-19 in Barnett, V., ed. *Interpreting Multivariate Data*. John Wiley & Sons, Inc., NY.

Kuhfeld, W.F. 1992. Marketing research: uncovering competitive advantages. *Proceedings of the SAS Users Group International Conference* 17, 1304-1312.

Manly, B.F.J. 1997. *Randomization, Bootstrap, and Monte Carlo Methods in Biology*. Chapman and Hall, Inc., London, 399pp.

Monteleone, E., L. Frewer, I. Wakeling, and D.J. Mela. 1998. Individual differences in starchy food consumption: the application of preference mapping. *Food Quality and Preference* 9 (4), 211-219.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors at:

E. Barry Moser
Louisiana State University Agricultural Center
Department of Experimental Statistics
Baton Rouge, LA 70803-5606
Work Phone: 225-578-8303
Fax: 225-578-8344
Email: bmoser@lsu.edu
http://www.stat.lsu.edu/faculty/moser

Detecting Anomalies in Your Data Using Benford's Law

Curtis A. Smith, Defense Contract Audit Agency, La Mirada, CA

ABSTRACT

Analyzing large amounts of data looking for anomalies can be a disheartening task. You need techniques that will allow you to quickly assess the data in ways that will highlight potential anomalies while keeping you from chasing the wind. Benford's Law is one such technique. Using Benford's Law and SAS© Software you can quickly identify one or more first digit patterns that defy statistical averages. Within this paper, the author will present SAS code that will enable you to quickly and easily find anomalies in the data you analyze. The SAS code will include the Data Step, the Merge statement, and the FREQ, REPORT, and GPLOT procedures. The author will also present some findings from the data he analyzes. The technique presented is powerful, yet easy to understand and use.

INTRODUCTION

Benford's law is so named after Dr. Frank Benford. Dr. Benford was a physicist working for General Electric in the 1930's. He noticed that certain pages of his logarithm book were more worn than others. After some study he realized that within a large enough universe of numbers that were naturally compiled that the first digits of the numbers would occur in a logarithmic pattern. The first digits of numbers are the non-zero, absolute value integers. For example, the first digit of 1 is 1; the first digit of 10 is 1; the first digit of -100 is 1; and the first-two digits of 1200 are 12. Dr. Benford tested 20,229 sets of numbers from many unrelated types of data. He found the same pattern to always exist. This statistical oddity provides an opportunity to those who need to analyze vast amounts of data for anomalies. If the first digits within the numbers of the data you are analyzing do not follow the Benford pattern, then something unnatural has happened with your data.

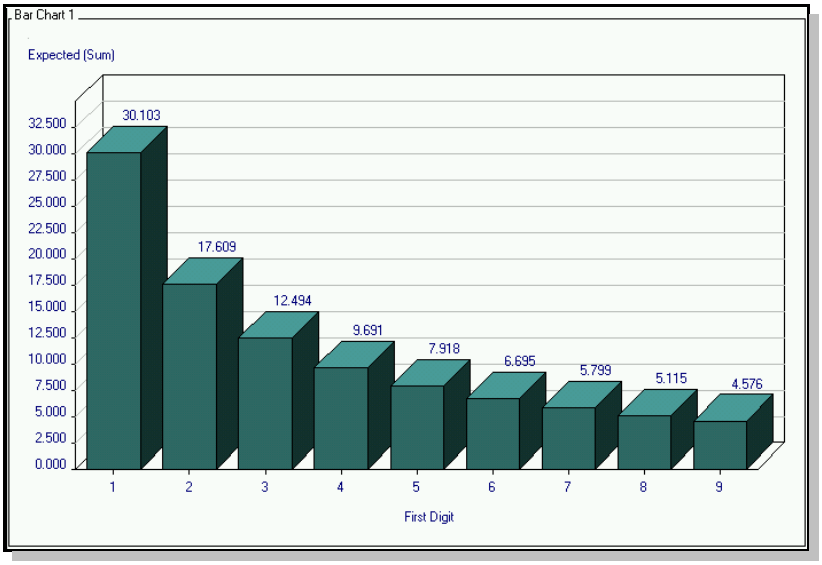
SO, WHAT'S THIS BENFORD'S LAW LOOK LIKE, ANYWAY?

Simply stated, any digit or combination of digits will follow the following logarithmic pattern:

$$x = \log_{10}(1+(1/n))$$

Where n is the digit or combination of digits being tested and x is the percentage of their occurrence.

Considering the first digits 1 to 9, the expected distribution is:



(By the way, I created this graph with SAS/Graph's Graph-N-Go.) So, why is the first digit of 1 so much more prevalent than other digits? And why the declining distribution from 1 to 9? Perhaps Benford's Law can be explained this way. In a series of numbers, to go from 10 to 20 requires a 100% increase, but to go from 20 to 30 requires a 50% increase, and so forth. So, if numbers are being incremented, it takes less incrementation to go from 8 to 9 than it does to go from 1 to 2. Once the number increments from 8 to 9, then with only a little incrementation, the 9 will increment to 10 (a first digit of 1). So, numbers tend to fall within a first digit of 1 more than any other digit.

However, this distribution will not exist within every set of numbers. First, the universe of numbers must be large enough for the distribution to take shape. Some have found that a universe smaller than 100 items will not exhibit the pattern. Second, the numbers must be free of artificial limits or origins. For example, when evaluating a data file of travel claims you might find that the first-two digit combination of 24 exists greater than expected with Benford's Law. This might happen if the company has a policy that reimbursement claims for $25 and above must be supported with receipts - so travelers claim a lesser amount, such as $24.95. If you analyze the transactions in my checkbook you will find the first-four digit combination of 1995 to occur at a high rate. This is because my ISP always charges me a monthly rate of $19.95.

HOW CAN BENFORD'S LAW HELP YOU?

If you are in the business of analyzing data, such as the noble profession of auditing, you might need to look for areas of fraud or areas of oddities. Dr. Mark J. Nigrini and others have successfully used Benford's Law to detect potential fraud. Dr. Nigrini termed the use of analyzing digits within numbers as "digital analysis." It is difficult for the fraudster to avoid detection from digital analyses because the fraudster typically cannot influence an entire data file. Thus, the fraudster will invariably alter numbers in such a way that destroys the Benford distribution. But, you don't have to be looking for fraud to benefit from Benford's Law. There are many non-fraudulent reasons why a universe of numbers can violate Benford's Law, yet still warrant your investigation.

> Benford's Law gives auditors the expected frequencies of the digits in tabulated data. The premise is that we would expect authentic and unmanipulated data to exhibit these patterns. If a data set does not follow these patterns, however, a few possible reasons exist to explain this phenomenon:
>
> 1. The data set did not meet the three tests, and/or,
> 2. The data set includes invented numbers, biased numbers, or errors.
>
> - Dr. Mark Nigrini

Start Digging

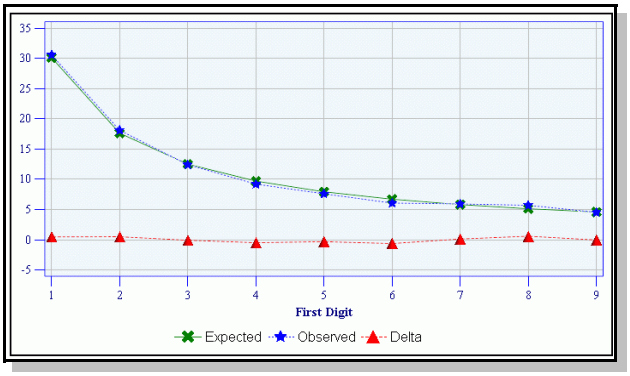
Here are some samples from labor transaction data I have analyzed. First, look at a first digit analysis. Notice the table showing the observed versus expected distribution and delta, then notice the plot showing the same.

First Digit(s)	Observed Frequency Count	Observed Frequency Percent	Expected Frequency Percent	Observed - Expected Frequency Percent
1	419	30.562	30.103	0.459
2	248	18.089	17.609	0.480
3	170	12.400	12.494	-0.094
4	126	9.190	9.691	-0.501
5	104	7.586	7.918	-0.332
6	83	6.054	6.695	-0.641
7	81	5.908	5.799	0.109
8	78	5.689	5.115	0.574
9	62	4.522	4.576	-0.054
	1,371	*100.000*	*100.000*	*-0.000*

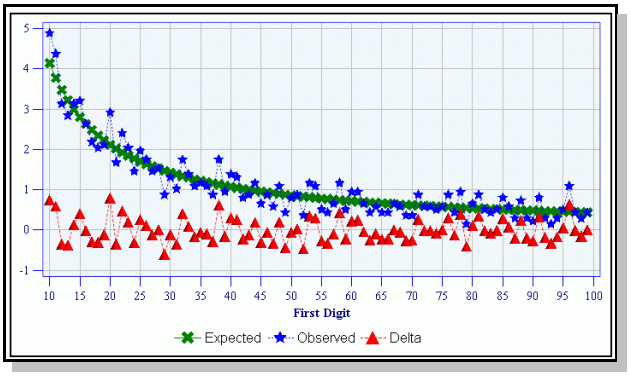
A bank auditor found that credit card balances written off as uncollectible had an excessive level of numbers with first-two digits 49. The investigation found that $5,000 was an internal write-off limit for internal collections employees. One employee was responsible for most of the 49s by working with friends and having them apply for a card and then running up a balance to just below $5,000. The employee would then write the debt off. The systematic nature of the fraud was evident from the first-two digits graph.

- Dr. Mark Nigrini

Then, look at the plot of the same information.



Nice curves. In this case, everything looks as Benford predicted. So, a dead end, right? No, just an opportunity to dig further.



Notice here the anomalies beginning to show themselves. The X (green) line represents the Benford expected distribution - a very nice curve, indeed. The star (blue) line is the observed, and the triangle (red) the delta. You can quickly see the anomalies. Yet, in this example, the anomalies are not that great.

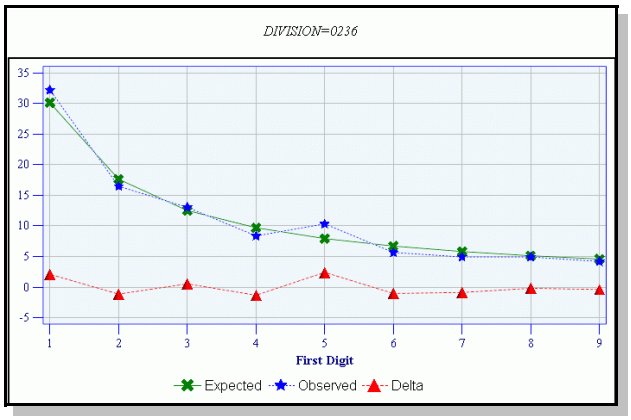
Dig Deeper Using Multiple Digits

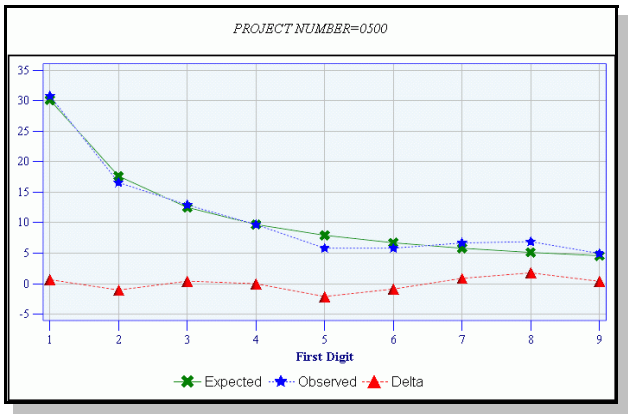
You can analyze the data using a combination of digits, such as the first-two digits. This can help find anomalies not apparent in a first digit analysis and can isolate further the anomalies found in a first digit analysis. Let's look at an example. First, take a look at a portion of the table report.

First Digit(s)	Observed Frequency Count	Observed Frequency Percent	Expected Frequency Percent	Observed - Expected Frequency Percent
10	67	4.887	4.139	0.748
11	60	4.376	3.779	0.598
12	43	3.136	3.476	-0.340
13	39	2.845	3.218	-0.374
14	43	3.136	2.996	0.140
15	44	3.209	2.803	0.406
16	36	2.626	2.633	-0.007
17	30	2.188	2.482	-0.294
18	28	2.042	2.348	-0.306
19	29	2.115	2.228	-0.112
20	40	2.918	2.119	0.799

Dig Deeper By Subsetting

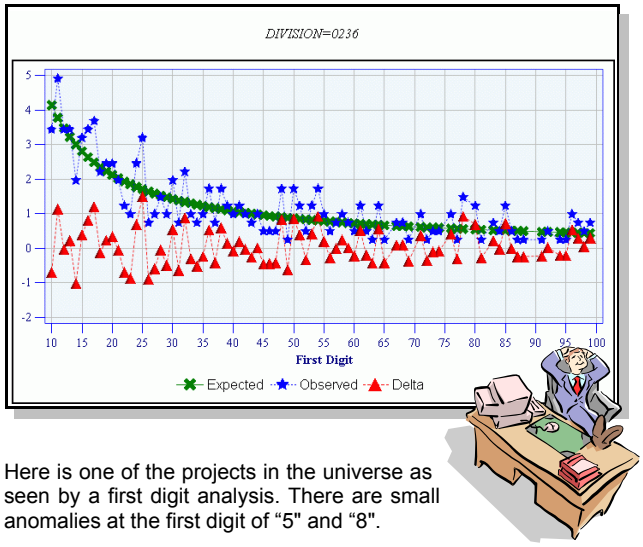
Another technique you can use is to subset your data. For example, if you analyze all of the labor transactions together by first digit, first-two digits, and so forth and find nothing, you might then look at subsets of the data. Rather than looking at all the labor transactions together, look at subsets by department, or by week, or by employee. You might then see anomalies showing up. Why? Because if one department or employee is doing something funny, or if something funny was being done during one week, looking at the entire universe can obscure the funny business. But isolating subsets can be revealing. Here are some examples of first digit and first-two digits analyses of labor data subset by division and then by project.





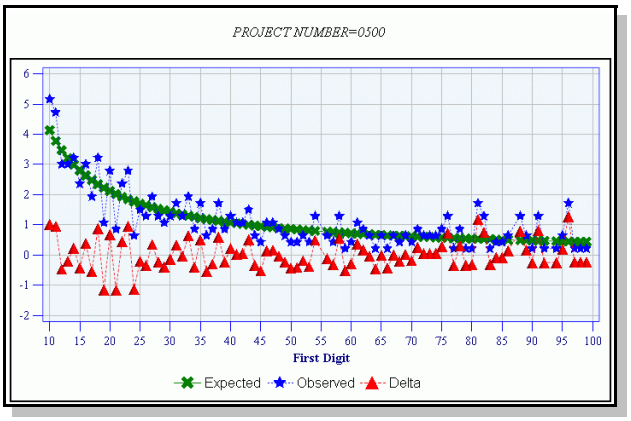
Notice in the first digit analysis for this one division there is an anomaly in the first digit of “5". While not a huge variance from the expected, a variance nonetheless. Then, looking at the same division at the first-two digits you can see anomalies all over the place. There are significant variances at values “50" and “54", both of which have a first digit of “5". But notice the even more significant variance at the value “25". This variance didn’t even register in our first digit analysis. This is evidence of why going deeper then just the first digit analysis can be useful.

Let’s look even further, by subsetting our universe by project.



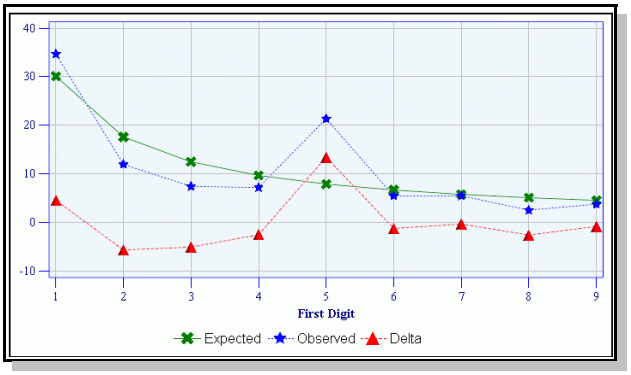
Here is one of the projects in the universe as seen by a first digit analysis. There are small anomalies at the first digit of “5" and “8".

Looking at the same project using a first-two digits analysis you can see anomalies all over the place.

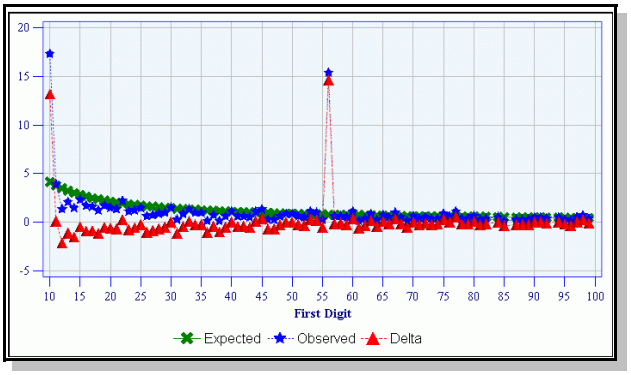


Big Findings

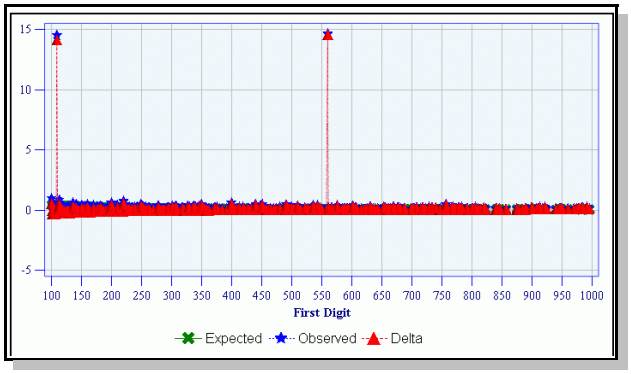
Well, enough of this fooling around. Let’s take a look at some striking examples. Subsetting the data by location turned out to be much more revealing. First, check out the first digit analysis.



Here you can see anomalies all over the place - but the anomaly at first digit “5" really gets our attention. So, let’s dig a little deeper. Feast your eyes on the first-two digit analysis.



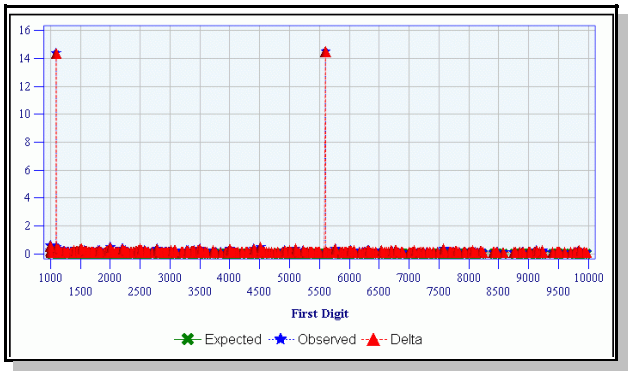
Wow! That first digit “5" anomaly turns out to be a really big first-two digit “56" anomaly. And, look at that first-two digit “10"anomaly that was hardly noticeable in the first digit analysis. This is going so well, let’s dig even deeper. Take a gander at a first-three digit analysis.



Now we can pinpoint the two great anomalies at “109" and “560". We can easily determine these values and their frequency by referring back to the table report, a portion of which is shown below.

LOCATION	First Digit(s)	Observed Frequency Count	Observed Frequency Percent	Expected Frequency Percent	Observed - Expected Frequency Percent
11	560	120	14.652	0.077	14.575
	109	119	14.530	0.397	14.133
	100	8	0.977	0.432	0.545
	113	7	0.855	0.383	0.472
	220	6	0.733	0.197	0.536

You might be wondering if you could go any deeper. Certainly. Let’s take a look at a first-four digit analysis.



In this case, the first-four digit analysis did not add any value because the two anomalies are at “1090" and “5600". Because the numeric variable is dollars, the “1090" value is probably $10.90 or $109.00 and the “5600" value is probably $56.00, $560.00, or $5,600.00. So, in both cases, digging to the next first digit will just add another zero to the end of our number, which won’t help pinpoint the anomalies any better.

What to Do

So, what can you do with these anomalies? You can start to cross check them, looking for something in common. Considering our example, look to see if the anomalies within divisions also occur within projects. Then, you can begin extracting the actual data records that contain the anomalies and use the information on those data records to get to the root of the anomalies.

SO, HOW WAS SAS USED?

To produce these results, I used SAS to do five main tasks, which are as follows:

- Determine the observed distributions of the first digits
- Determine the expected distributions of the first digits
- Merge the observed and expected distributions and compute the deltas
- Create a report of the observed versus expected distribution
- Create a plot of the observed versus expected distribution

That doesn’t seem too difficult. Let’s look at the code at a high level. The code shown is for a first digit analysis for an entire universe. The modifications to do a first-two digits or more digit combination and to do a BY group analysis are not too different and will not be presented.

Determine the Observed Distributions of the First Digits

Here’s the fundamental code I use to determine the distributions of the first digits.

```
DATA WORK.OBSERVED
  (KEEP=FIRSTDGT COUNT VAR INDEX=(FIRSTDGT));
  SET IN.INFILE.;
  FIRSTDGT=
    INPUT(SUBSTR(SCAN(PUT(VAR,BEST8.),1),1,1),
    BEST8.);
  COUNT=1;
RUN;
PROC FREQ DATA=WORK.OBSERVED;
  TABLES FIRSTDGT/OUT=WORK.BENFORD
    (RENAME=(PERCENT=OBSERVED));
```

In the code above, the user-specified file to analyze (IN.INFILE) is read and a new variable, FIRSTDGT, is created. This new variable is created by using the PUT function to convert the user specified numeric variable (VAR) to a character string. Then the SCAN function gets the first digit from the converted value. Then the INPUT function stores that first digit to the new numeric variable. Another new variable, COUNT, is created and set to 1. This will be used to summarize the frequency of the first digit.

Next, the FREQ procedure is used to create the frequency distribution of the first digit.

> Benford’s Law provides auditors with the expected digit frequencies in tabulated data. By examining the digit and the number frequencies, auditors can gain data insights that might be missed using traditional analytical procedures and sampling methods. The digit and number patterns could point to number invention, systematic frauds, data errors, or biases in the data.
>
> - Dr. Mark Nigrini

Determine the Expected Distributions of the First Digits
Here's the fundamental code to create the expected distributions.

```
DATA WORK.EXPECTED(INDEX=(FIRSTDGT) DROP=I);
  FORMAT EXPECTED 8.3;
  DO I = 1 TO 9;
  FIRSTDGT=I;
  EXPECTED=(LOG10(1+(1/I))*100);
  OUTPUT;
  END;
```

This data step simply creates a new data set that contains the first digit and the result of the log10 formula demonstrated by Dr. Benford.

Merge the Observed and Expected Distributions and Compute the Deltas
Here's the basic code for creating a data set with the observed and expected first digit distributions and the deltas for each digit.

```
DATA OUT.BENFORD;
  MERGE WORK.EXPECTED(IN=A)
    WORK.BENFORD(IN=B);
  BY FIRSTDGT;
  IF B;
  DELTA=SUM(OBSERVED,-EXPECTED);
```

This data step simply creates a new data set by merging the observed and expected data sets by the first digit and creates a new variable, DELTA, containing the difference between the observed and expected distributions.

Create a Report of the Observed Versus Expected Distribution
Here's the fundamental code to create the tabular report.

```
PROC REPORT DATA=OUT.BENFORD NOWINDOWS
  HEADSKIP MISSING;
  COL FIRSTDGT COUNT OBSERVED EXPECTED DELTA;
  DEFINE FIRSTDGT /ORDER 'FIRST DIGIT(S)'
    WIDTH=5;
  DEFINE COUNT /'OBSERVED FREQUENCY COUNT'
    WIDTH=12 FORMAT=COMMA12.;
  DEFINE EXPECTED /'EXPECTED FREQUENCY
  PERCENT' WIDTH=12 FORMAT=8.3;
  DEFINE OBSERVED /'OBSERVED FREQUENCY
    PERCENT' WIDTH=12 FORMAT=8.3;
  DEFINE DELTA /'OBSERVED - EXPECTED FREQUENCY
  PERCENT' WIDTH=12 FORMAT=8.3;
  RBREAK AFTER /SUMMARIZE OL UL;
```

This code is simply a REPORT procedure with the key variables defined.

Create a Plot of the Observed Versus Expected Distribution
Here's the fundamental code to create the overlay plot.

```
PROC GPLOT DATA=OUT.BENFORD;
  PLOT EXPECTED*FIRSTDGT OBSERVED*FIRSTDGT
    DELTA*FIRSTDGT / OVERLAY;
```

This code simply uses the GPLOT procedure to create an overlay plot of the OBSERVED, EXPECTED, and DELTA variables over the first digit variable.

My actual code is a bit more complicated, as I use macro variables to allow the user to make specifications before running the application. I also use ODS statements to make HTML files of the REPORT and GPLOT procedure output. And I use ActiveX controls in my GPLOT procedure output.

CONCLUSION

Analyzing large amounts of data for anomalies or potential fraud does not have to be a disheartening task. Using digital analyses, such as Benford's Law, you can easily find anomalies in your data. It was not my intention within this paper to provide all of the SAS code I used to create the output shown within this paper. If you would like my code, send me an e-mail request. For a limited time my code is free, in exchange for your Benford's Law success stories.

REFERENCES

"Following Benford's Law, or Looking Out for No. 1",
Malcolm W. Browne,
http://courses.nus.edu.sg/course/mathelmr/080498sci-benford.htm

"The Power of One", Robert Matthews,
http://www.newscientist.com/ns/19990710/thepowerof.html

"Digital Analysis: a Computer-Assisted Data Analysis Technology for Internal Auditors", Mark J. Nigrini, Ph.D.,
http://www.itaudit.org/forum/emergingissues/f108ei.htm

ACKNOWLEDGMENTS

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Curtis A. Smith
Defense Contract Audit Agency
P.O. Box 20044
Fountain Valley, CA 92728-0044
Work Phone: 714-896-4277
Fax: 413-383-6395
email: casmith@mindspring.com

How American Express Saved $1M in CPU charges

Hermes Villalobos, American Express, Co., Miami Lakes, FL

ABSTRACT

There are many advantages of using code generators, for example, reducing the number of programming hours, thus allowing the SAS users to focus on analysis. Another example, in cases like this, is reducing the cost of CPU.

This project was designed with interaction on multiple platforms: Unix, Windows, and Mainframe. It is difficult to synchronize the programs running on these different platforms. In order to achieve some harmony among the programs, CRT, Microsoft Access, and FTP scripts were incorporated in to the project.

This paper will describe how to create a code generator and how to solve the problems related to the multiple platforms and multiple users.

INTRODUCTION

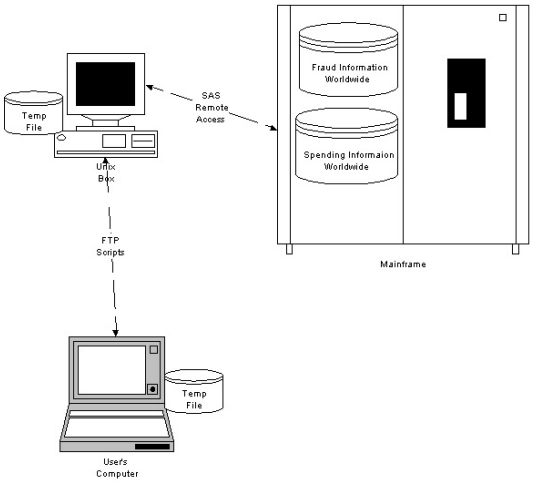
In 1997, the author was working as Senior Software Engineer for Fraud Prevention, analyzing large volumes of data from multiple sources and from multiple markets. All these analyses requests had something in common: all the layouts were pre-defined. Before starting a detailed analysis the author had to do some summarization of his information usually by country, city or industry. Then the author had to create a dataset with the information that he needed, then several procedures had to be run in order to analyze his information.

Next, the author researched and automated, as much as he could, the information extraction and analysis processes.

The author created a graphical user interface and SAS code generator in Microsoft Access. To transmit the programs to the Unix box the author used FTP scripts, and to submit the programs from Unix to the mainframe the author used CRT scripts.

ARCHITECTURE OF THE TOOL

The graphical user interface was located on the user's PC, the SAS programs ran on the Unix Platform, and the data repository was located on the mainframe as flat files.



HOW THE TOOL WORKS

The user interface generates a SAS program according to the user's input.

Then the user interface generates FTP script to transfer the program from the PC to the Unix Box.

Once the program is in the Unix box the user interface generates CRT script to submit the SAS program from the Unix.

This SAS program generates a temporary file with all the information requested. This is information that would be summarized or that could be the actual records from the databases.

Once the information arrives to the user's PC the information can be manipulated or exported to other media such as Excel, a SAS dataset in Unix, a flat file on the mainframe, or a comma-delimited file on the user's PC for further research.

SAS REMOTE ACCESS: ADVANTAGES AND DISADVANTAGES

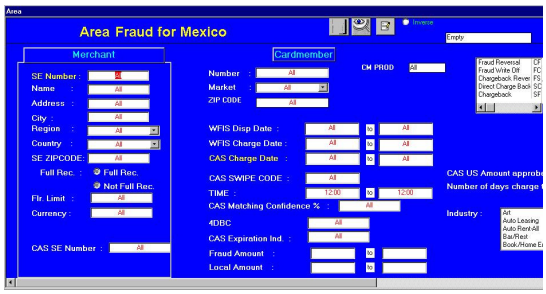
This tool generates and executes SAS programs in Unix. These programs access flat files remotely located on the mainframe. The main advantage of using this technique is that the CPU charges are zero. No mainframe programs are executed.

On the other hand, there is a restriction about the record length that can be accessed remotely. The author has developed tools with a maximum record length of 255 bytes and that still has good performance.

THE USER INTERFACE

The user interface allows the user to select the variables and criteria that the user needs to generate SAS programs according to need.

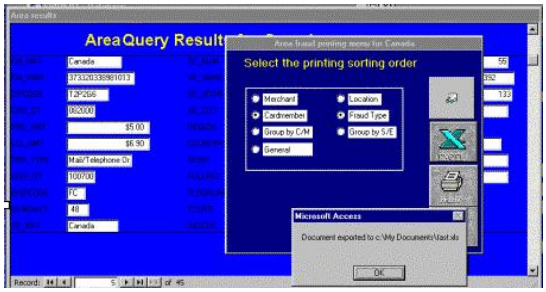
It also allows the user to focus on specific industries and cities.

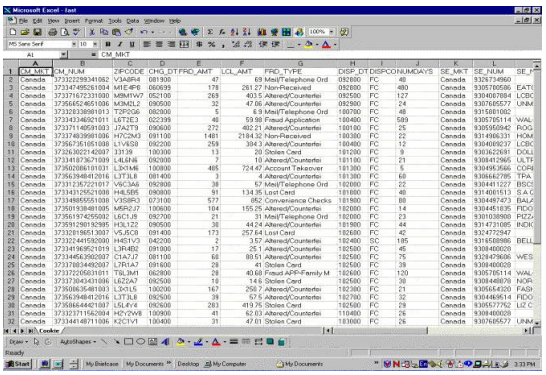


The user interface was developed in Microsoft Access. The core component is the code generator function.

Code is generated in SAS, FTP, and CRT to submit the programs.

Once the information is received, the user can massage the information, analyze it and export it to Excel.





Above is a sample of information exported to Excel.

SAS CODE

The SAS code generated by the user interface uses the remote access method.

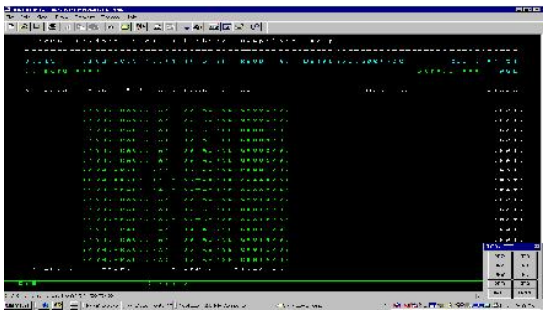
```
Filename my129 ftp "'FILENAME'"
host='IP ADDRESS' user='XXXXXX'
pass='XXXXXX'
RCMD='SITE RDW';

DATA nada129 ;
  infile       my129       RECFM=F
LRECL=298 ;
INPUT
 @1   CM_MKT         $EBCDIC14.
 @15  CM_NUM         $EBCDIC15.
 @30  ZIPCODE        $EBCDIC10.
 …
 ;
```

When a summarization is requested the SAS programs summarize the information using PROC SQL, and generate a flat file with a different layout.

DATA SOURCES ON MAINFRAME

All the information that the tool accesses, is based on flat files on the mainframe.

Here are some samples of such files. All the files have to be accessed since each one contains worldwide information.



IMPLICATIONS OF A MULTI USER TOOL

Since this tool was delivered worldwide, an algorithm was developed to avoid SAS programs and temp files from conflicting with one another.

The tool looks for the user's name on the system.ini file and generates all the programs and outputs based on the name found.

Since VILLA would be detected, for example, as the author's user name, then all the SAS programs and files are named using this name.

Here is a sample of FTP code generated to transmit a SAS program to Unix.

```
open IPAddress
User
Password
ascii
del VILLA.out
del VILLA.cas
del VILLA.in
del VILLA.tst
put c:\fast\fast.tst VILLA.tst
del VILLA.tst.log
bye
```

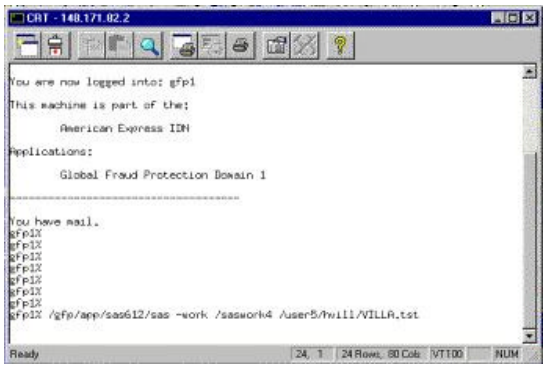
HOW THE TOOL SUBMITS A UNIX PROGRAM

As mentioned before the tool generates CRT code to submit the Unix program.

CRT it is a terminal emulator that allows the user to run scripts. It was developed by Van Dyke technologies.
Here is sample of CRT code.

```
// login1.csf
expect( "ogin: ");
send("MyUser\r");
expect( "assword: " );
send( "MyPass\r" );
expect( "gfp1% " );
send( "\r" );
send( "/gfp/app/sas612/sas -work
/saswork4
/user5/hvill/VILLA.tst\r" );
expect( "gfp1% " );
send( "\r" );
send("logout\r");
Disconnect();
CloseWindow();
```

This code logs in to the Unix box and submits the SAS program generated by the tool, in this case, VILLA.tst.



CONCLUSION

The code generators can significantly reduce the programming time meaning increase in productivity when analyzing data.

The remote FTP data access provided by SAS reduces to zero the CPU charges. This is particularly important in companies that must pay for every time they submit a JOB. The author has been using this method for 4 years and has never received a CPU charge.

Finally, you have to be careful with the input data formats when transferring data from one platform to another.

REFERENCES

Hermes Villalobos, May 2001

Avoiding a (Graphic) Identity Crisis with ODS HTML Styles

Jaclyn Whitehorn, The University of Alabama, Tuscaloosa, Alabama

ABSTRACT

The new SAS® Output Delivery System (ODS) enables any SAS programmer to produce professional-quality output. In particular, the ability to generate HTML output directly from every SAS procedure was highly anticipated by the SAS user community. However, it is doubtful that the default HTML output fulfills any site's Web publication standards. The capability of defining a custom ODS style is provided through PROC TEMPLATE, but it is not for the faint-of-heart. There are a number of new concepts to learn, and the amount of documentation is small but growing. Furthermore, there are some odd inheritance structures in the default templates provided by SAS Institute, which makes editing them a greater challenge than necessary. This paper is intended to serve as a "getting started" guide for those users that need to modify the default templates in order to create their own styles. It also introduces an alternative style inheritance structure.

INTRODUCTION

Imagine the following scenario: You have created your first HTML output using the new SAS Output Delivery System and are happily envisioning your new future without 200-page printouts, HTML macros, or cut-and-paste. However, your balloon is quickly deflated when the verdict comes down from your boss: "Well, it's kind of *ugly*. Can't you do something about that?"

An attractive Web page design generally consists of more than just various shades of gray. And even if your entire department is seriously artistically deprived and doesn't mind the gray, the institution you work for probably has pre-existing Web graphic standards. Furthermore, if it doesn't yet have standards, it probably will soon. So now your task is to take your new capability of producing HTML output and use it to create *attractive* HTML output, or at least output that conforms to your department's standards.

If you don't know where to start, or if you took one look at the PROC TEMPLATE documentation and the default style definition and threw up your hands in despair, this paper is for you. First, we will learn about the parts of a template. Then we will discuss the steps involved in creating your own style. Finally, we will consider ways to make the whole process easier by producing new default templates as starting points. While it cannot make you a full-fledged Web designer, hopefully this will put you on the right track for creating attractive, consistent HTML output.

REQUIRED PRIOR KNOWLEDGE

You should have a working knowledge of Base SAS and be able to produce some output that you would like to publish in HTML format.

While it is not strictly necessary, a good grounding in HTML will save you a lot of headaches later on. It is very difficult to tell the Output Delivery System how to write HTML if you do not know how to do it yourself. Even simple things like changing a font or color will be difficult unless you know how to create a Web page outside of the SAS System. Some experience designing Web pages will also help you make your styles more attractive and user-friendly, a skill that can only be learned by observing other Web sites and practicing making your own. If you think you will be doing extensive modifications, you should also learn about Cascading Style Sheets (CSS). There are many excellent HTML/CSS tutorials and references available both online and in hardcopy.

Finally, this paper will not discuss publishing HTML output to a Web server. This is mainly due to the wide variety of Web server configurations in use. Your institution's computer network support personnel should be able to assist you in publishing your results. If you are doing this privately, you will need an account with an Internet Service Provider (ISP) that includes space for a Web site and technical assistance for doing so.

CREATING ODS HTML OUTPUT

Before you can attempt to customize ODS output, you have to know how to create it. The following code fragment closes the LISTING destination, opens the HTML destination, and specifies where to send the output.

```
ods listing close;
ods html body='body.html' contents='toc.html'
 page='page.html'
 frame='frame.html'(title='My SAS Output')
 path='directory to put files in'(url=none)
 style=Beige;
```

Closing the LISTING destination means that no output will go to the output window or listing file. The four files named in the BODY=, CONTENTS=, PAGE=, and FRAME= options will be created in the directory given in the PATH= option. If files of those names already exist, they will be overwritten. The value of the TITLE= suboption for the frame file will appear in the browser title area and be used as the name for any bookmarks or "favorites" pointing to it, but it will not print on the page. The URL=NONE suboption is necessary to ensure that links between the four files do not use absolute references, which include the directory path, so that you can publish the HTML output to a Web server. The STYLE= option specifies which style template to use; this example uses the "Beige" template supplied by SAS Institute. If the STYLE= option is omitted, the output will be generated using the default style.

For those that are unfamiliar with HTML frames, some explanation may be useful. Each of the four files that the ODS creates (body.html, toc.html, page.html, and frame.html) is an HTML file. The body file is where all of the output will go. In fact, if you do not want a framed page, this is the only file that you need to request. The contents file is a table of contents to each piece of output. The page file is an index to the separate "pages" of the output file. These pages correspond to the separate pages that would have been created in normal output, not to where the HTML file will break when printed. You may have a table of contents, or a table of pages, or both, or neither. The frame file has no content of its own and is only needed if you have a contents or page file. It acts to hold all of requested pages in one browser window. To see your output, point your Web browser to the frame file. The default output style places the body file on the right and the table of contents and page index on the left (see Figure 1).

For more information about using the Output Delivery System and/or the HTML destination, see SAS Institute's *The Complete Guide to the SAS Output Delivery System*.

TEMPLATES AND TEMPLATE STORES

There are six types of templates utilized by the Output Delivery System: table, column, header, footer, tree, and style. Table, column, header, and footer templates control the display of output created by individual procedures and data steps. If you need to control the order of columns, the text used as column or row headers, or default data formats for a certain procedure's output, you will need to modify one or more of those template types. Tree templates are utilized internally by some procedures; users should never need to modify or create them. Style templates, on the other hand, control aspects of the overall presentation, such as background colors and default fonts. At this point we are only interested in creating a visual identity with style templates.

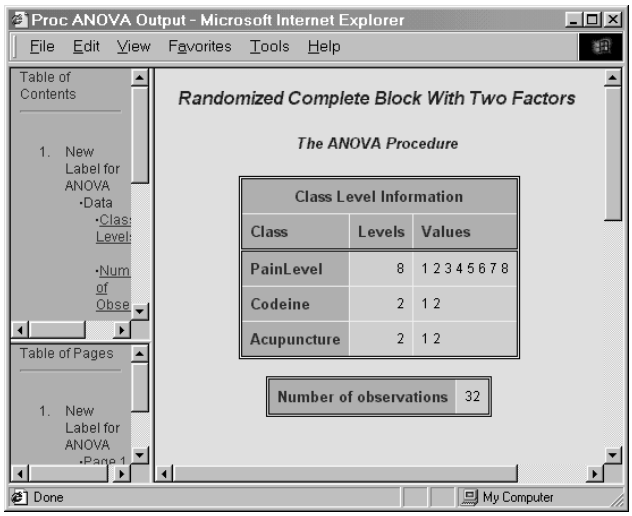


Figure 1: Default HTML output

Templates are saved in a template store, which is a special type of SAS file. An ODS PATH statement specifies which template stores are in use by that SAS session. In general, whenever a style is named for reading, the ODS searches each template store in the PATH, in order, until it finds one with that name. It uses the first one it finds and stops searching. When you need to store a style, the ODS will put it in the first template store in the PATH to which you have the proper access.

There are three different access levels assigned by the ODS PATH statement. "Read" access is the default and is given to any store named in the statement. "Write" access allows the user to write templates to the store, but it does not allow the user to *update* the store. This means that when a template is written to the store using write access, any templates that had previously been written to it are erased. "Update" access is required to add or change one template in the store without removing any existing templates. Keep in mind that users can issue their own ODS PATH statements at any time, so if you need to protect a template store, do so at the OS level.

If there will be more than one computer needing access to your new style, you should put it in a template store in an accessible place like a shared network drive. Then you will need to add the new template store to the ODS PATH statement for each computer that needs to use it to produce output; depending on your local configuration, this may be most easily accomplished with an autoexec.sas file.

The following code assumes that a template store exists in the library named on the first line. (We will be creating that template store later.) It gets added to the PATH with read access, which would be appropriate for general use. By listing it first, we tell ODS to search the new template store before the others. This means that output run without a style specified would look for a style named "styles.default" in the template store on the network drive before using the one supplied by SAS Institute. All pre-installed templates are stored in "sashelp.tmplmst".

```
libname ourstyle 'network directory';
ods path ourstyle.templat(read)
 sasuser.templat(update)
 sashelp.tmplmst(read);
```

HOW STYLE TEMPLATES WORK

A style template is primarily made up of style elements, each with attributes. Many elements serve to render a particular part of the HTML output, but some exist simply to provide a starting point for other elements. SAS Institute calls these "abstract" elements. For example, the "Cell" element in the default style is abstract, so it does not directly affect any visible output. However, the element "Data" controls the rendering of table cells containing data, and it is based on the "Cell" element. Other style elements that control special data cells are then based on the "Data" element. This is the first of two types of style *inheritance*; the second will be discussed shortly.

Style attributes control different aspects of each style element. Different attributes specify a font for the element ("font"), the font color ("foreground"), and a background color ("background"), to name a few. When one style attribute is based on another, it inherits all the attributes from the first one, but changes and additions can still be made.

Style templates are defined in a PROC TEMPLATE step. A STYLE statement is used in a PROC TEMPLATE step to define a style element. In the following example, the "Cell" element specifies a background and foreground color. The "Data" element inherits both of those, but then changes the foreground color. A data cell in output using this style would have a white background and gray text. (This code will not run by itself; it is part of a PROC TEMPLATE step.)

```
style Cell /
 background = white
 foreground = black ;
style Data from Cell /
 foreground = gray ;
```

Another type of style element used is a reference list. These are similar to associative arrays or "hashes" found in other languages like Perl. You can think of reference lists as providing nicknames for more complicated structures. One of the reference lists in the default style is called "fonts." This list sets up names like "docfont" and "TitleFont" to refer to particular font faces and sizes. In the following code fragment, "docFont" is set up to refer to font faces Arial, Helvetica, or Helv in HTML size 3, while the "TitleFont" is Arial, Helvetica, or Helv in HTML size 5, bolded and italicized. Then the "Cell" element has its font attribute set to "docFont," which is then inherited by the "Data" element.

```
style fonts /
 'docFont' = ("Arial, Helvetica, Helv", 3)
 'TitleFont' = ("Arial, Helvetica, Helv",
               5, Bold Italic) ;
style Cell /
 background = white
 foreground = black
 font = fonts('docFont');
style Data from Cell /
 foreground = gray ;
```

Reference lists are also used in the default style to define shortcuts to colors and to commonly used text and HTML.

For the second type of style inheritance, a style definition begins by declaring a "parent" style. The style being defined is the "child" of the parent style and inherits all of its elements. New elements can still be defined, and existing elements can be changed within the child style. The first way of doing this is to use the STYLE statement like before. The STYLE statement only changes the element named, *not any that may inherit from it.* Usually you will base an element in the child style on the element of the same name from the parent style. This allows you to only include the attributes that need to be added or changed. The following complete PROC TEMPLATE step bases a new style on the default and changes the font in the data cells.

```
proc template;
define style styles.new;
 parent = styles.default;
 style Data from Data /
  font = ("Times New Roman, Times, serif",3);
end;
run;
```

If you want other elements to inherit attributes that you set in the child style, you need to use the REPLACE statement instead of the STYLE statement. When you replace an element, the style template is built as if the element in the child template completely replaces the element of the same name from the parent template. Because of this, any attribute set for that element in the parent style will need to be reset in the element that is replacing it, even if it otherwise would not be changed. However, you can still base the new element definition on one from the parent style in order to preserve an inheritance structure. The following example replaces the

background color to the element “Data,” ensuring that any element that inherits from it gets the new background color. It inherits from the “Cell” element, which is an abstract element from the default style. The new “Data” element has all the attributes from “Cell,” plus the new foreground and background colors. It will not have any other attributes that may have been named in the “Data” element in the default style. (Note that this example uses a hexadecimal code to represent the color. This is used much more frequently than a color name.)

```
proc template;
define style styles.new;
 parent = styles.default;
 replace Data from Cell /
  foreground = colors('datafg')
  background = cxFFCCCC; /* light pink */
end;
run;
```

READING THE DEFAULT STYLE

We now have most of the information needed to start creating a style of our own. However, most people will not make one from scratch. There are a large number of elements, many of which you may never work with, that are needed in order to create output using a style template without warnings or notes in the log. To avoid these problems, it is best to base your style off of one that already has all the elements in it, like the default template provided by SAS Institute. You need to know how the template is defined before trying to edit it, so use the following code to save the source for the default template:

```
proc template;
source styles.default /
 file='external file location';
run;
```

You will see that the source code is quite long and can get very confusing. All the same, it is important to read through it to get some idea of the inheritance structure. Some of the more important style elements are listed in Table 1.

Some of the elements, like “SystemTitle” and “ProcTitle,” have associated “containers” as well. The Output Delivery System places each of these elements inside its own one-cell table, and each table is rendered by a container element that describes its width and border. The background color is controlled by the main element, not the container. The “SysTitleandFooterContainer” element holds the “SystemTitle,” while the container for “ProcTitle” is the “TitleandNoteContainer.” Figure 2 shows the position of many style elements in relation to the rest of the output. There is also an interactive guide to the style elements available at the author’s Web site, the address of which is given in the contact information section.

More details about the default style template supplied by SAS Institute can be found under PROC TEMPLATE: Concepts in *The Complete Guide to the SAS Output Delivery System*.

BEGINNING YOUR OWN STYLE

Assuming that you have already submitted the two statements under “About SAS Templates” to assign a library and set the ODS path, start your PROC TEMPLATE with the following:

```
libname ourstyle 'network directory';
proc template;
path sashelp.tmplmst(read);
define style styles.test /
 store=ourstyle.templat;
 parent = styles.default;
```

The PATH statement temporarily changes the ODS PATH used; this ensures that the “styles.default” that the style is based on is the one supplied by SAS Institute. The STORE statement saves the “styles.test” template in the shared template store.

In order for other people to be able to help you test this style, you must first have set up their ODS PATH statements as detailed above. Then, they must use the STYLE=styles.test option in their ODS HTML statement. Once you have your style completed, tested, and approved if necessary, you may want to name it “styles.default.” Since the ODS PATH statement names “ourstyle.templat” first, any time ODS is used to produce HTML output, it will find and use your default style instead of the one from SAS Institute. Therefore, the STYLE= option in the ODS HTML statement will no longer be necessary.

After the PARENT statement, include any STYLE or REPLACE statements you need to customize the output. The abbreviated list of style element attributes in Table 2 may be useful to you. Keep in mind that not all attributes make sense with all elements. “Mismatched” attributes will not give errors; they will just not have any effect.

ALTERNATE INHERITANCE STRUCTURES

Some people have found the inheritance structure built into the style templates provided by SAS Institute to be non-intuitive. While some of the complexity stems from the inherent flexibility of ODS output, alternate style templates can be created to better facilitate customization.

A major problem with the current structure is that most elements do not inherit defaults from the file in which they are contained. Many people see headings as part of the body file and cells as part of a table. However, in the default style, no headings inherit from the “Body” element, and the “Data” element doesn’t inherit from “Table,” even indirectly. This means that you may have to change font faces or colors in more places than you might think necessary. While the places to make the changes are readily apparent from studying the source code for the default template, this type of problem creates a steeper learning curve and may discourage some users from attempting to create their own styles.

Another issue is the use of reference lists. There are two just for colors alone. While the two-level redundancy may be more efficient for higher-level programmers, it adds unneeded complexity to the process for many users. Multiple pieces of output that may look related and be the same color by default can actually refer to two different entries in the “colors” reference list, thus breaking some of the benefits of inheritance.

In an attempt to make it easier for more users to create their own templates, the author has produced two templates featuring an alternate inheritance structure. One is a stripped-down version of the SAS default; most users will see very little cosmetic differences in output. The other has very few attributes and can be used as an inheritance framework. Those templates and their documentation can be found at the Web site given in the contact information below. The author encourages feedback on these templates and will attempt to keep them up-to-date with needed improvements.

CONCLUSION

Like the rest of The SAS System, the Output Delivery System is extremely powerful and flexible. Through PROC TEMPLATE, users have the ability to customize the overall style of HTML output so that it fits with the other material on their Web sites. This is not a simple process, but the result is worth it: a reusable style to quickly get any SAS output you or your team produces ready for publication on the World Wide Web.

There is plenty of room for innovation within the Output Delivery System. While the author hopes that her templates can be of some use, they should also initiate a discussion about other possible inheritance structures. There is also a need for additional style templates that could be used either as-is, or as starting points for additional modification. As of May 2001, there were no user-developed styles available in SAS Institute’s ODS Style Gallery <http://www.sas.com/rnd/base/index-gallery.html>. The ability to create professional output directly from SAS has been anticipated for some time; now the very talented population of SAS programmers should expand on it.

REFERENCES

A comprehensive source of reference information on the Output Delivery System (including PROC TEMPLATE) is:

SAS Institute Inc., *The Complete Guide to the SAS Output Delivery System*, Cary, NC: SAS Institute, Inc., 1999

Other available sources:

The Template FAQ, http://www.sas.com/rnd/base/topics/templateFAQ/Template.html

Olinger, Christopher, "Twisty Turny Passages, All Alike – PROC TEMPLATE Exposed," *Proceedings of the Twenty-Fourth Annual SAS Users Group International Conference*, 1999

The Base SAS Community Web site, http://www.sas.com/rnd/base/

The output used for Figure 1 is a basic ANOVA adapted from an example in the SAS/STAT® User's Guide portion of SAS Institute Inc., *SAS OnlineDoc® Version 8* , Cary, NC: SAS Institute Inc., 1999. It was modified so that it would only produce HTML output.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Jaclyn Whitehorn
The University of Alabama
Box 870346
Tuscaloosa, AL 35487-0346

Work Phone: (205) 348-8720
Fax: (205) 348-3993

E-mail: jaclyn@bama.ua.edu
Web: http://bama.ua.edu/~jaclyn/sasods

TABLES

TABLE 1: SELECTED STYLE ELEMENTS

Style Element	Description
Body, Frame, Contents, Pages	Control appearance of the various output files (background, default font, etc.)
ContentTitle, PagesTitle	Control the titles for the Table of Contents and Pages files.
SystemTitle	Controls appearance of system-provided titles (such as those produced by TITLE statements)
ProcTitle	Controls appearance of procedure-defined titles
Header, RowHeader	Renders table cells that act as column or row headers
Data	Renders table cells that contain data
ContentProcName, PagesProcName	Controls appearance of procedure names in Table of Contents and Pages files. (If you use the ODS PROCLABEL statement to change the text used to label the procedure, you will want to edit the ContentProcLabel and PagesProcLabel elements instead.)

TABLE 2: SELECTED ELEMENT ATTRIBUTES

Attribute	Description
font	Set the font for the element. Requires font list in the form ("font-face list", font-size, keyword list). Font faces should be separated by commas. Sizes are given in HTML form, 1 through 7. Keywords pertain to font weight and style, such as bold or italic. A reference to the "font" list can also be used.
background	Background color. For tables, sets the color between cells.
foreground	Usually controls font color, border color for containers. (There is also a bordercolor attribute, which can also be used if the border is turned on. Borders do appear differently in Internet Explorer and Netscape; test thoroughly!)
frame	Outside border for tables/containers. Common settings are "VOID" (no border) or "BOX" (border all the way around).
linkcolor, visitedlinkcolor	Sets colors for unvisited and visited links. There is apparently no attribute for active links.
cellspacing, cellpadding	Sets the space between and within cells, respectively. The color of the cellspacing is set by the table background, while the padding is controlled by the cell background.
pretext, posttext, prehtml, posthtml	Inserts text or HTML code before or after the element. Note that the use of "before" or "after" can be a little deceptive. The text of the "ContentTitle" is contained in "pretext," while you can use "prehtml" with "Body" to insert a custom page header.

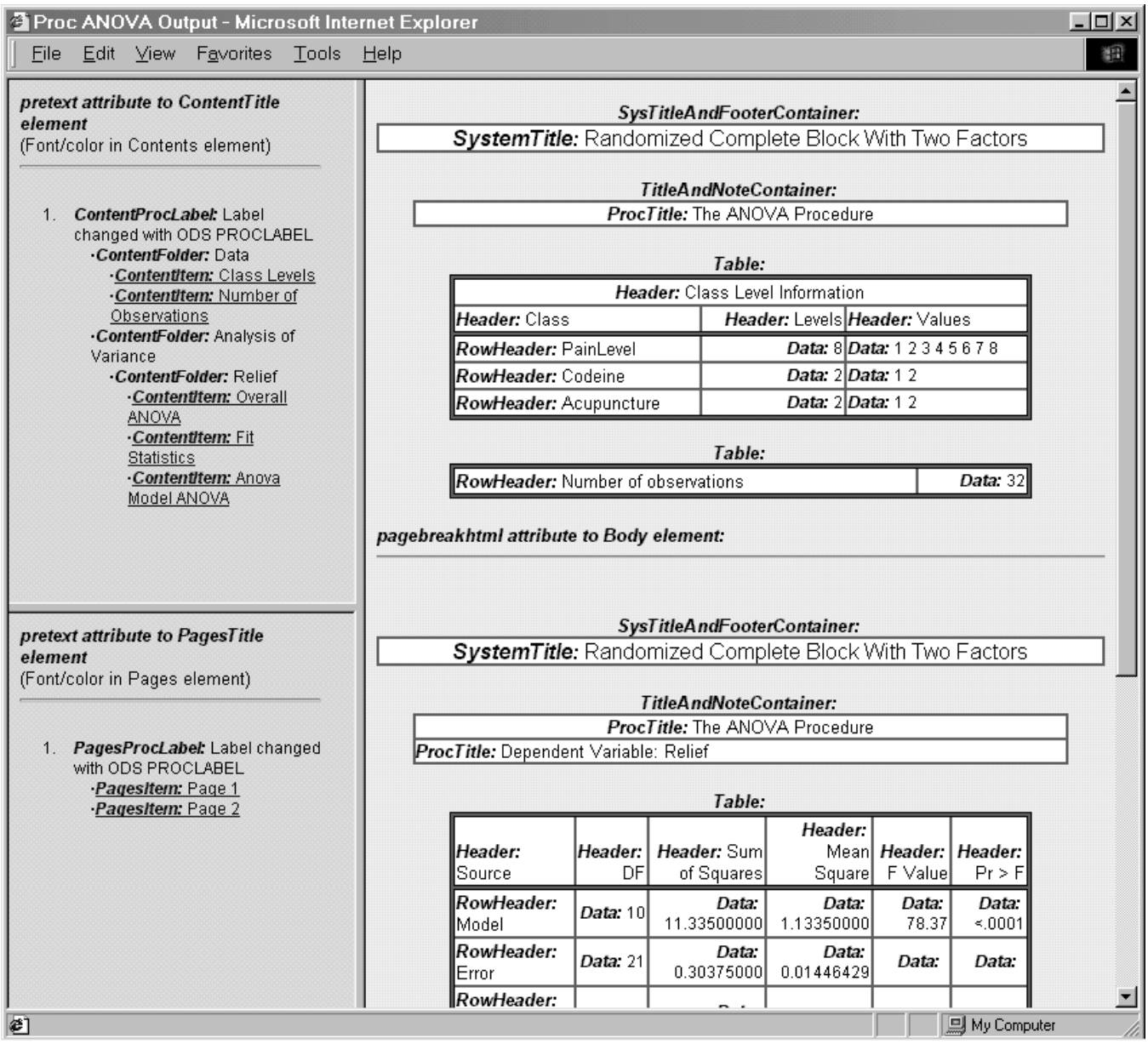


Figure 2: HTML output labeled with attribute names

Implementing Digital Data Analysis Using SAS®

Thomas J. Winn, Jr.
Texas State Auditor's Office, Austin, Texas

Abstract

Benford's Law is an empirically-verified mathematical formula which describes the expected distribution of digits in a collection of everyday numbers. Since most people aren't familiar with these theoretical frequencies, it often is possible to compare an actual distribution of digits with the Benford probabilities, in order to determine if the data were contrived (invented, fraudulent, or rounded) somehow. The paper describes the SAS code that was written by the author for use at the Texas State Auditor's Office.

Introduction

Benford's Law says that the probability of the first digit being a "d", P(D1=d) = log10(1 + (1/d)), where d=1,2,3,...,9. This phenomenon concerning everyday numbers was first observed empirically by astronomer Simon Newcomb in 1881, and afterwards by physicist Frank Benford in 1938. Benford referred to it as "the law of anomalous numbers". This mathematical property concerning first digits can readily be extended to provide probabilities for second digits, first and second digit combinations, etc.

For Benford's Law to apply to a set of data, the data must have the following characteristics:

(1) the numbers should represent the sizes of similar phenomena,
(2) there should be no built-in maximum or minimum values,
(3) the data should not consist of assigned numbers, that is, numbers that are used for naming purposes, but without quantitative meanings (for example, telephone numbers or social security numbers),
(4) there should be more small items than large items. (Moreover, the mean would be greater than the median, and the skewness value would be positive.)

Interestingly, in 1961, Roger Pinkham proved that the distribution of first digits is scale invariant under multiplication, so that multiplication of all of the numbers in a Benford set by a nonzero constant would result in a new set of numbers, which also would follow Benford's Law.

In 1992, Mark Nigrini's doctoral dissertation observed that many types of accounting data, including sales figures and expenditure claims, follow Benford's Law. He also showed how deviations from the expected distribution might be detected using statistical tests. Nigrini's work is the beginning of using Benford's Law for fraud detection in auditing. It should be pointed out that Mark Nigrini sells DATAS, an application software package that he wrote for performing digital data analysis. Nigrini's book and his DATAS package set forth certain other statistical methods, which are not implemented in the program described in this paper. These are the number duplication test, the number frequency factor test, the relative size factor test, and the distortion factor model.

Early in 2001, the Texas State Auditor's Office became interested in implementing this approach, and in developing its own computer programs for auditing purposes. I developed the SAS programs that are used at the State Auditor's Office, independently of the DATAS programs. However, I acknowledge that I closely followed the procedures that were outlined in Nigrini's book. This paper describes the primary program, the SAS program which performs the most fundamental digital data analysis tests. Besides the program herein described, I also have written other digital data analysis programs that are used at the S.A.O.

Explanation of the Program

The program begins by reading in the data which are to be analyzed for consistency with Benford's Law. Only positive data values are selected. If any negative data values exist, these would be analyzed separately. Various combinations of digits (first, second, first two, first three, and last two digits, as well as the decimal fractional components) are stripped from the data values, using various SAS character-string manipulation functions. Then PROC FREQ is used to compute the frequencies and proportions for each digit combination, and PROC SUMMARY is used to count the total number of observations that went into each frequency table.

Using Benford's Law and its logical extensions, expected probabilities are computed for all of the first, second, first-two, and first-three digital combinations. Theoretically, each of the last-two-digital combinations and decimal fractional components would be equally likely.

Next, the actual and expected frequency distributions for the digital combinations are combined and compared. For each digital combination under consideration, the differences between the actual and expected proportions are calculated. And z-statistics are computed, as a measure of the statistical significance of each of the differences.

The z-statistic is computed using a formula that is found in Nigrini's book (on pages 42 and 73). It is as follows:
zstat =
(abs(p-benfp)-(1/(2*num)))/sqrt((benfp*(1-benfp))/num).
The 1/(2*num) term in the numerator is a continuity correction term and is only used when it is smaller than the absolute difference which appears as the first term in the numerator. Then, 95% upper and lower tolerance limits for the expected proportions are calculated. The upper and lower bounds used in the tolerance intervals are based on the z-statistic. Nigrini (p. 43) asserts that, for values of the z-statistic which are greater than 1.96, the difference between the actual and the expected proportions will be significant at the 5

percent level (and the z-statistic cutoff at the 1 percent significance level would be 2.57).

Just before printing the reports, the mean absolute difference between the actual and expected proportions is computed. The mean absolute difference is an unweighted average of all of the absolute differences for each digit-combination under consideration. Nigrini says that this is the best goodness-of-fit measure for use in an auditing context (page 79). It has the advantage of being easy to understand, and it does not vary according to the size of the data set. Another of my SAS programs compares the mean absolute differences associated with the values of a BY-variable which is used for classification purposes, listing the mean absolute differences in descending order.

Finally, the reports are produced. These include comparative listings of the actual vs. expected proportions for each instance, and also include the computed differences, z-statistics, and 95% tolerance intervals. Some graphical outputs also are generated, to help analysts to visualize some of the digital data analysis results.

Interpreting the Data

As with any statistical procedure, before running the digital data analysis program, it is a good idea to use standard exploratory data analysis methods to identify the general character of the data, including any outliers. The EDA plots that are produced by PROC UNIVARIATE are particularly helpful.

Above all, one should use good sense in examining digital distributions. Just because a report appears to contain some statistical results is not a good justification for the abandonment of critical thinking. A pitfall of digital data analysis is that sometimes the computed statistics can be misleading.

Data sets containing only a few hundred, or even a few thousand, records may fail to conform to Benford's Law just because of their smallness.

The z-statistic is only one clue among many regarding the identification of numbers that are fraudulent or invented. The z-statistic may not be entirely reliable, particularly with small datasets. A few apparent instances of "significant" noncompliance with Benford's Law may not be particularly material. However, if the instances of noncompliance are both large and numerous, then one might be suspicious about how they were generated.

The z-statistic suffers from an excess power problem. For large data sets, small differences may be flagged as being statistically significant even when the differences may be immaterial. If the actual proportion is reasonably close to the expected proportion, disregard the z-statistic and the associated tolerance intervals.

In the context of selecting cases for auditing, negative deviations from Benford's Law generally are not important (Nigrini's book, p. 43).

The first- and second-digit tests both are high-level tests of reasonableness. Unless there are blatant problems with the data, the first-digit test probably won't turn up anything. The second-digit test may indicate some problematic data, but because there generally would be a large number of observations having the same second digit, by itself this test probably would not be an efficient tool for selecting specific cases for examining.

The first-two-digits test is performed to identify anomalies in the data that may not be apparent from the separate first- and second-digits tests. Very large positive deviations would indicate excessive duplications, and could be used to specify a coarse selection criteria of items for further consideration. The first-three-digits test is designed to provide a finer amount of precision than the first-two-digits test. It can be used to narrow the focus of attention in selecting cases for a careful examination.

The last-two-digits and decimal-fractional-components tests are designed to find invented numbers and rounded numbers. Rounded numbers may signal that the numbers were estimated, or were arbitrarily assigned. If the data to be tested all contain "dollars and cents" magnitudes, then the last-two-digits and the decimal-fractional-components tests would yield identical results. If excessive number duplication is suspected, then the data could be sorted and subsetted so as to identify the most frequently recurring numbers.

Conclusion

This paper presents a SAS program that performs an analysis of the distribution of certain digital combinations occurring in a batch of data values. It compares the actual distributions with the expected frequencies according to Benford's Law. Since most people might expect the distribution of digits to be uniform, this empirical distribution has been useful in detecting fraud.

My SAS code is available upon request.

Suggestions for Further Reading

Benford, Frank (1938). "The Law of Anomalous Numbers," Proceedings of the American Philosophical Society, Vol. 78, No. 4, pp. 551-571.

Browne, Malcolm W. (1998). "Following Benford's Law, or Looking Out for No. 1," The New York Times, August 4, 1998.

Feller, William (1966). An Introduction to Probability Theory and Its Applications, Volume II, p. 62, John Wiley & Sons, Inc., New York.

Matthews, Robert (1999). "The Power of One," New Scientist, July 10, 1999, pp. 26-30.

Newcomb, Simon (1881). "Note of the Frequency of Use of the Different Digits in Natural Numbers," American Journal of Mathematics, Vol. 4, pp. 39-40.

Nigrini, Mark J. (1992). "The Detection of Income Tax Evasion Through an Analysis of Digital Distributions," Ph.D. dissertation, University of Cincinnati.

Nigrini, Mark J. (2000). Digital Analysis Using Benford's Law: Tests and Statistics for Auditors, Second Edition, Global Audit Publications, Vancouver B.C.

Pinkham, Roger S. (1961). "On the Distribution of First Significant Digits," Annals of Mathematical Statistics, Vol. 32, pp. 1223-1230.

Author Information

Tom Winn
CAATs Team
Texas State Auditor's Office
P.O. Box 12067
Austin, Texas 78711-2067

Telephone: 512 / 936-9735
E-Mail: twinn@sao.state.tx.us

Appendix – Output Sample (Selected Reports)

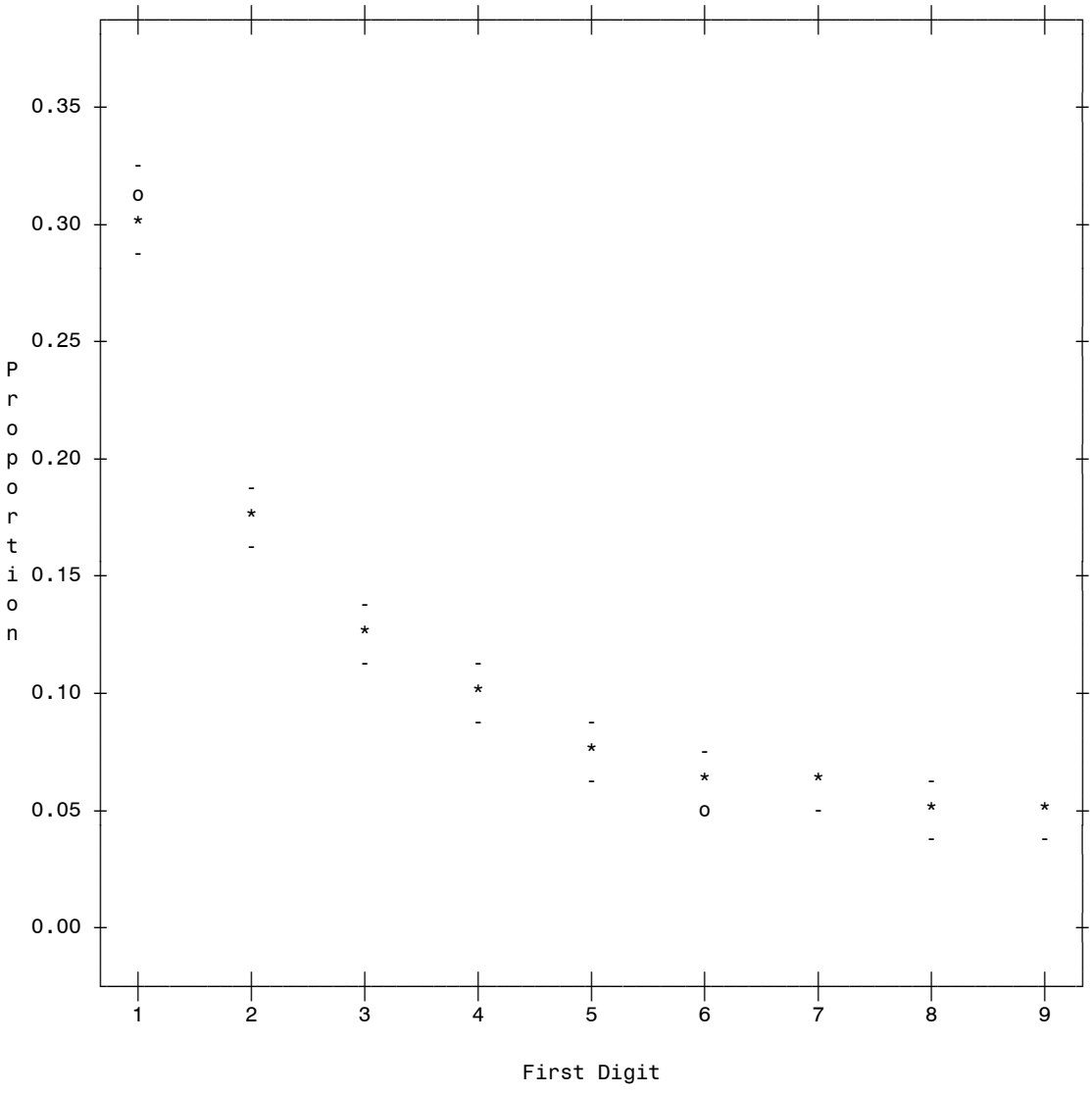
Analysis of the Frequency Distribution of First Digits
of the Positive Values in the Data
NOTE -- The Mean Absolute Difference is 0.00659

First Digit	Actual Count	Actual Proportion	Expected Proportion	Difference	Lower Bound	Upper Bound
1	774	0.30985	0.30103	0.00882	0.28284	0.31922
2	422	0.16894	0.17609	-.00716	0.16095	0.19123
3	335	0.13411	0.12494	0.00917	0.11177	0.13811
4	258	0.10328	0.09691	0.00637	0.08511	0.10871
5	211	0.08447	0.07918	0.00529	0.06839	0.08997
6	134	0.05364	0.06695	-.01330	0.05695	0.07695
7	138	0.05524	0.05799	-.00275	0.04863	0.06736
8	117	0.04684	0.05115	-.00432	0.04231	0.05999
9	109	0.04363	0.04576	-.00212	0.03736	0.05415
	2498	1.00000	1.00000			

First Digit	Z-Statistic	Comment
1	0.93898	within 95% tolerance interval
2	0.91273	within 95% tolerance interval
3	1.35564	within 95% tolerance interval
4	1.04281	within 95% tolerance interval
5	0.94143	within 95% tolerance interval
6	2.62043	significantly different, at 5% level
7	0.54477	within 95% tolerance interval
8	0.93352	within 95% tolerance interval
9	0.45982	within 95% tolerance interval

Analysis of the Frequency Distribution of First Digits
of the Positive Values in the Data
(*=Expected, o=Actual, bars(-) represent 95% tolerance interval)

Plot of benfp1*d1. Symbol used is '*'.
Plot of lower95*d1. Symbol used is '-'.
Plot of upper95*d1. Symbol used is '-'.
Plot of p1*d1. Symbol used is 'o'.



NOTE: 10 obs hidden.

SAS Solutions & Vertical Products

Section Chairs

Matt Becker
PharmaNet, Inc

E. Barry Moser
Louisiana State University

Kasi Peek
Blue Cross Blue Shield of TN

SAS Solutions & Vertical Products

OLAP Best Practices

What You Need to Consider When Building and Deploying an OLAP Application

Greg Henderson, SAS Institute, Cary, NC

ABSTRACT

SAS/OLAP Server® Software is a powerful, flexible and scalable OLAP solution. It's hybrid OLAP (HOLAP) architecture provides the flexibility to store data in whatever format is most appropriate based on the characteristics of the data and how the data will be used. This also means, however, that choices must be made as to what is the most appropriate data model for a successful OLAP application. This paper will identify and discuss the issues that need to be taken into account when designing and deploying an OLAP application.

INTRODUCTION

As OLAP applications have evolved and matured from simple sales analytics to other analytic areas throughout the organization, OLAP tools have also had to mature to meet the ever-increasing demands for aggregate analysis. Nowhere is this more evident than in the recently conceived areas of e-intelligence and web analytics. The nature of data generated by the web has put a tremendous strain on many systems, OLAP included. When you consider the vast amounts of data generated, combined with the inherently high cardinality of that data, many traditional OLAP technologies just cannot scale up to the challenge.

To answer these increased demands, SAS Institute made a strategic decision late in the version 6 development cycle to extend its traditional multidimensional data base engine, SAS/MDDB Server® software, to support a new Hybrid OLAP (HOLAP) architecture that would provide the necessary scalability and flexibility to answer the demands of a web driven world. In version 8, the HOLAP architecture has been completely integrated into the SAS System, and now carries a new name, SAS/OLAP Server® software.

As OLAP technology evolves and matures, so too must our thought processes, data modeling strategies and presumptions. Although there is not enough space in this document to cover every minute detail of the process of building successful OLAP data models, the intent is to get the reader to think in new ways. For, with the new HOLAP infrastructure, we are no longer limited by technology, but only by our imagination and creativity.

WHAT IS OLAP?

No OLAP document would be complete without first defining what is meant by OLAP. In a formal sense, OLAP (OnLine Analytical Processing) is defined as fast access to large amounts of summarized data. Implied in this definition is the concept of dimensionality. For without dimensions, there would be nothing to summarize the data by. Thus, a more generalized definition might be the ability for users to quickly interrogate large amounts of data, at varying levels of detail, across a variety of combinations business dimensions.

OLAP is full of a myriad of terms and acronyms, which are often ill defined. Thus, before digging too deeply into the bowels of OLAP, it is prudent to provide a few basic definitions of some of the OLAP terms that will be used in this paper:

Dimension – A business perspective that is useful for analyzing data by or across. Often times referred to as a hierarchy. Examples are Time, Product or Geography.

Level – Dimensions are often made up of various levels of detail. For example, the Time dimension may consist of Year, Quarter and Month levels. Some dimensions will only have one level, in which case the level is implied. When referring to the physical representation of data, a dimension level is sometimes referred to as simply a dimension, or for those familiar with SAS terminology, a class variable.

Member – A given value of a dimension level. For example, members of the Year level of the Time dimension could be "1998", "1999", and "2000". The number of unique members at any dimension level is referred to as the **cardinality** of that dimension level.

Measure – The ultimate business measure that is being aggregated. For example, Sales or Profits. Measures also have statistics associated with them such as Sum, Count, Average, etc. Those familiar with SAS terminology may refer to these as analysis variables.

OLAP DATA STORES

What really differentiates OLAP from other query and reporting systems is the idea of fast access (or high performance), combined with large amounts of data. Traditional query and reporting systems just are not designed for this level of online performance. Although the definition for OLAP does not explicitly specify any type of storage scheme for the underlying data, it is presumed that in order to accomplish the desired performance, some specialized data stores are required.

These specialized data stores contain data that is pre-summarized, or aggregated, across those combinations of dimensions that users want to see. Thus, an OLAP database can be visualized as a series of subtables, where each subtable represents a specific combination of dimension levels. Within each subtable, the rows represent each unique combination of the members of the dimension levels for that subtable. The columns represent the aggregated values of the measures for each unique combination.

For example, let's consider a simple situation where there are two single level dimensions, Year and Country, and one measure, Sales. If these are aggregated into a multidimensional database, the result is potentially 3 subtables – one for the summaries of each Year, one for the summaries of each Country, and one for the cross tabulation of Country by Year. If there are 2 years of data, and 3 countries, the Year subtable would contain 2 records, the Country subtable would contain 3 records, and the Country * Year subtable would contain 2*3 = 6 records. Table 1 illustrates what the resulting subtables would look like.

Year	Sales
1999	$850,000
2000	$1,060,000

Country	Sales
US	$1,300,000
Canada	$500,000
Mexico	$110,000

Year	Country	Sales
1999	US	$600,000
1999	Canada	$200,000
1999	Mexico	$50,000
2000	US	$700,000
2000	Canada	$300,000
2000	Mexico	$60,000

Table 1: Simple Aggregate Subtables

Note the word "potentially" in the previous paragraph. Again, the definition of OLAP does not mandate that all possible combinations are stored as persistent aggregations (ie. physical subtables). As long as the Year * Country subtable exists, Year and Country aggregates could be independently derived from the Year * Country subtable at run time. In this case, that would involve rolling up only 2 or 3 values respectively, so there would not be any significant performance degradation.

In a more generalized sense, the statement can be made that the only subtable that is required in an OLAP database is the one that crosses all of the dimensions levels, which we call the N-Way subtable. Most real world databases, however, are not as simple as the one described above, and to get acceptable performance, some subtables need to be pre-aggregated and persisted in the OLAP database. A good portion of this paper will be spent discussing how to determine which subtables to persist in order to obtain a good balance between performance and storage requirements.

STORAGE ARCHITECTURES

Once it is determined which aggregations to persist in the OLAP database, decisions must me made as to how to physically store that data. Traditionally, OLAP systems have operated on top of one of two underlying data architectures, multidimensional OLAP (MOLAP) and relational OLAP (ROLAP). Both of these architectures provide the capability of presummarizing data across the various dimensions that users want to see. The differences have to do with performance, scalability, simplicity, and resource utilization.

The earliest OLAP engines were primarily multidimensional (MOLAP) in nature. This required a specialized data store that would hold the aggregated data in a format whereby it could be easily retrieved by multidimensional queries. Most MOLAP's use a single operating system file to store the entire database, and have indexing implied and built into the structure.

In the early 1990's, however, the concept of a dimensional data model that could be represented in a relational database (RDBMS) was presented by Ralph Kimball. The basis of this concept lies in the star schema, and its derivative, the snowflake schema. As this concept gained in popularity, some vendors began to structure OLAP architectures on top of the star schema model, and thus was born the ROLAP architecture.

ROLAP OR MOLAP, WHICH IS BETTER?

As these two technologies evolved, a large debate began as to which was the superior architecture for OLAP applications. There were passionate opinions on both sides, but in reality each had it's own benefits and drawbacks. Which was best ultimately depended upon the underlying reporting requirements and the nature of the data.

MOLAP engines were preferred by many due to their simplicity. Because all of the aggregation rules, relationships and indexing were inherent in the data structure, they were very easy to build and maintain. In addition, they were often much smaller and much faster than comparable ROLAP architectures. There was, however, one key drawback. The MOLAP model was not highly scalable. As the amount of aggregated data increased, the simplicity of the single file architecture thus became its nemesis.

Especially problematic for MOLAP's was high cardinality data. Relational databases do a much better job of storing and retrieving small subsets of large data. Thus, the star schema and ROLAP architecture was often more suited for high cardinality data. In fact, many OLAP applications were forced into being implemented as ROLAP's simply due to the cardinality of one key dimension, often times a customer or product dimension.

In addition to scalability, many IT shops preferred ROLAP's because they used technology that most IT professionals were already familiar with, the RDBMS. In addition, if the data were stored in an RDBMS, it would be open for use in non-OLAP applications as well.

The following table highlights the key benefits and drawbacks to each approach:

Architecture	Benefits	Drawbacks
MOLAP	Fast	Not Scalable
	Small	Unknown Technology
	Easy to Maintain	
ROLAP	Very Scalable	Difficult to Maintain
	More Familiar Technology	RDBMS Overhead
	Flexible	

Table 2: OLAP Architecture Comparison

HYBRID OLAP (HOLAP), THE POWER TO CHOOSE

A true hybrid OLAP approach lets the application designer choose a combination of ROLAP and MOLAP architectures based on the reporting requirements of the user, the system resources available, and the nature of the data. For example customer and product dimensions are sometimes problematic in a MOLAP architecture due to their high cardinality. In a HOLAP architecture, these dimensions can be stored in a more scalable ROLAP schema, while all other dimensions are stored in a more manageable MOLAP architecture.

In addition to having the flexibility to store the data in different underlying architectures, the SAS HOLAP architecture allows each piece of the model to be stored on a separate computing platform, thereby further increasing the scalability across host systems.

Because of this increased flexibility, the application designer must make choices about not only what subtables to store in an OLAP database, but also how to physically store them. The next few sections will discuss some guidelines on how to make these choices. Since each set of choices has tradeoffs, it is imperative that the application designer understand the nature of the data that is being put into the model, as well as how the end users will be using the model. A data and business requirements assessment, thus, becomes one of the most critical steps in the application design process.

OPTIMIZING THE MODEL: DETERMINING PERSISTENT AGGREGATES

The introduction briefly touched on the idea of persistent subtables in an OLAP data model. On one extreme, the model could be optimized for minimum disk space, in which case we would only store one persistent subtable, the N-Way. This would have the benefit of a small data store, most likely at the expense of degraded performance since most rollups and cross tabulations would have to occur at run-time of reports.

On the other extreme, every possible combination of dimension levels could be persisted as a series of subtables. In theory, this would maximize performance; however, in practicality this is not true due to the huge size and complexity of indexing required to implement such a scheme. Although this technique might apply to the simple data model laid out in the introduction, real world models will typically have many more dimensions and dimension levels than this. The number of possible subtables for any OLAP model can be defined as

$$2^n-1$$

where n is the total number of dimension levels that exist in the model. Thus, for a model with 20 dimension levels, there would be over a million possible subtables!

To further illustrate how the above formula is derived, consider a binary example whereby each dimension level represents a digit within a binary number. The number of digits is equal to the total number of dimension levels. To represent each possible combination of dimension levels, it's binary digit can either be "1" indicating it is present, or "0" indicating it is not present. This would yield 2^n possible combinations. Since there would be no use in a subtable that didn't include any dimension levels, 1 is subtracted for the case where all digits would be "0". For those familiar with PROC SUMMARY, this binary representation is how the _TYPE_ variable is derived.

So, now that it is understood that persisting every subtable nor only persisting the N-Way is optimal for most OLAP models, let's consider some other techniques for determining which subtables to persist in a well-optimized OLAP database. Which technique is best primarily depends upon how users are going to be reporting against the data. The goal, however, is to persist those subtables that are going to be accessed most often, and minimize the number of times that the lowest level subtables (ie. N-Way) are accessed.

STAIRSTEP METHOD

The stairstep method involves persisting those crossings that exist on a set of predefined reports. Typically, this involves crossing all of the levels of dimensions that will exist on a single cross tabular drilldown report, and then repeating this process for each subsequent predefined report.

For example, let's consider a report where the Product dimension (consisting of Category, Item and SKU) is crossed with the Time dimension (consisting of Year, Qtr and Month). In order to make sure that there is an exact match subtable for each possible combination of levels of these dimensions, first start with the Cartesian product of all levels of both dimensions. Then, "stairstep" down each dimension by dropping off levels in sequence. The result would look like the following:

```
Category Item SKU Year Qtr Month
Category Item SKU Year Qtr
Category Item SKU Year
Category Item     Year Qtr Month
Category Item     Year Qtr
Category Item     Year
Category          Year Qtr Month
Category          Year Qtr
Category          Year
```

If the report contains more than two dimensions, the process is simply extended to include the additional dimension. Also, if the reports need the capability to be subset on certain dimension levels not contained in the axes dimensions, those dimension levels would need to be added to each crossing. This is a very important point in the stairstep model, because if the subsetting dimension level is not represented in any subtables, the N-Way will be used to satisfy all requests that include the subset.

Note that this technique is optimized only for drill downs operations (ie. subsetting the data by the selected parent member when moving to the next level). It is not fully optimized for down operations, which is when the report jumps to the next level without subsetting on any member of the parent. To fully optimize for this type of reporting, it would be necessary to include every possible combination of levels for the dimensions used in the report.

As you can see, the stairstep method is very effective if the types of reports can be predetermined. However, one of the key benefits of OLAP is that it allows the user to "slice and dice" the data any way they want. The next technique will optimize a model for more ad-hoc reporting.

SPIRAL METHOD

Although the Stairstep technique is very effective for determining which subtables to persist in a controlled reporting environment, the real power of OLAP databases is the ability for the user to create ad hoc reports, or slice and dice through the data. A very common use of OLAP tools is for a user to identify an anomaly in some business measure, and then slice this anomaly across various combinations of dimensions to try and determine why the anomaly exists.

To optimize for these types of reports, the modeling effort must be approached a little differently. Since ad hoc queries often combine a limited number of levels from many different dimensions (as opposed to many levels of a limited number of dimensions in the Stairstep example), it is beneficial to persist subtables that contain levels from many of the dimensions in the model. To keep the size of the overall database manageable, it is also desired to persist high cardinality data in as few subtables as possible.

The Spiral technique provides a model for this ad hoc environment. To get subtables that contain levels from as many dimensions as possible, and also limit the number of high

cardinality dimension levels in those subtables, the dimension levels need to be ordered based on their dimension and cardinality.

Figure 1 illustrates how this can be accomplished. Notice that each dimension has been placed on a set of vectors originating from a common center point. Next, the levels for each dimension are placed on the appropriate vector, with the highest level of summary furthest from the center, and the lowest level of summary closest to the center. It is also helpful to list the cardinality of the level next to it on the vector. Typically, cardinality will increase when moving from high summary to low summary levels.

Once this is done, the levels can be ordered by starting at the highest level of the dimension that will be most commonly used in the application, and then draw a line to the highest summary level of the next most common dimension. Continue this until the highest level of all dimensions is connected. Then, move to the next level and repeat the process. Notice that the diagram begins to look like a spiral leading to the central intersection of the dimension vectors. As the line approaches the center, some dimensions will run out of levels before others. Once this happens, use the cardinality of the levels to determine where to go next, going from lowest cardinality to highest.

When complete, the diagram should look like Figure1.

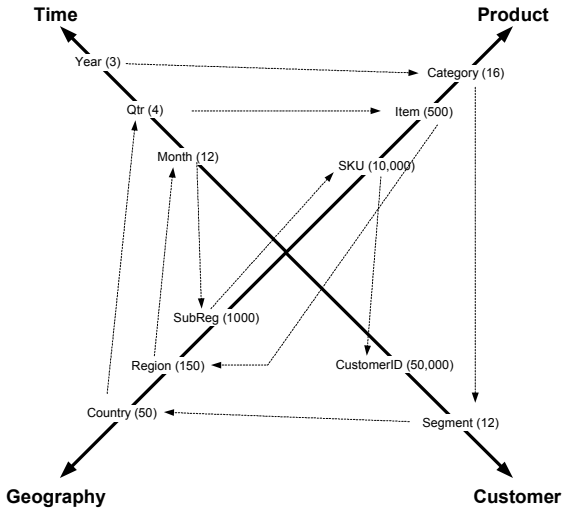


FIGURE 1 – SPIRAL OPTIMIZATION TECHNIQUE

By following the lines, a prioritized list of dimension levels can be determined. For our example, it would look like:

```
Year Category Segment Country Qtr Item Region Month SubReg SKU CustomerID
```

This simply produces an N-Way, but if we then stairstep this subtable down using the priority order, we should obtain a reasonably optimized set of subtables. The resultant model would look like:

```
Year Category Segment Country Qtr Item Region Month SubReg SKU CustomerID
Year Category Segment Country Qtr Item Region Month SubReg SKU
Year Category Segment Country Qtr Item Region Month SubReg
Year Category Segment Country Qtr Item Region Month
Year Category Segment Country Qtr Item Region
Year Category Segment Country Qtr Item
Year Category Segment Country Qtr
Year Category Segment Country
Year Category Segment
Year Category
Year
```

Since the highest cardinality dimension levels only occur in a very few subtables, the size of the resultant database is controlled. Also, every subtable contains levels from as many dimensions as possible. Although many queries will not be answered by an exact subtable with this model, most queries should be able to be satisfied without going all the way back to the N-Way.

OTHER TECHNIQUES

The Stairstep and Spiral techniques are just a sampling of strategies that can be used to optimize the persistent subtables that are stored in an OLAP model. These techniques can be used on their own, combined with each other, or combined with other techniques that will allow the OLAP model to ultimately serve the performance requirements of the user.

These techniques are used as a starting point when designing the model. Once the model is deployed, it will need to be monitored, and then adjusted based upon how users are actually exploiting it. How to monitor the model is discussed in more detail in the section on Deploying the Model.

OPTIMIZING THE MODEL: PARTITIONING STRATEGIES

Once it has been determined which subtables are to be persisted in the OLAP database, the next step is to decide how that data will be physically stored. The SAS HOLAP architecture provides an extremely flexible and scalable way to physically implement the model.

Due to the simplicity and performance benefits of MOLAP on data that has small to medium cardinality, a single MOLAP structure, or MDDB, should be used as a starting point for any model. Only when performance issues, data size issues or other special circumstances arise should one consider partitioning the model into separate physical structures.

There are two ways that an OLAP model can be partitioned. Stacking refers to storing different subtables in separate physical structures. Racking refers to taking a single subtable, or set of subtables, and partitioning them based on the value of one or more dimension levels. Each of these techniques, and when they might be appropriate, is discussed in the following sections.

STACKING

One way to partition an OLAP database is by stacking subtables. In stacking, each persistent subtable can be stored in its own physical structure. The structure can be any data format that SAS can read, but typically it is SAS MDDB's, summary datasets, or RDBMS tables.

Stacking is typically used to solve the following issues:

- High cardinality data that does not scale well in an MDDB.
- Large subtables that do not fit in memory.
- Special aggregation rules.
- Summary data already exists in a non-MDDB format.

High Cardinality

As mentioned in the introduction, high cardinality dimension levels often do not scale well in MOLAP structures. To get around this, stacking can be used to store those persistent subtables that contain high cardinality dimension level is a more appropriate structure. What defines "high" cardinality is a relative term, and will depend on the hardware platform, the total number

of dimensions and levels in the model, and the number of persistent subtables, among other things. However, dimension levels that are approaching 1000 members should be monitored for performance issues.

Using the example illustrated in the Stairstep technique, let's assume that the dimension levels have the following cardinality: Year(3), Quarter(4), Month(12), Category(16), Item(500), SKU(10,000). Obviously SKU is the dimension level that will be of the greatest concern. To stack this dimension level, we would take every persistent subtable that contains this level (including the N-Way) and break those subtables out of the MDDB. In this case we would remove the following subtables from the MDDB and stack them on top of it:

```
Category Item SKU Year Qtr Month
Category Item SKU Year Qtr
Category Item SKU Year
```

These subtables could then be stored in a RDBMS or SAS dataset, which are typically more suited to retrieving small subsets of values from a large dimension level. When stacking subtables for high cardinality dimension levels, it is imperative to index the tables on the dimension level variable that caused this subtable to be stored separately – in this case, SKU.

Memory Issues

Although medium to high cardinality dimensions do not always create enough of a performance problem to warrant stacking them, addressable system memory often will become an issue. SAS requires each subtable of an MDDB to be able to fit into memory to be exploited. On 32 bit systems, this limit is 2GB; on 64 bit systems, the limit is typically controlled by the amount of RAM allocated to the SAS session. By partitioning subtables that exceed these limitations into other structures, the limitations can be extended. Typically high cardinality dimension levels will not be accessed without first being subset to a reasonable number that can be presented on a report. By letting the RDBMS subset this data before it is summarized, the amount of data passing through the OLAP engine is reduced.

Special Aggregation Rules

In the introduction, one of the benefits mentioned for MOLAP architectures was the fact that the aggregation rules and logic were built into the OLAP engine. Most data can be summarized across any dimension using simple aggregation rules that are common to all measures across all dimensions. There are however, rare instances where certain business measures do not conform to these rules. For example, account balances are not additive across the levels of the time dimension, since (unfortunately!) the balance for the year is not equal to the sum of the balances of the individual days, months or quarters. These are often called "non-additive" measures, and their special aggregation rules can normally be handled more flexibly in a two dimensional structure.

Stacking allows us to break out all subtables that contain the non-additive dimension, and manually aggregate those subtables using the appropriate business rules in SAS Data Step or SQL code. When these types of dimensions exist, it is imperative to persist **all** possible subtables that can include any levels of the dimensions that are not additive. Otherwise, the OLAP engine may attempt to perform some aggregations at run time using the predefined aggregation rules of the OLAP engine.

Obviously, this can potentially lead to a very large number of persistent subtables, so a star schema model is often an appropriate storage mechanism.

Existing Summarized Data

Another application for stacking is to include existing summarized data without replicating it in the OLAP database. This data might be in an existing data warehouse or some other source. Instead of replicating the already summarized data, it can simply be combined with other summarized data that is not in the warehouse, and "stacked" into the model. SAS/Access® software makes it very easy to include existing RDBMS based star and snowflake schemas into an OLAP data model.

As you can see, stacking can provide the OLAP model with much needed scalability and flexibility, adding complexity to the model only when needed. One further benefit of stacking is that each stacked data source can be partitioned onto separate server platforms to further increase the scalability. In addition, SAS/Access® software will pass much of the subsetting work through to the RDBMS, which limits the amount of data that moves across the network.

RACKING

Whereas stacking involves partitioning multiple subtables into multiple physical structures, racking involves partitioning a single subtable into multiple physical structures. The Time dimension is often used in racking models, whereby the separate structures are built for each time period. For example, a separate MDDB could be built for each month. When using racking, the OLAP engine only accesses the physical structures needed to satisfy any given query. Thus if the model contains monthly MDDB's and a report needs just one month, only one MDDB would be accessed. If a report requested multiple months, then multiple MDDB's would be accessed and joined together.

One real benefit of this approach is at build time. If data is refreshed on a monthly basis, then instead of rebuilding the entire OLAP database, only the month MDDB affected by the new data will be rebuilt.

Racking can also be used on other dimensions when reports are typically subset on that dimension. An example might be a marketing analysis application where each marketing manager has a set of products they are responsible for. Each manager could thus have their own MDDB containing only the data for their products. The advantage to this approach is that by subsetting the high cardinality (Product) dimension, the subsets should be small enough for MOLAP implementation. If a consolidated view is needed, the HOLAP racking capabilities can join the disparate MDDB's together in a report when needed.

In summary, racking and stacking provide the OLAP model with scalability and flexibility. It allows the majority of the model to be implemented in a single, manageable MDDB, while only breaking out the pieces necessary to address scalability or flexibility issues. In addition, stacking and racking can be combined in the same model, as needed, to accomplish some of the benefits of both techniques. This is referred to as "stracking."

SKELETON MDDB

In order to hide the complexities of the underlying model from end users, the SAS HOLAP implementation uses the concept of a skeleton (or proxy) MDDB. This skeleton contains all of the attribute information of the entire OLAP model, but no data. Instead, it stores metadata about the distributed data model that has been defined. The key benefit of this is that changes made

to the underlying physical structure of the model can be transparent to the end user. Thus, it is prudent to consider using the HOLAP skeleton, even if the initial data model is not partitioned. This will allow a more seamless conversion to a partitioned model later if one is needed.

DEPLOYING THE MODEL

Once the model has been designed, and the physical layer has been determined, the final step is to actually build it and deploy it to users. The focus of this document is really on the modeling issues, however for completeness, I'll address some deployment topics at a high level.

SECURITY

As with any data, sensitivity is often an issue with OLAP databases. The SAS OLAP Server provides a role based access control subsystem that facilitates the following types of security:

Cube Level – Which cubes
Application Level – Which reports
Hierarchy Level – Which levels
Column Level – Which measures
Row Level – Which members

Security issues should be addressed during the planning and design phase, and implemented through the access control subsystem accordingly.

In addition to security, the access control subsystem can be used as a filtering tool to facilitate easier report development. For example, if product managers each need a report for only their products, the row level security can be set for each product manager so that they can only view data for their products. Thus, the report developer can build a single report for all product managers, and the access control subsystem will control what data is surfaced to each user.

More information on the Access Control subsystem can be found in the SAS/OLAP Server Administrator's Guide.

BUILDING THE MODEL

There are several ways to build the model once it is designed. In addition to PROC MDDB, both SAS/EIS® Software and SAS Enterprise Guide® Software provide GUI interfaces for building MDDB's. However, SAS/Warehouse Administrator® Software is the preferred tool. It encompasses the entire ETL process, not just the creation of the OLAP model, and provides facilities for scheduling refresh jobs and documenting the process via metadata. In addition, it has built-in support for many of the optimization techniques discussed in this paper. Further information on building OLAP models using SAS/Warehouse Administrator can be found in the SAS/Warehouse Administrator Reference Guide.

If using PROC MDDB, remember to order the HIER statements for building subtables from those with the greatest number of dimension levels to those with the fewest. This will optimize the build time performance of the MDDB because each subsequent table can be built from a previous summary table instead of always going back to the N-Way or detail data.

Also, if the model has been partitioned using racking or stacking, it is possible to build the various physical structures in parallel. This is especially relevant in a multi platform environment, and can significantly reduce the total amount of time required to rebuild or refresh the database.

END USER REPORTING

Once the database is built, a reporting environment needs to be established to facilitate exploitation of the database. SAS provides a variety of OLAP client interfaces that are appropriate to different types of end users. SAS/EIS® software provides a full client interface that includes a variety of tabular and graphical objects. In addition, having the full power of SAS on the client provides additional analytic capabilities that are outside the scope of most OLAP applications.

The web can also be a very effective delivery mechanism for OLAP reports. SAS offers OLAP client interfaces based on several platform independent technologies such as HTML, Java Applets and Java Server Pages. The web based OLAP viewers are also highly integrated into the SAS Business Information Portal, allowing user to subscribe to content contained in OLAP databases.

Finally, SAS supports the OLE DB for OLAP standard as both a provider and a consumer. Thus, SAS OLAP databases can be surfaced via OLE DB for OLAP compliant interfaces such Microsoft Excel™ or SAS Enterprise Guide® software.

ONGOING OPTIMIZATION

Once the model has been designed and deployed to end users it is imperative to revisit the design on a regular basis to ensure that the assumptions made during the design phase are still valid, and that nothing substantial was overlooked.

The HOLAP data provider generates a log file that contains information about all requests for data. The types of information contained in the log are the time it took to satisfy the query, the name of the data source(s) and subtable(s) used, the number of items returned, etc. By default, this file is stored as a SAS dataset in the WORK library for each SAS user session. It is, however, often advantageous to store the log files in a more permanent location. This can be accomplished by setting an attribute on the data provider class.

Because the log file displays subtable names, it is a good idea to provide a descriptive name for each subtable that is persisted in the model. Otherwise the log will display the subtable names as a number that was system generated when the subtable was created. A good naming convention is to use the names of the dimension levels that are included in that subtable.

CONCLUSION

The HOLAP capabilities contained in SAS/OLAP Server® software provide a robust level of scalability and flexibility. However, to fully leverage the power of HOLAP, a data model must be carefully planned and executed that takes into account both user reporting requirements and the characteristics of the data to be reported. By carefully understanding both the requirements and the data, highly scalable OLAP applications can be delivered to help end users solve sophisticated and complex business problems by leveraging vast amounts of data, which are aggregated and presented in an easily consumable format.

REFERENCES

Weinberger, Ann and Mattias Ender. *The Power of Hybrid OLAP in a Multidimensional World*. SUGI 25, Paper 133-25.

Wright, Ken. *New Features in Warehouse Administrator*, SUGI 25, Paper 114-25.

Moorman, Mark. *The Art of Designing HOLAP Databases*. SUGI 24, Paper 139.

SAS/EIS® Technical Report: HOLAP Extensions, Release 6.12. SAS Publication 56564.

SAS/OLAP Server Administrators Guide, Release 8.1. SAS Publication 57924. SAS Institute Inc, Cary, NC

SAS/Warehouse Administrator Users Guide, Version 2.0, First Edition. SAS Publication 56799. SAS Institute Inc, Cary, NC

Kimball, Ralph. 1996. The Data Warehouse Toolkit: Practical Techniques for Building Dimensional Data Warehouses. (John Wiley and Sons, 1996).

ACKNOWLEDGMENTS

I'd like to thank Rob Stephens for providing the opportunity to collect many of my random ideas into this document. Also, Mark Moorman, Stu Levine and Ben Zenick for invaluable content and review assistance. Finally, thanks to the entire OLAP development team at SAS Institute for putting together the most flexible and robust OLAP technology possible.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Greg Henderson
SAS Insitute Inc.
SAS Campus Dr
Cary, NC 27513
Ph: (919) 531-2152
Fax: (919) 677-4444
Email: greg.Henderson@sas.com
Web: http://www.sas.com

Paper P502

Use of SAS/AF V8e to Compare Death Certificate Data with Health Survey Data from the National Center for Health Statistics

Gretchen K. Jones, NOVA Research Company, Hyattsville, MD
Sandra T. Rothwell and Christine S. Cox, National Center for Health Statistics, Hyattsville, MD

ABSTRACT

Determination of Fact of Death is one of the follow-up activities which the National Center for Health Statistics (NCHS) conducts for health surveys. The National Death Index (NDI) is a data base containing information on all U.S. deaths since 1979. To validate a death, the NCHS orders copies of Death Certificates to compare against a more robust set of data than is available from the NDI.

The authors have developed a SAS/AF (V8e) application, using the Form Viewer attached to a SAS Dataset Model, which provides a tool to compare NCHS survey data with actual Death Certificates. Access is provided (through push button control) to frames containing data from three different data files. The user compares and "scores" the extent of match of the data variables on a frame with six pages. The final frame gives a summary of scores from the previous frame, with a box for final evaluation of whether the Death Certificate matches the survey data on the files. The presentation will include a demo of the application.

This paper is appropriate for persons with intermediate SAS skills.

INTRODUCTION

This paper describes a SAS/AF V8e application of the Form Viewer attached to a SAS Dataset Model. The application involves being able to view the National Health and Nutrition Examination Survey (NHANES) data from subjects now believed to be deceased. The purpose is to allow an evaluator to compare variables on the frames with information recorded on a death certificate thought to be a possible match. The evaluator will score and comment using the frames, thus updating the underlying SAS file.

Background is given describing the data and how the previous system worked; the frame-building process is explained (with coding); screen prints of sample frames are presented; and improvements of the application over the previous system are listed.

BACKGROUND ON FOLLOW-UP AND LINKAGE

One of the most useful tools for the epidemiologist is the longitudinal or follow-up study. These studies provide baseline measures that can be compared to subsequent health outcomes to determine risk factors for some of the more common chronic diseases as well as to provide some information on the natural course of these diseases and conditions.

A major health outcome is the fact of death, the underlying cause of death (or the condition that led to the eventual death), and any other contributing diseases or conditions (such as previous heart attacks, etc.)

THE NATIONAL DEATH INDEX (NDI) AS A PROVIDER OF DATA RECORDED ON DEATH CERTIFICATES

The National Center for Health Statistics (NCHS) provides a data base, the National Death Index (NDI), for use under fairly strict guidelines, that can provide data on fact and cause of death. The NDI user provides a limited set of variables and the NDI software provides all potential matches to the person described by this set of variables. State of death and certificate number are provided on each potential match.

It is then the job of the user to determine which matches could be the death certificate for a subject in an epidemiologic study. The NDI does not provide any more confidential information than is necessary for the user to make some kind of informed decision. For example, if the user provides the subject's Social Security Number (SSN), the NDI provides only information that specifies which digits of the SSN match. The SSN itself is not provided. Another issue is the fact that birth dates can often be mis-reported in the survey and/or on the death certificate. Often there is a close match on SSN and a birth date and the name is a commonly used one. This causes any match to be tentative. Both probabilistic and deterministic methods are used to winnow out bad matches or to find the very good or perfect matches. There is always a large number which are neither obviously matches nor non-matches. These cases are resolved by ordering copies of the death certificate and reviewing all socio-demographic variables present on the certificate and collected during the survey participant interview. This frame application was developed to facilitate this review process.

REASONS FOR ORDERING DEATH CERTIFICATES

Both the survey and the death certificate itself contain more information than is available from the NDI match. For example, whether the subject ever served in the armed forces and the subject's major occupations or industries in which he or she worked can usually be found on the death certificate. The name of the informant on the certificate is often someone who lived in the house with the subject when they were interviewed either at baseline or at some later follow-up contact. Finding m

matches to this information on certificates can often confirm or deny the match. Ordering the certificates can sometimes discover the death of a twin, where the last names are the same, the first names sound alike, the birth dates match perfectly and the SSNs are very similar. If the twin was not in the study, an incorrect death can be eliminated.

STATEMENT OF THE TASK TO BE PROGRAMMED IN SAS/AF

Data from several SAS source files (a Subject File, a Roster File, and an Address File) are to be displayed on screens so that an evaluator, usually female, (hence feminine pronouns are used in reference to her throughout) may manually compare the information with corresponding data on the death certificate which has been ordered from the State in which the person presumably died.

For every variable on the Subject File looked at, there is a place for the evaluator to enter a "Match Code," which can be an "E" for exact match, a "P" for partial match, or an "N" for no match. Next to the Match Code combo box, there is a text entry field where the evaluator must enter the data from the death certificate if she has selected a "P" or an "N" from the Match Code combo box. And the evaluator is not allowed to enter anything in the text entry field if she has selected an "E" from the Match Code combo box. The frame is programmed so that any box which must have data entered into it has a background color of yellow.

From comparing the address on the death certificate with data from the Address File, the evaluator is able to assign a Code of "N," for no address match, an "S" for State only match , a "C" for City and State matches, or an "E" for exact match with the Household Address, or with the Mailing Address. And lastly, comparing the name of the informant listed on the death certificate with data on the Roster File, the evaluator assigns an "N," a "P," or an "E" again for no match, partial match or exact match on informant in household, or an other contact.

Privacy and security considerations must be programmed into the system. Only someone who takes the privacy oath and has an authorized USERID will be allowed to bring up the system. The evaluator must be able to navigate between screens as much as possible, and when finished with match coding, will go to a decision screen, where she will be able to look at frequencies of match codes she has assigned on previous screens, and to come to a decision as to whether the person described on these files is the one on the death certificate in front of her. She then must have the ability of assigning a "decision" of "N" (not a match), "M" (a match), or "R" (needs to be reviewed by client).

HOW OLD SYSTEM WORKED

The previous system was written in SAS/AF version 6. It was set up to handle one (and only one) survey. It was strictly "fill in the blank", no push buttons, no objects with inheritance. And navigation was difficult. The system provided no summary statistics such as what proportion of the potential certificates had been reviewed or what proportion of review certificates were considered "true".

THE NEW SYSTEM - HOW IT IS DESIGNED

To implement the new system, it was necessary to use SAS/AF Version 8e, because we needed to make use of the form viewer, which doesn't exist in Version 8. The system consists of five frames or screens, one containing six pages.

The first screen has the Title ("Death Certificate Verification System") and the confidentiality statement. There is a push button for the evaluator to click on if she accepts the statement, and the second screen opens. The evaluator is asked to choose the survey being reviewed from a drop-down box, and to enter her USERID. If the evaluator is found to be authorized, the third screen opens, where the she enters the CASEID of the subject for the appropriate survey.

Screen four consists of six pages, which can be navigated using the "page turns" in the upper right or left-hand corners, or the push buttons at the bottom of the frame. Pages 1-3 contain data to be reviewed and updated from the Subject File, Page 4, from the Address File, and Page 5 from the Roster File. Page 6 is the Comment Screen, where the evaluator is given a place to provide additional input if needed.

When the evaluator is ready to make a decision as to whether the death certificate is a match, she navigates to the fifth frame, where statistics as to her previous scoring on individual items are displayed in text control boxes, and she will enter her decision code.

Figure 1 below is a sample of the first page of the Subject Screen. The evaluator will choose E, P, or N from the combo-boxes in the middle column for each variable in the first column. Then if E is not chosen, the evaluator is forced to enter (in the last column) what was written on the death certificate for that variable.

Figure 2 below is a sample Decision Screen. Frequencies of the codes that the evaluator entered on the Subject Screens are shown, as well as the Match codes which she entered on the Roster Screen. After careful consideration, she chooses Match, False, or Review as a Final Decision.

HOW THE APPLICATION IS CODED

Much of the programming of the application could be done automatically by merely dragging and dropping objects onto a frame, and setting attributes through the properties window. However, a considerable amount still had to be hard-coded, using SAS Component Language (SCL). The SCL attached to the frame is called the Frame SCL. When one is using a Viewer attached to a Model, one also uses Model SCL, which can be accessed through right-clicking the mouse on the Viewer. Many of the functions of objects on the Viewer were coded through Model SCL.

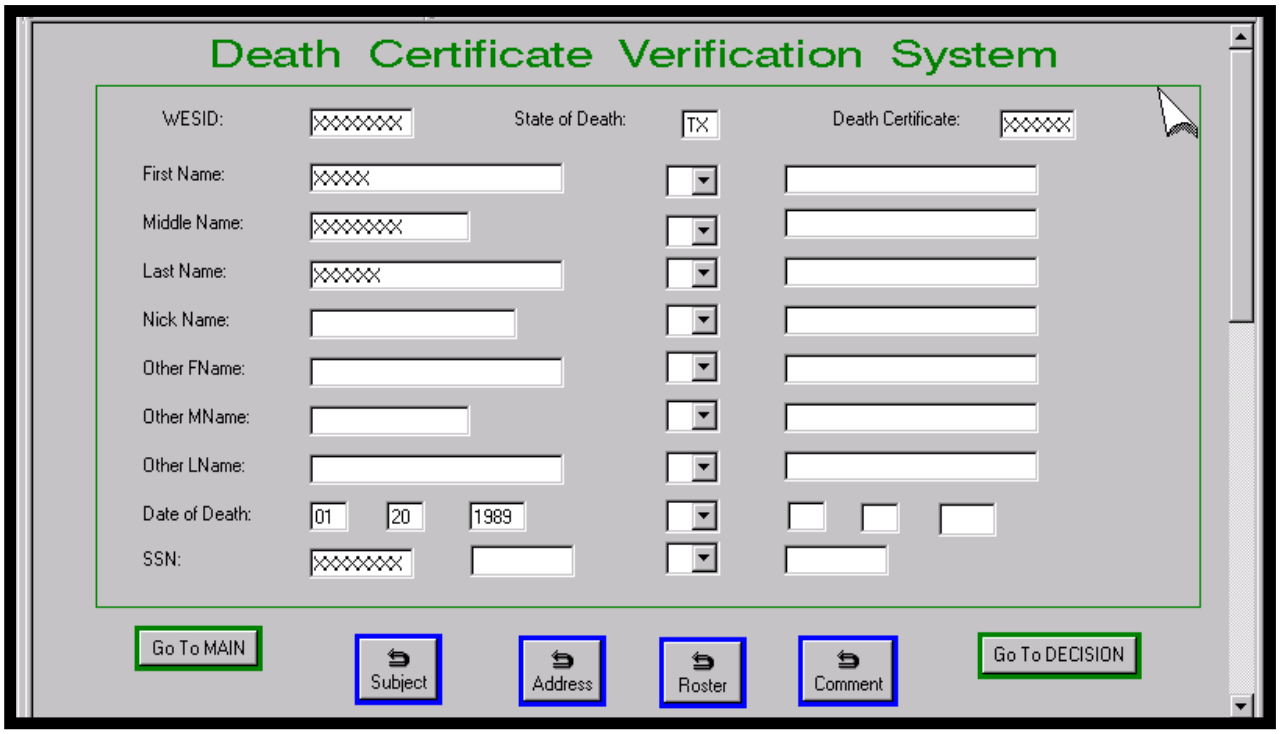


Figure 1. Subject Screen (first page)

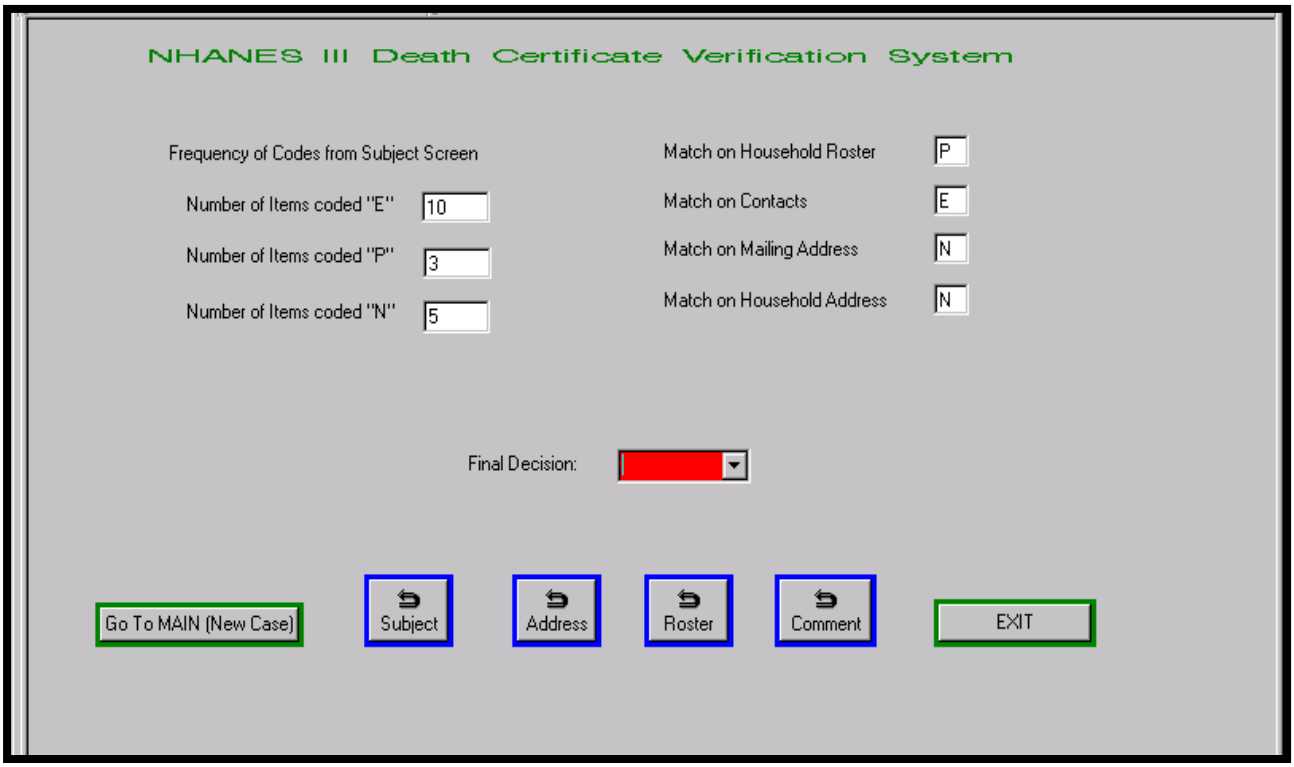


Figure 2. Decision Screen

Following is an example of the Frame SCL for the Subject Screen (see Figure 1).

```
entry SURV: num SURVEY Q1-Q22
IDPASS: char;
dcl num rc;
dcl list msgList=
{'Invalid WESID.  Return to MAIN.'};

INIT:

SASDataSet1.table='DCert.Subjct5';

SASDataSet1.where='WESID =' ||
   quote(IDPASS);

rc = sysrc();
if rc ne 0 then do;
choice=
MessageBox(msgList,'I','O','Error');
if choice = 'OK' then Call Display
   ('Screen3.Frame',survey,
     surv);
   end;
return;
Subroutine:
SASDataSet1._GetColumnText('MA',Q1);
SASDataSet1._GetColumnText('MB',Q2);
SASDataSet1._GetColumnText('MC',Q3);
   .
   .
   .
SASDataSet1._GetColumnText('MU',Q22)
;
return;

PBSubject:

   Formviewer1._gotoPage(1);
   return;

PBAddress:

  Formviewer1._gotoPage(4);
   return;

PBRoster:

  Formviewer1._gotoPage(5);
   return;

PBComment:

  Formviewer1._gotoPage(6);
   return;



PBDecision:

Link Subroutine;
  Call display
('SCREEN8.FRAME',SURV,SURVEY,Q1,Q2,
Q3,Q4,Q5,Q6,Q7,Q8,09,
Q10,Q11,Q12,Q13,Q14,Q15,Q16,Q17,Q18,
Q19,Q20,Q21,Q22,IDPASS);
return;

PBMain:
  Call Display ('SCREEN3.FRAME',
SURVEY, SURV);
return;
```

SURV, SURVEY, and IDPASS are SCL variables that are passed from Screen3. In the INIT section, the Model SAS Data Set is opened, and an observation is searched for where the variable WESID = IDPASS, the ID which the evaluator entered at Screen3. If a match is not found, an error message is displayed and control is returned to Screen3. Otherwise control stays with Screen4. The Subroutine puts the values of the "match codes" that the evaluator has entered into the SCL variables Q1-Q22. The coding for the Subject, Address, Roster, and Comment push buttons merely takes one to the appropriate page on the Form Viewer. The push button for the Decision Screen passes all the previous SCL variables, plus the ones assigned in Subroutine, and transfers control to the Decision Screen (Screen8). The push button for Main merely transfers control to the Main Screen (Screen3).

IMPROVEMENTS UNDER 8E

Under the new system, any number of surveys may have potential death certificate matches loaded. A judicious use of inheritance will ease the programming job for this. We can put more easily understood and used navigation objects on the screens and we can provide better security. We can provide better editing of the data entry fields to be sure all available information is being used in the judgement. We can be sure that there is a decision on each potential match for a given survey before the work is turned back to the linkage staff. We can provide some summary statistics available from a menu or push-button. And, finally, the system will be much more pleasing to the eyes.

CONCLUSION

The National Center for Health Statistics has a need for a tool to enable an evaluator to view and score data from national survey files to compare against Death Certificates which have been ordered from the various states. The purpose is to determine fact of death for particular subjects who have been examined in the NHANES surveys. The Form Viewer with SAS Data Set Model implemented in SAS V8e, which provides such a tool, has been described in this paper.

The application also provides an electronic record of the entire death certificate evaluation process for a given NDI match.

REFERENCES

SAS Institute Inc. *SAS/AF Software: Application Development I Course Notes.* Cary, NC: SAS Institute Inc., 2000.

SAS Institute Inc. *SAS/AF Software: Application Development Ii Course Notes.* Cary, NC: SAS Institute Inc., 2000.

SAS Institute Inc. *SAS/AF Software: Changes and Enhancements in Version 8 Course Notes.* Cary, NC: SAS Institute Inc., 2000.

ACKNOWLEDGMENTS

The authors wish to thank Cay Loria, formerly of NCHS, who originated the idea of using SAS/AF as a tool in matching survey data with information on the death certificate and Ann Rockett of SAS Institute, whose technical assistance has been invaluable in developing the application.

CONTACT INFORMATION

Gretchen K. Jones
6525 Belcrest Road, Room 730
Hyattsville, MD 20782

Phone: 301/458-4301
Fax: 301/458-4038
Email: gkjones@cdc.gov

Paper P503

Creating Visit Specific CRF Checklists for a Longitudinal Study using a SAS/AF® Application

Authors: Emily A. Mixon; Karen B. Fowler, University of Alabama at Birmingham

ABSTRACT

A longitudinal study recently opened at our Pediatric AIDS Clinical Trials Unit (PACTU) requiring complex study visits, a large number of case report forms (CRFs) and an increased frequency of study visits. To assist both clinical and data management staff in managing and maintaining a visit checklist of CRFs required, we modified a previous SAS/AF® application. Five sites in Alabama, Georgia, and Florida send CRFs for data entry to UAB and we have created a Data Management Application (SAS v.8) that generates the expected visits for new patients randomized and also generates the data for the visit specific CRF checklists for each participant's subsequent study visits. The visit specific CRF checklist includes the Patient Id Number, expected visit date range for the next study visit, the study calculated year, month and visit week of the upcoming visit and a list of all the CRFs required for the visit per protocol. Using the data obtained in the randomization section of the application we are able to calculate the differing follow-up schedules based on the age and infection status of the patients. This information is then exported to WordPerfect where a macro generates the visit specific CRF checklists for the participants.

INTRODUCTION

The data management team for the Southeastern Pediatric AIDS Clinical Trial Unit (PACTU), generates monthly reports of expected study visits, and also an ongoing list of delinquent case report forms (CRFs) not in the national database. With the recent addition of a longitudinal study (protocol 219C), the number of study visits and number of CRFs required has significantly increased. In order to assist the clinical and the data management staff, an older version of the PACTU Data Management Application has been updated to generate visit specific CRF checklists for each participant's study visits. The PACTU Data Management Application has been previously discussed in the SESUG 99 paper entitled "Using SAS/AF® To Create Applications for the Administrative Aspects of Data Management in Clinical Trials" (Mixon, et al. 1999).

We have created a SAS/AF® FRAME Application in Version 8.0 of the SAS System on the Windows 2000 platform. Upon entering the PACTU Data Management Application, the user enters the main menu for PACTU Randomizations and Visit Schedules and is presented with four choices of how to proceed (Fig. 1). This paper will address each of these options (New Randomizations, Search and Edit, Generate Visit Reports, and 219C CRF Data Set) in the following respective sections.

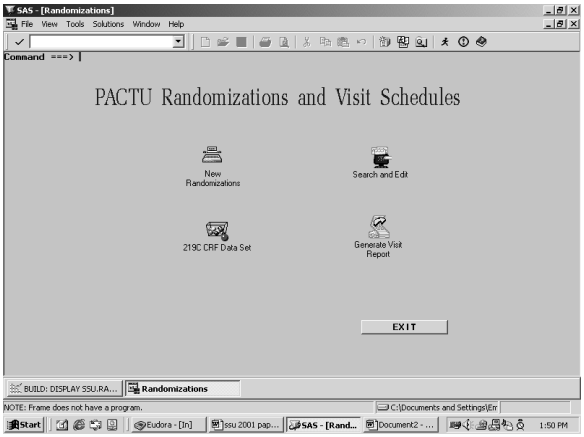


Figure 1

NEW RANDOMIZATIONS

As described in the SESUG 1999 paper, the purpose of the New Randomization frame is to enter the information needed for generating the expected visits report (Fig. 2). We have updated the frame using new SAS/AF® Version 8 features. The randomization frame has a Data Form with Text Entry Control components for entering the data and Text Label Control components for labeling the different fields. Both the heading of the frame using a Graphic Text Control and the Command Push Buttons at the bottom of the frame are objects from SAS/AF® version 6.12.

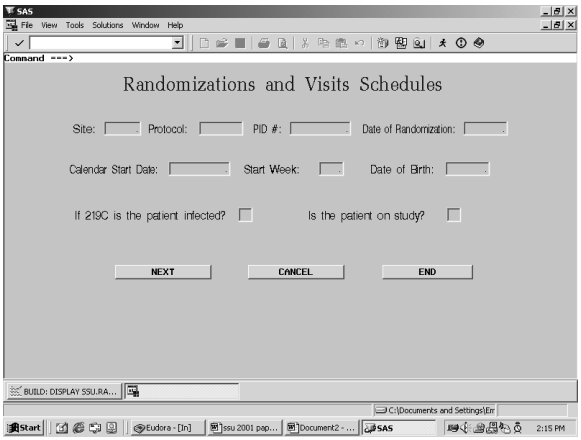


Figure 2

The SCL for the New Randomization frame calls the Data Form allowing a new row to be added when needed. The SCL behind the Data Form sets all the fields to missing, moves the cursor through the fields, and calculates a visit schedule based on the data entered by the user.

SEARCH AND EDIT RANDOMIZATION ENTRIES

The Search and Edit Randomization Entries frame allows the user to sort the randomization data set by either site number (from five study sites in Alabama, Georgia, and Florida) or Patient Identification (PID) number (Fig. 3). The Search and Edit frame is similar to the one described in the SESUG 99 paper except for using Version 8 controls. The Graphic Text Controls for the site numbers subsets the Data Table by the user selected site. Entering a PID number in the Text Entry Control and clicking the "Sort Table" Push Button Control subsets the Data Table for only those entries with the specific PID number. Selecting the row in the Data Table pulls up the New Randomization frame containing the Data Form with the selected observation allowing the user to edit the record (Fig. 4).

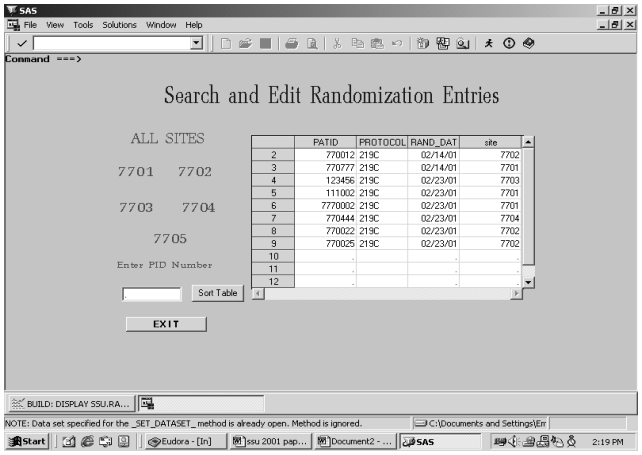


Figure 3

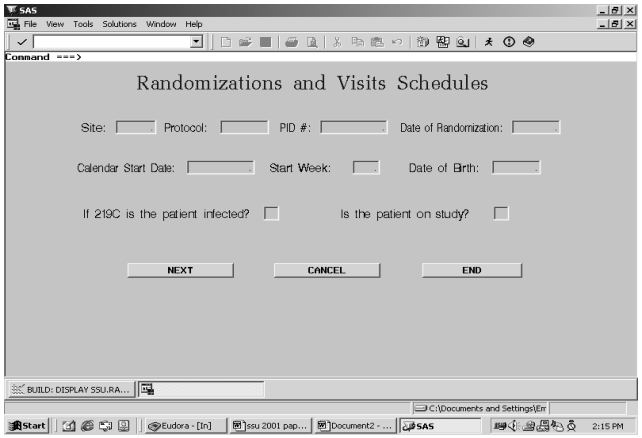


Figure 4

The SCL for the Search and Edit frame begins by declaring an object called ID. The data set is opened by the ID._SETDATASET statement and lists are created using the MAKELIST method. The Graphic Text Controls are coded to change color and to subset the Data Table using the _SET_WHERE_ method. The Data Table is also sorted when a PID number is entered into the Text Entry Control and the Push Button Control is selected. Once the Push Button is pressed, a PUT statement with a CALL PUTLIST routine is used to check the value entered into the Text Entry Control and to verify that the value from the MAKELIST is correct. These events are followed by the Data Table being subset using the ID._SETWHERE method. To view a single observation displayed on the Data Form, the SCL for the Search and Edit frame uses the _SET_INSTANCE_METHOD_ and the _SELECT_ methods as described in the SESUG 99 paper. When the user exits the frame, the data set is closed and the lists are deleted.

GENERATE MONTHLY EXPECTED VISITS

The Monthly Expected Visits frame generates a report of expected visits by site and time interval specified by the user (Fig. 5). The site number is selected from a list in a Radio Box and the date range is entered into Input Fields. Once the user has selected the values for the fields, the "Print" Icon is pressed generating the expected visits report.

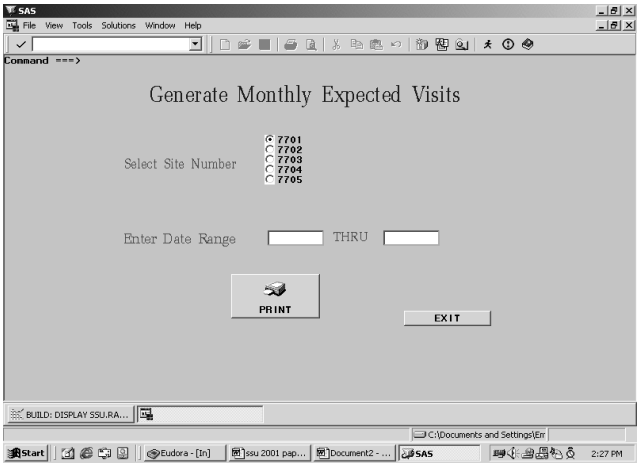


Figure 5

The SCL for the Monthly Expected Visits frame begins when the "Print" Icon is pressed. The data are subset based on the site and date range specified by the user. A macro is then executed to check all of the date variables in the

data set to determine if they fall within the date range specified. If the dates within the specified range are identified, the dataset is subset using an OUTPUT statement and labels are assigned for the variables appearing on the report. Multiple reports may be generated while in this frame with the user exiting from the frame upon completion.

219C CRF CHECKLISTS

The 219C CRF Checklist frame saves the data set that will be imported into WordPerfect® for generating the visit specific CRF checklist for each individual participant (Fig. 6). The frame consists of Graphic Text Controls, an Input field for specifying the PID number needing the checklists, a "Generate Checklist" Icon, and a Command Push Button. The SCL behind the frame is initiated within a SUBMIT CONTINUE routine when the "Generate Checklist" Icon is pressed. The data set is subset by both on the PID number entered by the user and protocol equal to 219C. Visit specific variables (PID number, expected visit date, study calculated year, month and visit week) are kept and saved as a DBF file. The user then exits the application and opens WordPerfect®. A WordPerfect® macro imports the DBF file, creates a WordPerfect® data file and then merges the data file with the 219C CRF Checklist forms (Fig. 7). The 219C Checklist forms may be printed and distributed to the clinical personnel.

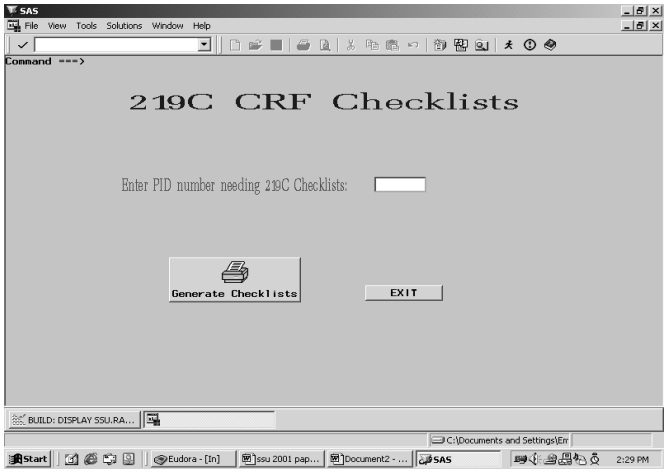


Figure 6

CONCLUSION

Modifications made to a previous SAS/AF data management application enables the generation of CRF checklists for a longitudinal study recently opened at our PACTU sites in Alabama, Georgia, and Florida. These checklists aid the clinical and data management staff in managing the large number of CRFs required at the specific study visits. This application could easily be revised for use in the data management of other research studies.

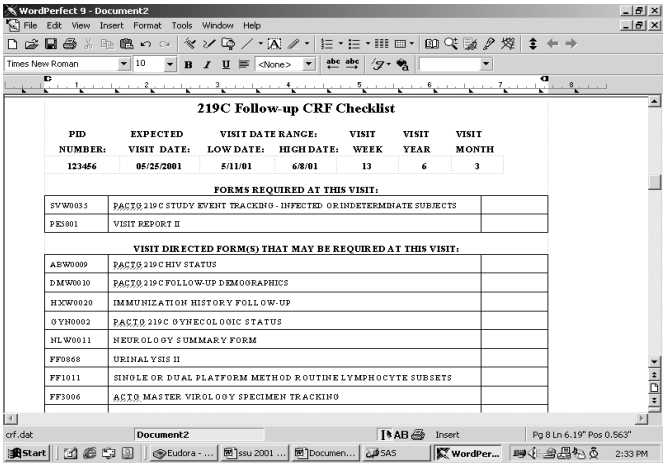


Figure 7

REFERENCES

Mixon, Emily and Fowler, Karen. (1999), "Using SAS/AF to Create Applications for the Administrative Aspects of Data Management in Clinical Trials". Proceedings of the Seventh Annual Southeast SAS Users Group Conference. USA.

Birdeson, Patti M. (1996), "Creating a Contacts Application Using New Classes in SAS/AF Software". Observations, First Quarter 1996.

Carpenter, Ann E. And Leone, Lynn P. (1996), "Introducing a GUI Approach to Data Entry Using Data Table and Data Form Objects in SAS/AF FRAME Software". Observations, First Quarter 1996.

Rocket, Ann Carpenter. (1997), "An Introductory Overview of the Data Form and Data Table Objects in SAS/AF Frame Entries". Proceedings of the Fifth Annual Southeast SAS Users Group Conference. USA.

SAS Institute Inc. (1997), Building SCL Applications using Frame Entries Course Notes, Cary, NC: SAS Institute Inc.

Wilkins, Scott. (1997), "Data Table and Data Form Enhancements in Release 6.12". Observations, First Quarter 1997.

ACKNOWLEDGMENTS

This work was supported in part by the National Institutes of Health, National Institute of Allergy and Infectious Diseases Grant U01 AI41025. The Authors would also like to thank Lynn Leone and others at SAS Technical Support for their helpful advice and suggestions in the development of the SCL in this application.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Emily Mixon
UAB Department of Pediatrics
CHT 752, 1600 7th Ave. South
Birmingham, AL 35233
Work Phone: 205-939-6687
Fax: 205-975-3221
Email: emixon@uab.edu

Supplier Management with SAS® Supply Chain Solutions

Ed Hughes, SAS Institute Inc., Cary, NC

ABSTRACT

This paper presents an overview of the SAS® approach to supplier management, focusing on the use of SAS' well-established optimization expertise to aid in selecting and evaluating suppliers. The supplier management solutions are explored individually and are shown in the context of a complete spectrum of SAS supply chain solutions.

INTRODUCTION

The purpose of this paper is to introduce and describe the SAS approach to supplier management, part of an integrated set of knowledge-based supply chain solutions being developed by the SAS Supply Chain Center. The paper touches briefly on the overall SAS approach to supply chain optimization, and then moves on to describe the business problems solved and the techniques employed by the SAS supplier management solutions.

Briefly stated, managing a diverse group of suppliers involves much more than simply accounting for the goods that they contribute to the supply chain. Equally important are the relative strengths and weaknesses of the suppliers in delivering those goods, along with their contribution toward satisfying business rules controlling the makeup of your supplier portfolio. SAS supplier management solutions add structure, order, and direction to the process of managing and evaluating suppliers, and form an integral part of the SAS solution for Supplier Relationship Management (SRM).

The two supplier management solutions discussed here, Supplier Performance Rating (SPR) and Supplier Portfolio Optimizer (SPO), employ mathematical optimization to aid in evaluating suppliers and restructuring supplier portfolios. Each uses a Web interface that eliminates the need for any in-depth knowledge of optimization but enables users to take full advantage of the insights that it can provide.

SUPPLY CHAIN ISSUES AND SOLUTIONS

The supply chain symbolizes the web of facilities, activities, processes, and relationships that enable the flow of goods from raw materials and essential components to finished products. Suppliers provide goods used in manufacturing or assembly of products, which are in turn shipped to distribution centers, local warehouses, or retail locations. The goal of any supply chain is to provide finished goods to consumers.

At each stage of the supply chain issues arise that in turn raise questions critical to the success of the supply chain. Transportation questions—modes of transport, fleet sizing, and more—apply throughout the supply chain. At the consumer end of the supply chain customer service requirements must be set and decisions on customer sourcing and customer relationship management (CRM) must be made. At the central manufacturing and distribution stages questions about optimal inventory replenishment and production planning and scheduling are uppermost. Finally, at the initial supplier stage buyers must decide what to buy, in what quantities, where, and from whom. . Figure 1 shows the structure of a typical supply chain and the issues that arise at various stages.

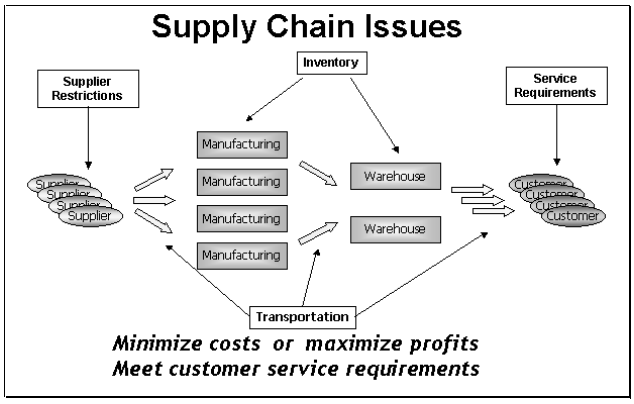


Figure 1. A typical supply chain and related issues.

"PUSH" OR PRODUCTION-FOCUSED PLANNING

Traditional planning techniques took too little notice of the linkage between the stages of the supply chain. At the consumer stage, a sales forecast predicted what goods could be sold and in what quantities. Back at the manufacturing stage planners looked at goods on hand, plant capacities, and work in progress—but not at the sales forecasts—and planned to manufacture the mix and quantities of products that they believed their operations could produce. These goods were then "pushed" through the remainder of the supply chain to be purchased (hopefully) by consumers. Too often, demand for sought-after items went unmet due to insufficient production while inventories of low-demand, over-produced items piled up.

"PULL" OR CUSTOMER-FOCUSED PLANNING

A more modern approach emphasizes that all supply chain activities are interrelated and links all planning to the sales forecast. Forecast sales combine with on-hand stock information to drive inventory and warehousing needs, which in turn feeds requirements to the manufacturing stage. Combined with work in progress and components/materials on hand, the manufacturing needs drive procurement requirements. Thus, all activity in the supply chain is pulled forward by the demand at the consumer level, in a "pull" or customer-focused model. This is the approach that SAS adopts in its supply chain optimization solutions.

SAS SUPPLY CHAIN SOLUTIONS

In addition to adopting the customer-centric "pull" model for supply chain planning, SAS supply chain solutions carry a number of other distinct advantages. SAS' data access and data warehousing skill enables SAS supply chain solutions can draw relevant information from ERP (Enterprise Resource Planning) systems, corporate legacy systems, or any source of interest. In generating the sales forecast, SAS solutions leverage the outstanding forecasting capabilities of the SAS System. Modeling and optimization utilizes the established SAS expertise in operations research and management science.

Overall, SAS supply chain solutions are being designed to be easily customizable to meet specific supply chain needs and to add value whether they are used individually or in an integrated supply chain optimization solution. Web interfaces for these solutions are being designed to provide a consistent, familiar

appearance that gives business users an intuitive grasp of the issues and options without requiring an in-depth knowledge of the underlying methods used in supply chain planning.

SAS supply chain solutions fall into four major categories. Supplier Management is the focus of this paper, and includes both Supplier Performance Rating (SPR) and the Supplier Portfolio Optimizer (SPO). Production Planning leverages SAS' established and field-tested project and resource management capabilities, and includes such solutions as Advanced Planning and Scheduling (APS) and Cycle Time Reduction (CTR). For more information on Cycle Time Reduction, see Jennings and Kulkarni [2001].

Further along the supply chain, Demand and Inventory Planning includes both the SAS Demand Planning Solution and the Inventory Replenishment Planner (IRP) for identifying, testing, and maintaining replenishment policies that meet customer service goals at lowest cost. Enterprise Supply Chain Planning takes a broad view of the supply chain, using a network approach to plan for production, packaging and distribution. Finally, a specialized data model, the Supply Chain Data Warehouse, is being developed to serve as a foundation for these solutions.

SUPPLIER MANAGEMENT SOLUTIONS

As noted earlier, SAS supplier management solutions consist of Supplier Performance Rating (SPR) and Supplier Portfolio Optimization (SPO). SPR and SPO are also key components of SAS' Supplier Relationship Management (SRM) solution, complementing the Procurement Vision product and adding analytical planning power to the visibility into current purchasing practices that Procurement Vision affords.

SUPPLIER PERFORMANCE RATING (SPR)

Supplier Performance Rating (SPR) is designed to rate and rank suppliers when considering multiple performance criteria simultaneously. It provides an automatic method for scoring and comparing suppliers, eliminates the need to determine weights for calculating the scores, and can account for business rules in the process.

SUPPLIER PERFORMANCE RATING: BUSINESS PROBLEM

In many businesses, data on supplier performance is collected and used for evaluating suppliers. Unfortunately, grading and comparing suppliers based on this data often is not straightforward, due to the presence of numerous and possibly conflicting evaluation criteria. For example, if one supplier outperforms all others according to one performance criterion but fails to achieve satisfactory levels on other criteria, it becomes unclear how to proceed with the comparison.

The traditional solution is to assign a fixed weight to each criterion to form an aggregated, weighted score for each supplier. Usually, weights are chosen to support specific business rules (such as weighting quality measures more heavily than financial measures to reflect their greater importance). Several problems can arise from this approach, including the following:

- Weights are subjective, difficult to agree on, and have a tremendous effect on the final scoring.
- It is difficult to balance relatively strong and relatively weak performances for different criteria.
- Differences in units of measurement for the criteria can

distort the influence of the weights used in the scoring.

The business problem can be stated most simply as **how to best rate suppliers on the basis of multiple, possibly conflicting, performance measures and account for business rules.**

SUPPLIER PERFORMANCE RATING: SOLUTION

Supplier Performance Rating (SPR) implements an innovative solution to this business problem. SPR solves the problem of differing measurement units among criteria by normalizing the performance data. This eliminates units from the measures, removes distortions associated with differences in units, and provides for more balanced comparisons.

SPR solves the difficult problem of determining weights by evaluating each supplier in isolation and optimizing the supplier's performance relative to all other suppliers (based on Data Envelopment Analysis). Automatically calculating optimal weights for each supplier's performance criteria avoids the problems resulting from assigning fixed weights to all suppliers.

SPR uses the optimally calculated weights to compile the relative scores used to compare and rank the suppliers. SPR captures business rules by enabling you to place limits and other restrictions on the weights used for the various performance criteria. This establishes rules on the relative importance of the criteria, and can also be used to account for imprecise performance information.

The SPR methodology is driven by the performance data. After normalizing each performance criterion, SPR uses linear programming to calculate weights on each supplier's performance criteria that optimize the supplier's overall performance rating. This rating is the weighted sum of the supplier's individual performance scores, with the weights being determined individually (and optimally) for each supplier. When this process is complete a ranking of all suppliers is possible.

SUPPLIER PERFORMANCE RATING: REPORTING

SPR produces supplier rating reports in two formats: ranked listings of suppliers and ranked bar charts of suppliers, each annotated with ranking, tier, and relative score (scaled from 0-100). For each type of report, the tier classification can be done on the basis of either the relative supplier score or the percentiles on the relative supplier score. Figure 2 shows one such report, with the tiered optimal supplier scores displayed in chart form.

Optimal ranking using SPR

Tier (by optimal scores)	OSM Ranking	Unit	Name	Relative score
1	1	614733525	VICROY ELECTRONICS	100.00%
	2	108323585	DELTA ELECTRONICS CO	84.64%
	3	026424556	MORRIS INC	66.83%
	4	117395095	LABELSTAR	63.69%
2	5	007322423	SIGNCO COMPANY	61.57%
	6	053122107	MARKETING SALES CO	60.47%
	7	002895175	MILTON EQUIPMENT	57.64%
	8	189839350	GHZ ENTERPRISES	57.14%
	11	872995345	JERSEY ELECTRONICS	56.17%
	10	038804662	MMA INC.	56.17%
	9	043565324	ESS WEAVER INC	56.17%
3	12	043308105	COMPUTER IMAGE SYSTE	47.98%
	13	054368774	MARKMATE	45.96%
4	14	075101345	MEDCO SALES CO	45.88%
	15	043817774	QUESTCO COMPANY	39.55%
	16	035453638	DCL INC	36.56%

Figure 2. A Supplier Performance Rating Report.

Upcoming releases of SPR will add the ability to determine supplier tiers by more advanced methods, including the use of statistical clustering techniques.

SUPPLIER PERFORMANCE RATING: POSSIBLE USES

SPR has a broad range of possible uses, and the underlying techniques used by SPR have almost unlimited applicability. Within the confines of supplier management, the most immediate use of SPR is in the periodic review of supplier performance. In this role, SPR can easily provide its ratings, rankings, and tier assignments as input to the Supplier Portfolio Optimizer, discussed in the following sections of this paper. Another opportunity to use SPR occurs in the Request For Proposal (RFP) or Request For Information (RFI) process, during which the purchasing company may need to shorten the list of bidding suppliers based on their past performance. In this scenario SPR's performance ratings could easily drive the reduction in the pool of bidding suppliers.

SUPPLIER PORTFOLIO OPTIMIZER (SPO)

The Supplier Portfolio Optimizer (SPO) assists in structuring and restructuring portfolios of suppliers, with a goal of maximizing the buyer's benefit while meeting specific requirements on portfolio makeup. SPO relies on rationalized supplier data describing suppliers and the goods that they provide, available from a number of sources such as Dun & Bradstreet. SPO provides guidance for answering strategic questions such as:

- Who should we buy from?
- Should we spend more or less with a supplier?
- What should our expected risk be?
- What should we buy from a given supplier?
- How should we alter our buying practices?

SUPPLIER PORTFOLIO OPTIMIZER: BUSINESS PROBLEM

Often, businesses don't have enough information on their suppliers of parts, raw materials, and other critical items. They cannot readily and easily determine how much they are spending, what they are purchasing, who their top suppliers are, or the answers to many other important questions. Surfacing such information effectively is one key to improving the buyer's position and to negotiating better relationships with suppliers. Another key to managing supplier relationships is an organized approach to moving from your current supplier portfolio to your desired supplier portfolio. In improving a supplier portfolio, one of the most often-mentioned goals is better negotiating leverage. Typically, though, buyers face restrictions as they pursue this goal. These restrictions or requirements may originate from internal business rules on supplier selection, from regulations specific to the business's industry, or from other sources. Some examples of these restrictions include:

- "At least 5% of our purchases should be made with small businesses."
- "We should have 5 to 10 suppliers of paper goods."
- "To ensure quality, buy at least 25% of all supplies from ISO compliant businesses."
- "The average financial stress score (FSS) of our supplier portfolio should not exceed 1.7 and the average supplier evaluation risk (SER) should not go beyond 2."

What's needed is a method for reshaping the supplier portfolio, focusing on the goal of maximizing negotiating power while adhering to whatever restrictions may apply.

SUPPLIER PORTFOLIO OPTIMIZER: SOLUTION

SPO's solution technique is based on mathematical optimization (specifically, mixed-integer programming) and is aimed at making procurement choices that maximize negotiating leverage within the restrictions created by the business rules governing the makeup of the supplier portfolio. As is true with SPR, the user interface provided by SPO does not require users to understand the fine points of mathematical optimization, but only to grasp the concepts motivating the demographic, budgetary, geographic, and other business rules on the supplier portfolio.

The SPO user selects from a menu of possible business rules to apply, and customizes each rule used by specifying parameter values such as minimum percentages, budgetary upper and lower bounds, and regional distribution targets. SPO then finds the combination of purchases and vendors that meets or exceeds the specified requirements and maximizes purchasing leverage. SPO achieves this by consolidating purchases to buy mainly from suppliers for whom the company is a major client, while simultaneously maintaining a balanced and diversified portfolio as required by the business rules.

SUPPLIER PORTFOLIO OPTIMIZER: REPORTING

SPO can produce a wide variety of reports describing the optimized supplier portfolio and the accompanying purchasing recommendations. Examples include reports on detailed and aggregated purchases from suppliers, recommended suppliers for specific commodities, and geographical purchasing summaries. Additionally, a summary executive report offers multiple views of the current and optimized supplier portfolios, highlighting the advantages that optimization offers.

Detailed Purchases
Scenario: Reduction_DB_NewData

UN/SPSC	Commodity	D-U-N-S	Supplier	Current Purchases ($)	Proposed Purchases ($)	Change (%)
11101705	ALUMINUM	059651240	TTR METALS	416	383	(8.00%)
11101715	COPPER	056778780	UPTON PRODUCTS	92	85	(8.00%)
11131600	OTHER ANIMAL PRODUCTS	113317523	SCI-KON CORP	23	21	(8.00%)
11162108	WIRE MESH FABRIC OR CLOTH	151549912	THUNDER FABRICATION	5	5	(8.00%)
12000000	INDUSTRIAL CHEMICAL AND GAS MA	006337463	ARKATOOL	7	1,173	17700.95%
		038804662	MMA INC.	29	.	.
		051439222	DATA STORAGE CORP	71	.	.
		109150086	BANDWITH SYSTEMS	45	.	.
		113317523	SCI-KON CORP	1,124	.	.
12000000	INDUSTRIAL CHEMICAL AND GAS MA			1,275	1,173	
12000000				1,275	1,173	
12102105	FREONS	034829390	KAPICOM SUPPLIES	58	54	(8.00%)

Figure 3. A Supplier Portfolio Optimizer report.

SUPPLIER PORTFOLIO OPTIMIZER: POSSIBLE USES

The usage possibilities for SPO are quite similar to those for SPR. First, SPO is well suited for use in the periodic review of a supplier portfolio. This may be especially important if, for example, the purchasing firm completes a merger or acquisition and must deal with an augmented supplier portfolio. Additionally, SPO can (like SPR) assist in the Request For Proposal (RFP) or Request For Information (RFI) process. In this role, SPO can be refocused to minimize overall spend while satisfying the requirements outlined in the RFP or RFI, identifying a lowest-cost set of bidders.

SUPPLY CHAIN SOLUTIONS: COMPONENT INTEGRATION

The integration of the SAS supply chain solutions mirrors the causal relationships highlighted in the "pull" model of the supply

chain. Demand Planning at the customer end of the supply chain feeds critical information to Inventory Replenishment Planning at the warehousing and distribution level, which in turn supplies requirements for Advanced Planning and Scheduling at the manufacturing stage. Other influences on manufacturing scheduling include Cycle Time Reduction and Equipment Maintenance Scheduling. The manufacturing schedule generates requirements for supplies, which are assigned to current and potential suppliers via Supplier Management. The result is an end-to-end plan for the improvement of overall supply chain performance.

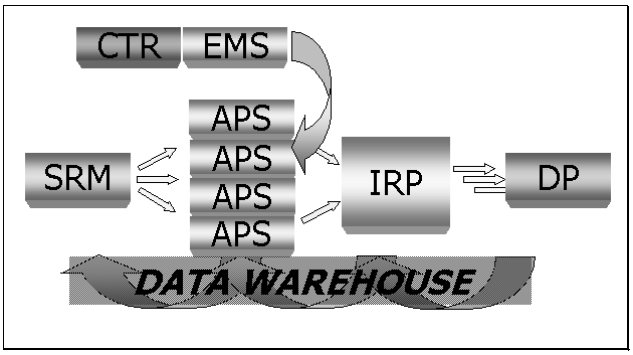


Figure 4. Component Integration

For any particular situation, any or all of these solutions can be implemented, and can add value to the supply chain either individually or by working together.

SUPPLY CHAIN SOLUTIONS: SUPPORTING DATA ARCHITECTURE

Underlying and supporting this planning work is the Supply Chain Data Warehouse, the repository of all relevant data on the past and current performance and future direction of the supply chain. Owing to the ease with which SAS can access data from virtually any source and in any format, the Supply Chain Data Warehouse can draw needed information from ERP systems, business legacy systems, or wherever the data resides.

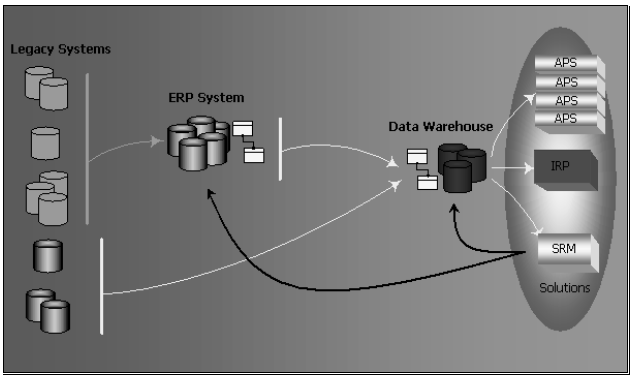


Figure 5. Supporting Data Architecture

The supply chain solutions draw data from the Supply Chain Data Warehouse and produce tactical and strategic plans for the supply chain. The information describing these plans is stored in the data warehouse and can also be fed back to ERP systems and legacy systems in order to put the plans into motion.

SUPPLY CHAIN SOLUTIONS: SERVER ARCHITECTURE

All SAS supply chain solution are designed with Web interfaces and can run in a distributed processing environment with a thin client. Only a Web browser and the necessary authorization are needed to access the solutions and their reports, while an application server can support each solution and another server can perform the needed optimization. By enabling such a thin client interface, SAS supply chain solutions make it easy to distribute both the planning power and the strategic and tactical supply chain plans that they provide.

CONCLUSION

SAS is developing a set of solutions targeted at assisting with the assembly, structuring, restructuring, and management of a large supplier portfolio. With Supplier Performance Rating and Supplier Portfolio Optimization, SAS applies its established optimization expertise and power to the problems of rating supplier performance in a balanced and comprehensive manner and configuring supplier selection and purchasing to maximize leverage. Each solution is designed to be customized to meet individual client needs, and can participate not only in a complete Supplier Relationship Management solution but also in larger Supply Chain Optimization solutions.

REFERENCES

Jennings, D., and Kulkarni, R. (2001), "Cycle Down To Launch: Streamlining Production at Lockheed Martin," paper presented at SUGI 26 (SAS Users' Group International), Long Beach, CA.

SAS Institute Inc. (2000), "Supplier Management: Optimally Managing the Supplier-Buyer Relationship," Internal Document, Cary, NC: SAS Institute Inc.

WEB RESOURCES

For more information on SAS Supply Chain Optimization, see http://www.sas.com/supplychain . This site includes information on the various solutions being developed by the SAS Supply Chain Center as well as demonstration versions of many of the solutions.

For information on the SAS Solution for Supplier Relationship Management, see http://www.sas.com/solutions/srm .

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Ed Hughes
SAS Institute Inc.
Office R-4109, SAS Campus Drive
Cary, NC 27513
Phone: (919) 531-6916
Fax: (919) 677-4444
Email: Ed.Hughes@sas.com
Web: http://www.sas.com/supplychain

Paper # P505
Florida Community College System
Putting Minds to Work

Jeanette C. Humphrey
Tallahassee Community College

Howard Campbell
And
Brian Walsh
Division of Community College
Department of Education
State of Florida

In the Beginning:

In 1988 the legislators in the State of Florida mandated that the Florida Community College System, comprised of the 28 community colleges in Florida, begin reporting student type information to the state in a format to be defined by the Division of Community Colleges (DCC). The purpose of this directive was to give the state access to detail student level information, allowing the state to produce federal Integrated Postsecondary Education Data System (IPEDS) reports, state reports for funding, and conduct detail data research and analysis. Information Systems of Florida (ISF) was awarded the state contract to produce a long-range plan and database design specifications, including data definitions.

The Plan:

ISF's completed its study and suggested the following:

- The data should be reported by the community colleges annually to the state in the following manner:
    a. Summer term data should be submitted once, in October;
    b. Preliminary fall term data should be submitted once at the beginning of the term (October) and final fall data submitted after the term is completed (January.)
    c. Preliminary spring term data should be submitted once at the beginning of the term (January) and final spring data after the term is completed (June.)
- For the Initial Phase of the project, the data should be used only for federal reporting, such as IPEDS. During this initial phase, the colleges should receive reports, which they could verify locally. After allowing the colleges a few years to verify their data, the state should then use Full-Time Equivalent (FTE) generated from the Student Data Base (SDB) for funding and research.
- The colleges should report the data in an ASCII format. The DCC could then use the software of its choice to read the ASCII files into a relational type database.
- The following data record definitions were suggested:
    a. Demographic Record – (max occurs 1 time) with demographic type information, including accumulators for total student hours taken by term and

to date. Include only students enrolled, graduating, or receiving acceleration credit during the current term.

b. Entry Level Test Record - (can occur multiple times) with most recent Entry Level Test (ACT SAT) information.
c. Acceleration Record – (can occur multiple times) with Credit by Exam taken during reporting term.
d. Program of Study Record – (can occur multiple times) with the Students declared program(s) of study.
e. Completion Record – (can occur multiple times) with the program completion information.
f. Course Record – (can occur multiple times) with information on course(s) taken during reporting term.
g. Financial Aid Record – (can occur multiple times) with information on financial aid received during reporting term.

ISF's plan was accepted, as proposed. Detail information on the current Student Data Base Dictionary is available at the following DCC web site: http://www.dcc.firn.edu/dccpubs.htm#deds

From the college's perspective, the DCC realized the financial hardship created from expending resources to verify data that was produced primarily as a reporting tool. Producing the data was difficult enough; to expect the colleges to verify the data to the level desired was unrealistic. In a presentation to the *Florida Assn. for Institutional Research* in 1989, Howard Campbell shared his vision for the future: the state would use this data for reporting, funding, and research. In addition, the colleges could use this same data for research and local reporting purposes, utilizing the efforts expended for local benefit. The state would also benefit from college participation: the increased utilization by the local institution, would allow more data anomalies to be found, explained, and corrected.

After reviewing system requirements, the DCC chose the SAS programming language, primarily because of its efficiency, ease of use, and connectivity to IBM's Data Base DB2 relational database, which was mandated by the state legislature.

To assist the DCC in data related questions, the state created Management Information System Task Force (MISATFOR), an advisory group for the purpose of discussion of questions and problems related to the data. Currently, the President of each college appoints one member to MISATFOR. In addition, personnel from the DCC and the registrars group are included. MISASTFOR meets six times a year.

As the years passed, the use of this data by the state has increased dramatically. To provide follow-up information on our students, this data is now matched against other state databases. For example, to determine success of our students after leaving the community college, the college data is matched to state unemployment files, the State University System, Private College Stipend files, Public High School files, Job Training Partnership Act (JTPA), WAGES, Public Assistance Files, and military files. Community colleges are now aware of how many of their students go on to get a high wage jobs, go into the military, or eventually graduate with a bachelors degree. In addition, the information fed back to the colleges identifies whether students were economically or academically disadvantaged.

Another use of the data is performance based funding and accountability. Much of the college's state revenue is now based on outcomes, not the number of seats filled in a classroom. As the funding became more and more tied to the data, the importance of accuracy increased.

Micro-computer Project – Sharing the Information:

In 1995, Carol Hawkins, Dean of Information Technology at Polk Community College (PCC) in Winter Haven, Fl., wrote a proposal to the Division of Community College to fund the creation of a micro-computer system that would emulate the system and data stored at the DCC. Ms. Hawkins proposed that PCC provide the technical expertise for the project, in exchange the DCC would share costs for the project and distribute the resulting system to the other community colleges.

Phase I: The following year the DCC gave PCC the first of three contracts to produce a stand-alone point and click turnkey computer system to be shared with the other Florida Community Colleges. The programming language for this turnkey system was designated by the DCC to be SAS. The core of this system was to interact with a CDROM of all the data from all the community colleges for one academic year. The DCC would supply a CDROM to the colleges after the close of each reporting year.

Included in the initial Phase I were funds to purchase hardware, a Dec Alpha AXP; software, including the SAS Academic Computing Offer; and funds to provide SAS training. In addition, the contract included funds for personnel costs to design the system and develop a prototype. Jeanette C. Humphrey, Coordinator of Research and Reports at PCC, was designated as the project leader, analyst, and programmer. At this point in time, Ms. Humphrey had programming experience; however, had never used SAS.

Phase II: In Phase II, conducted the second year, Ms. Humphrey transported the prototype system to run on a microcomputer running Windows NT, which was now able to handle the large data files. In addition, she added additional reporting modules, on-line documentation, and a module to interact with 'Local' ASCII Student Data.

The system contained programs to load the ASCII data into SAS datasets with similar data names as those found on the CDROM. This ASCII data was the same ASCII student data sent by the college to the DCC each term and contained all the student information of enrollments at the local college, including student social security number and student name. This step was an important inclusion for a number of reasons. First, the data could be used in a timely manner. As soon as the data was pulled for submission to the state, the ASCII data could be read into SAS datasets and used by the college research personnel to verify the data submission, before the end of the submission window. Second, since this data contained the student social security number and name, the college mainframe system could be used to verify questionable data. Finally, for complex projects, like tracking systems, this was the best data, since it contained student identifiers that could be matched against additional data from the institution's mainframe and other local sources.

Phase III: The final Phase of the project provided funds to purchase a computer for each of the 28 community colleges, provide training, and user documentation to the colleges. In addition, enhancements included a Local side for the relatively new state mandated Personnel Data Base

and Facilities Data Bases. College personnel were trained at PCC by Ms. Humphrey and returned to their home campus with their computer.

System Requirements:

The following is a synopsis of the important components of the Micro-computer Project.

Hardware/Software Requirements: The basic system provided to each of the 28 Florida Community Colleges was a COMPAQ 400 MHz Pentium with 12 gig hard drive, running Windows NT. Minimum SAS components necessary to run the system were: Base SAS, SAS/AF, SAS/FSP, SAS/Graph, SAS/Assist, and SAS/STAT. In addition, Word and Excel were also included. Application software required 2 gig of hard drive, leaving 10 gig for reports and SAS temporary data sets. In testing, we found that more memory the better, since SAS uses all the memory it can find. Also, a significant amount of free disk space was required when working with large datasets.

Data: The Micro-computer system used either SAS Datasets provided to the colleges on a CDROM or SAS datasets from the Local ASCII data. The CDROM contained all the data for all the community colleges for one year. The student's name was deleted from the dataset and an identifying number, generated by an algorithm, replaced the social security number. This same algorithm was also used in successive years, allowing the tracking of a specific student over the years, without identifying that student. The Local data created by the system was from the same ASCII student data provided term by term to the Division of Community Colleges.

Documentation/Training: Ms. Humphrey at PCC provided the initial training to the community colleges. In addition, a step-by-step documentation manual was included with each system. SAS Training by SAS Institute was highly recommended.

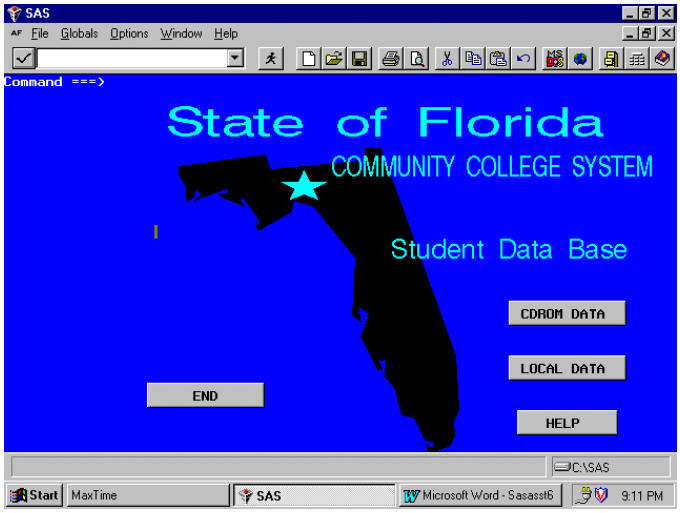
Updates/New Procedures: In the years following, the DCC provided new CDROMS of the current years data to the colleges at the close of each reporting year. These CDROMS have continued to work with little or no maintenance of the original system. Enhancements and updates to this system have been taken over by the DCC and are maintained at the state level. The state is currently evaluating the value of the new SAS/SCL, now SAS Component Language.

Knowledge: **Without the excellent training by SAS Institute and support by SAS Technical Support, this project would not have been possible.**

System Design:

Initial phase of the system included two parts: The CDROM section provided interaction with the Annual CDROM. This CDROM contained all the data for all the community colleges for one year. The Local Data section allowed interaction with the original ASCII student data sent term by term to the Division of Community College. This original data was read into SAS Datasets by the Micro-computer system.

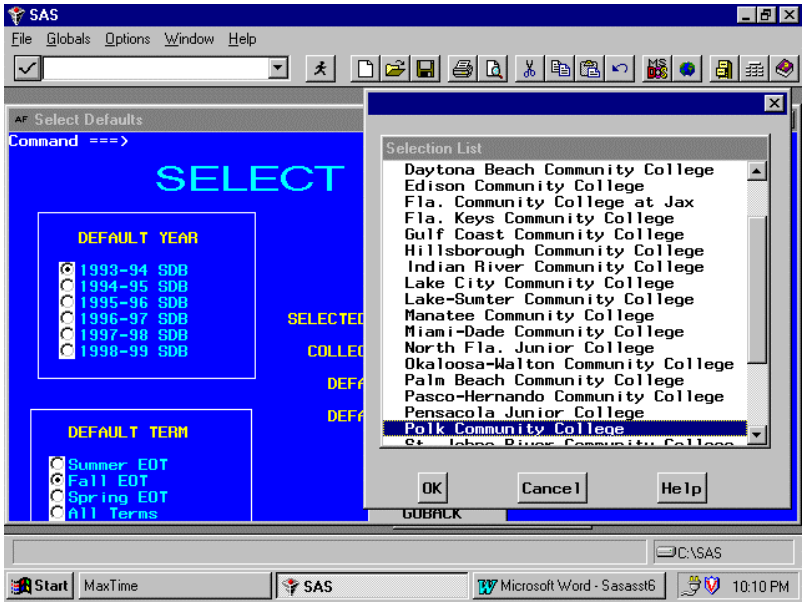
the Annual CDROM. This CDROM contained all the data for all the community colleges for one year. The Local Data section allowed interaction with the original ASCII student data sent term by term to the Division of Community College. This original data was read into SAS Datasets by the Micro-computer system.



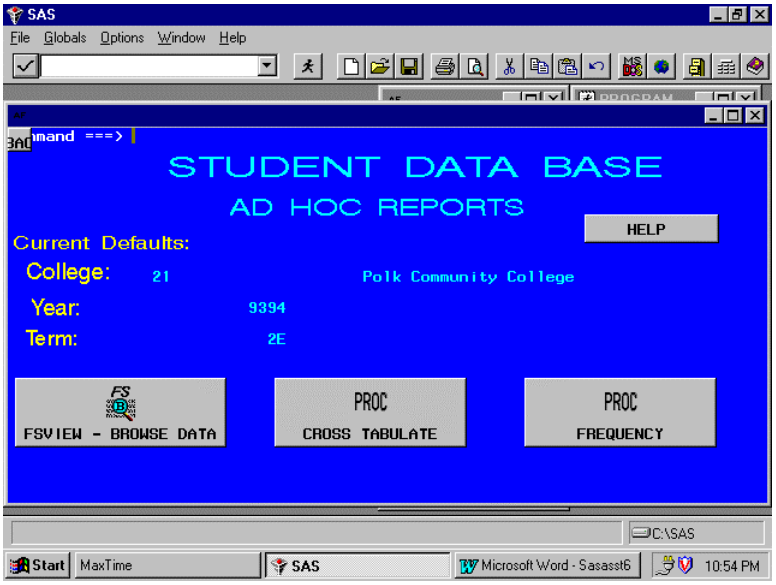
System Data Structure: The location of the SAS data sets for use by the system was an integral part of the system design of the 'Local' side of the system. Each term of student data resided within a subdirectory named the actual term identifier. For example, the Fall Term, known as term 2, in the year 1999 would be identified as 299. Each of the subdirectories are located under C:\SAS_DATA, so Fall 1999 would be found under C:\SAS_DATA\299. Using this structure enabled the use of macros for processing with relative ease.

Similarly, in the CDROM side of the system, the datasets on the CDROM have the relevant year embedded as part of the name of the dataset. Using this method, verification that the correct CDROM resides in the drive can take place before reading a dataset.

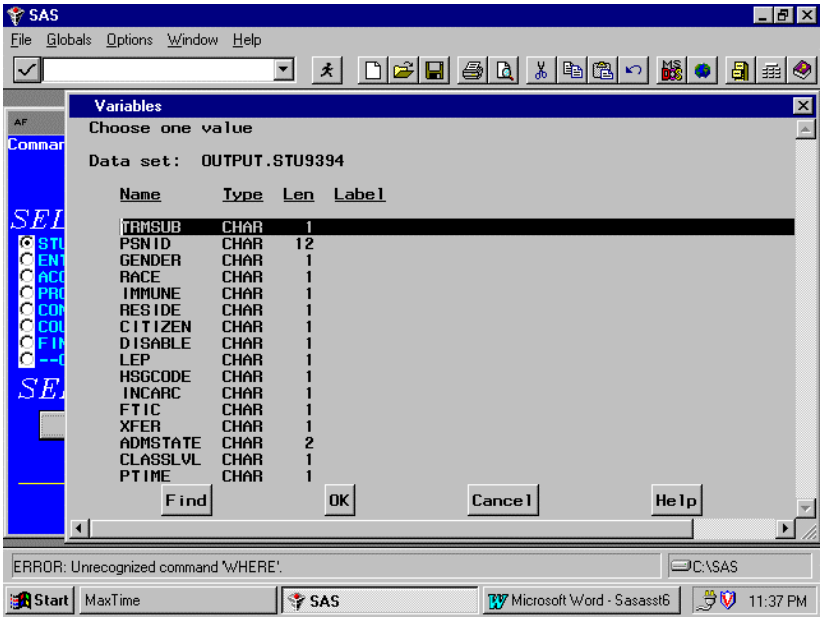
The system took advantage of many of SAS/AF abilities to provide Pop-up selection lists and radio boxes.



Another beneficial section of the system allowed the users to use FSVIEW to scroll through their data or produce cross tab or frequency tables. Again, the system utilized the point and click



features of SAS/AF.



Some Favorite SCL Code:

- VARLIST - to populate a pop-up list from a SAS Dataset, without hard coding variable names (see example above) ;
- PMENU – to provide interactive pull-down menus;
- PREVIEW WINDOWS – to view output;
- SUBMIT CONTINUE – to imbed SAS Code in SAS/SCL;
- REPLACE – to store string in a SAS/SCL Variable;
- Modular ‘Link’ Programming;
- MACRO Substitutions;

- Passing Variables between SAS/SCL Programs;
- Where statements applied to a SAS/SCL Dataset;
- On-line Help using the Help functions.

Conclusion:

As a result of this project, both the state and the community colleges have benefited. The most obvious benefit to the state has been more accurate reliable data. In addition to the state benefits, many colleges have experienced the following benefits:

- Establishing the Student Data Base as a research base has provided the Florida Community College System with a reliable source of data for research and improved the accuracy of the data used in state reporting and funding.

- Sharing of SAS programs between sister community colleges has increased good will among the sister colleges. Since all the colleges are using the same data structures, only minor modifications are necessary for the programs to run at the sister college. Many SAS programs have been shared between the reports/research personnel at the colleges, including some extensive SAS systems. For example, Tallahassee Community College recently shared a tracking system written in SAS using the Local SDB data. This system is macro driven and easily modified for use at any of the colleges. Seminole Community College also shared a comprehensive tracking system developed by consultant, W. M. Consulting in Tallahassee. They are hoping that future costs of enhancements to their tracking system may be shared by other community colleges. Of course, future benefits would also be shared.

- Using this data for research has increased the understanding of the databases, how they relate to the college, and how they relate to other state databases.

- Colleges are now able to replicate the state SAS programs, which are used to report data and produce funding. This ability allows colleges to further understand and verify the state's programs and identify reasons for data anomalies.

- Florida Community College SAS Users group was created to share resources and provide training. The Users Group meets in conjunction with the semi-monthly MISATFOR meetings and has successfully addressed many SAS data issues pertinent to the community college system.

Acknowledgements:

TCC for the time to prepare this paper and its support of the FCCS SAS Users Group.

Carol Hawkins, Vice-President of Institutional Effectiveness, Planning, and Information Systems of Seminole Community College, formerly employed by PCC, for sharing the vision, writing the grant proposal, and supporting the project.

Florida State Legislature for funding the projects.

SAS Technical Support for valuable support with quick response time.

Addresses:

Questions pertaining to this article should be addressed to:

Jeanette C. Humphrey
Tallahassee Community College
444 Appleyard Dr.
Tallahassee, Florida 32304-2895
(850)201-6083
Humphrej@tcc.cc.fl.us

Howard Campbell or Brian Walsh
Bureau of Research and Information Systems
Division of Community Colleges
Department of Education
1314 Turlington Building
325 W. Gaines Street
Tallahassee, Florida 32399-0400
(850)488-8597
Howard@flccs.org Brian@flccs.org

Using Recursion in the SAS® System

David Ward, InterNext, Inc., Somerset, NJ

ABSTRACT

Recursion is a friend of most 3rd or 4th generation language programmers. It can be used to slice hundreds of lines of code from complex algorithms, or bring a system to its knees after having consumed all its resources. The concept of recursion and useful applications of it using SAS®/Macro and SAS®/AF software will be presented. You will learn how to traverse directories, trees of information, and other interesting yet challenging topics. After reading this paper you should have a better understanding of when and how to use recursion in the SAS® system.

INTRODUCTION

Recursion refers to the case when a function or sub-routine calls itself. In order for recursion to successfully take place the programming language must provide separate memory areas for the local variables in each function call. Therefore, the data step can not be used to execute a recursive algorithm (although it can be mimicked). Instead, a SAS® programmer must rely on either the SAS® Macro language or SCL to institute recursion. Often the first example of recursion programming students encounter is in the world of sorting. For simplicity's sake, this paper will not examine sorting algorithms in SAS® because simpler examples exist to get you comfortable with recursion.

OUR FIRST ALGORITHM

Let's say you have genealogical data that is updated each month. Each person could potentially have a different number of ancestors, and that number could also change from month to month. Your assignment is to print a report for each person, listing their ancestors (assume fathers).

```
Name          Father         Mother
====          ======         ======
John Jones    Mark Jones     Mary Fawcett
Sarah White   Henry Little   Jane Smith
Mark Jones    David Jones    Harriet Lawler
```

From the data above you can see that John has two ancestors listed (Mark and David Jones), and Sarah White only has one ancestor listed. The thought might occur to you to use proc SQL to solve the problem of getting all ancestors for each person, maybe something like:

```
Proc sql noprint;
  Create table ancestors as
  Select d1.name, d2.father as ancestor1 from
  Data as d1, data as d2 on d1.name=d1.father;
Quit;
```

You should quickly see the problem with this approach – you need one join for every ancestor a person has. If a person has 20 ancestors you would need 20 join statements. Instead, using recursion, we do not need to know how deep we must go for each person to obtain their ancestors. A basic outline of the algorithm:

```
1)  Find the father of person A
2)  If person A has no father listed, leave
3)  Otherwise person A's father becomes person A
4)  Go to step 1
```

The algorithm will leave at the appropriate step for each person. The author is aware that this example could be solved with some data step tricks, but it has been chosen for simplicity sake. A macro approach could be:

```
%global ancestors;
%macro getAncestors(person);
  %local father; %let father=;
  data _null_;
    set data(where=(person="&person"));
    call symput('father',father);
  run;
  %if %length(&father)>0 %then %do;
    %let ancestors=&ancestors/&father;
    %getAncestors(&father);
  %end;
%mend;
%let ancestors=;
%getAncestors(John Jones);
%put NOTE: Ancestors: &ancestors;
```

Which produces this in the log:

```
NOTE: Ancestors: /Mark Jones/David Jones
```

EXAMPLE 1: SEARCHING FOR FILE NAMES

You can use recursion to get a list of all files matching a pattern name in all sub-directories under a given directory. The idea is simple: loop through the names of the files in the current directory, then, for all sub-directories, call the macro on itself with the sub-directory as the new parent directory. A sample macro implementation follows:

```
data files;
  length file $200;
stop; run;
%macro findMatches (dir,pattern);
  %put NOTE: Searching &dir;
  /** DEFINE LOCAL MACRO VARAIBLES **/
  %local files i dnum did matches rc
         filename;
  /* SET FILENAME TO CURRENT DIRECTORY */
  filename dir "&dir";
  %let did=%sysfunc(dopen(dir));
  %let dnum=%sysfunc(dnum(&did));
  %do i = 1 %to &dnum;
    %let files
      =&files/%sysfunc(dread(&did,&i));
    %if %index(
        %upcase(%scan(&files,&i,/)),
        %upcase(&pattern)) %then
        %let matches
        =&matches/%scan(&files,&i,/);
  %end;
  %let rc=%sysfunc(dclose(&did));
  /** WRITE MATCHES **/
  %if %length(&matches)>0 %then %do;
    data matches;
      length file $200;
      %let i = 1;
      %do %while(
        %length(%scan(&matches,&i,/))>0
       );
         file="&dir\%scan(&matches,&i,/)";
         output;
         %let i=%eval(&i+1);
      %end;
    run;
    proc append base=files new=matches;
    run;
  %end;
  /** CALL MACRO FOR EACH SUBDIRECTORY **/
  %let filename=file;
  %local i;
  %do i = 1 %to &dnum;
    %let rc=
      %sysfunc(filename(filename,
                &dir\%scan(&files,&i,/)));
    %let did=%sysfunc(dopen(file));
    /** IF CANT OPEN FILE ASSUME DIRECTORY **/
```

```
    %if &did^=0 %then %do;
      %let rc=%sysfunc(dclose(&did));
      %findMatches(&dir\%scan(&files,&i,/),
                   &pattern);
    %end;
  %end;
%mend;
options nomprint;
%findMatches(d:\dward,.exe);
```

EXAMPLE 2: CONVERTING AN SCL LIST INTO A STRING

The SCL list is a powerful tool available to SAS®/AF developers. Lists are filled with items that can be numeric, character, or another list, thus a hierarchy of data can be put into a list in the form of data tree. In order to traverse a tree structure you can use a recursive SCL method. Once the method has been created you could use it like this:

```
Dcl char string;
init:
  ** BUILD THE SAMPLE LIST **;
  l={ one=1, two={ a='LETTER A', b='LETTER B',
      c={ I='RN I', II='RN II' } } };
  ** PUT THE LIST INTO A STRING **;
  string=method('cptools.trackuser.methods.scl',
         'stufflist',l);
  ** DISPLAY THE STRING TO THE LOG **;
  put string=;
  ** PUT THE STRING INTO ANOTHER LIST **;
  m=method('cptools.trackuser.methods.scl',
    'expandlist',string);
  rc=putlist(m,'m',0);
return;
```

The above example uses two methods, stufflist and expandlist. The code for the methods (it doesn't format nicely in this single-column document):

```
** STUFFLIST 1/10/01 DWard
   Create a string that represents a list
   Restrictions: You cannot have a list item
that contains braces {}, and equals sign, or a
comma **;
length string name value $32767;
stuffList: method listid:i:list
optional=string:u:char return=char;
  if listlen(listid)<=0 then
     return('{}');
  do i = 1 to listlen(listid);
    if i=1 then string=trim(string)||'{';
    else string=trim(string)||',';
    if nameitem(listid,i)^='' then
       string=trim(string)||
       nameitem(listid,i)||'=';
    select itemtype(listid,i);
      when ('C') string=trim(string)||
         quote(getitemc(listid,i));
      when ('N') string=trim(string)||
         compress(getitemn(listid,i));
      otherwise rc=method('methods.scl',
         'stuffList',getiteml(listid,i),
          string);
    end;
    if i=listlen(listid) then
       string=trim(string)||'}';
  end;
  return(string);
endmethod;
** EXPANDLIST: Take a string and convert it into
a list
   String should be from stuffList
   1/10/01 DWard **;
Dcl list sublist;
expandList: method string:u:char
    optional=listid:u:list i:u:num
    return=list;
  if listlen(listid)>-1 then
     sublist=listid;
  else sublist=makelist();
  if length(string)<=2 then
     return(sublist);
  if i<=2 then i=2;
  do i = i to length(string);
    if substr(string,i,1)=',' then i=i-1;
    if substr(string,i,1)='}' then do;
      i=i+1;
      return(sublist);
    end;
    else do;
      name='';
      value=substr(string,i);
      np=indexc(value,'},');
      oi=i;
      i=i+np-1;
      if substr(string,i,1)='}' then
         i=i-1;
      value=substr(value,1,np-1);
      value=scan(value,1,',}');
      np=index(value,'=');
      if np then do;
        name=scan(value,1,'=');
        value=substr(value,np+1);
      end;
      if substr(value,1,1)='{' then do;
        sublist2=makelist();
        rc=insertl(sublist,sublist2,
           -1,name);
        i=oi+np+1;
        rc=method('methods.scl',
         'expandList',string,sublist2,i);
      end;
      else if substr(value,1,1)='"' then
        do;
        value=tranwrd(substr(value,2,
              length(value)-2),'""','"');
        rc=insertc(sublist,value,
           -1,name);
      end;
      else rc=insertn(sublist,value,
           -1,name);
    end;
  end;
  return(sublist);
endmethod;
```

The author used this technique when two SAS® sessions needed to communicate an SCL list over a TCP/IP socket connection that did not support sending the list, only ASCII data.

CONCLUSION

Recursion is a powerful technique that will allow you to perform advanced functions with your SAS® programs, whether they are written in SAS®/Macro or SCL.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

David L. Ward
InterNext, Inc.
254 Resnik Ct.
Somerset, NJ 08873
Work Phone: (732) 470-6783
Email: dward@SAS®help.com

Paper P507

Creating Student Academic Profiles

Janice K. McBee, Virginia Tech, Blacksburg, VA

ABSTRACT

University academic advisors rely on data from various sources ranging from a student's experiences prior to entering the university and a student's performance at the university. High school rank, grade point average, and SAT, ACT or other such scores typically make up some of the data elements. Transfer of credits from advanced placement high school courses or from courses taken at other institutions, e.g., community colleges adds to the student profile. In addition, performance at the university should be included in a student profile. The purpose of this paper is to join/merge data from various sources (MS Excel files, Banner Oracle tables and SAS datasets) using SAS on data from various sources ranging from a student's experiences prior to entering the university and a student's profile to be used in academic advising. SAS Enterprise Guide tasks and SAS procedures were used to develop a decision model to aid the advisors of freshman students in a particular department.

INTRODUCTION

This paper is intended for the beginning or intermediate SAS or SAS Enterprise Guide user. It is an example of using SAS Enterprise Guide to join SAS datasets with a client's (end-user's) provided data. The resultant SAS dataset becomes the data for a new SAS Enterprise Guide project and can be exported for additional use by the client.

The data that we use comes in various forms. We have student census extracts (SAS datasets and flat files) that are created each term. Clients will have their own data that they want merged with university data. It is important to know your data. When one joins datasets the variable/column descriptions should match to avoid erroneous results. To develop a student profile first, edit and verify your data, join datasets, and verify the resulting dataset.

STUDENT CENSUS DATA

Prior to Summer 2000 the student census data was legacy data and the flat files were converted to SAS datasets. As of Summer 2000, student census extracts (SAS datasets) are generated. A SAS version 8 program with PROC SQL was used by members of another department to join Banner Oracle tables and create student census SAS datasets. Our Decision Support Services team modifies the student census data by creating variables and adding labels in SAS version 6.12. This accommodates our Mac users/clients and creates SAS datasets that are compatible with Strategic Enrollment Management, a software product from the SCT Corporation. The student census extract is created every term and is used for official university reporting.

CLIENT DATA: MS EXCEL FILES

MS Excel files import directly into SAS Enterprise Guide (Figure 1). After importing, one can use the Tasks to analyze the data (Figure 2). Guide accesses data from various forms, from SAS datasets to HTML files (Figure 3). The data then can be saved as SAS datasets and merged with existing data.

Figure 1.

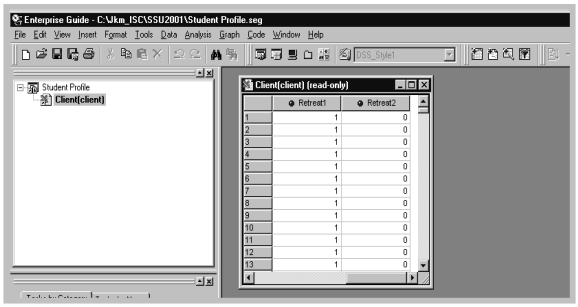


Figure 2.

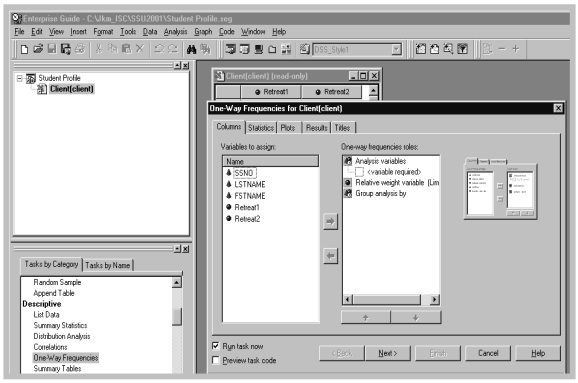
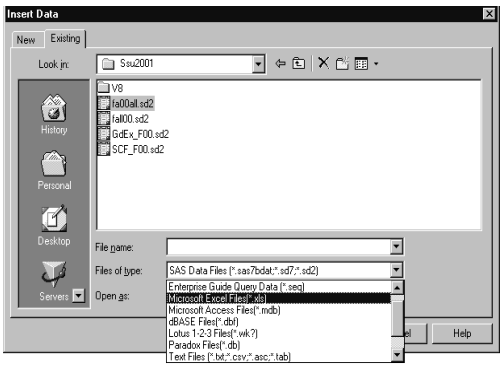


Figure 3.



The MS Excel file from the client originally had student identification (SSNO) defined as General. I changed the column attribute to text; however, when inserted into the Guide project, SSNO was interpreted as a numeric data type. Our student census files have student identification (SSNO) defined as character. Guide does not allow joining of data on mixed typed variables. Therefore, I wrote a program to modify the imported file to change the format of the SSNO to character and saved it at a SAS dataset [Client] .

JOIN/MERGE DATA

When data is in SAS datasets SAS Enterprise Guide is a versatile tool to use to merge data and then to analyze the resulting dataset. With Enterprise Guide one can merge SAS version 6.12 data with version 8 data. Note that it is better to keep SAS version 6 datasets is a different folder than SAS version 8 or one can convert datasets to a common release of SAS. In this study the census (version 6.12) data and term grade (version 8) were merged with the client's data (MS Excel converted to SAS version 8). In addition, term course grade data (version 6.12) was used to provide a current term profile (grade and description) of courses.

To merge the SAS datasets to be joined were placed in the project. I did the following:

- Placed data in project (optional)
- In Query Builder (Filter under Data) clicked on the Tables Tab (a in Figure 4)
- Clicked on Add Data button (b in Figure 4)
- In Add Data clicked on SCF_F00
- In Add Data clicked on grades_f00
- Renamed Query (a in Figure 5)
- Clicked OK (b in Figure 5) to create the Query
- Save joined data (optional)

Figure 4.

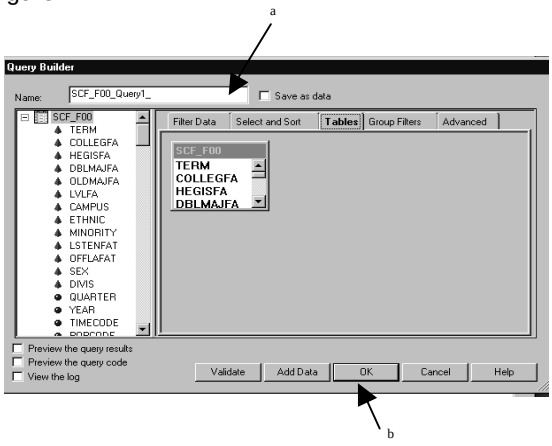
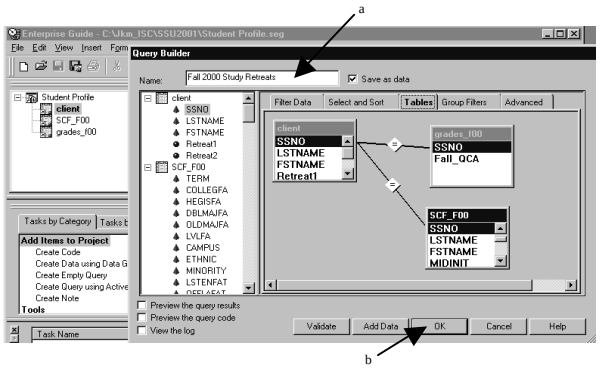


Figure 5.



ANALYSES

SAS Enterprise Guide was used two ways to assist academic advisors. One provided a profile of courses taken for the term and overall grade average. The second was used for comparison and prediction.

Profile

Faculty viewed a student's profile in Guide using the Student Term Profile project. The project uses grade extract data where each row of data represents a course that a student took during the current term. To create the profile the following was done:

- Selected Filter from the Data pull down menu (Figure 6)
- Filtered on a student's identification number (PIDM) or on name (Figure 7). I filtered on PIDM and renamed the query with the student's name. Please note that the identification numbers and names have been modified in the figures.
- Selected List Data under Descriptive in the Tasks box and chose variables (Figure 8)

Figure 6.

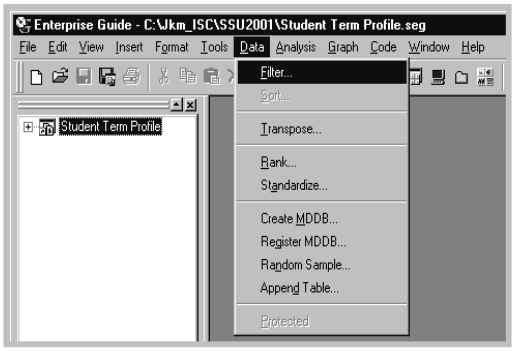


Figure 7.

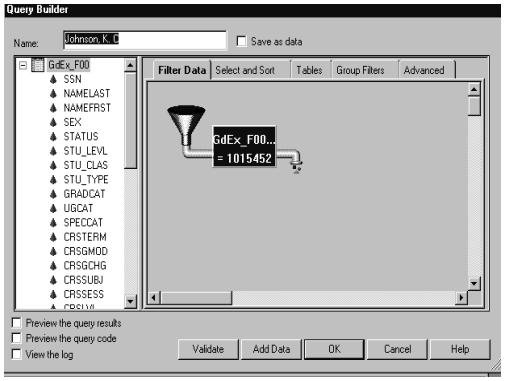
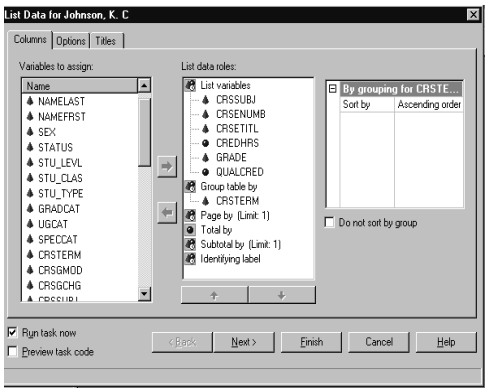
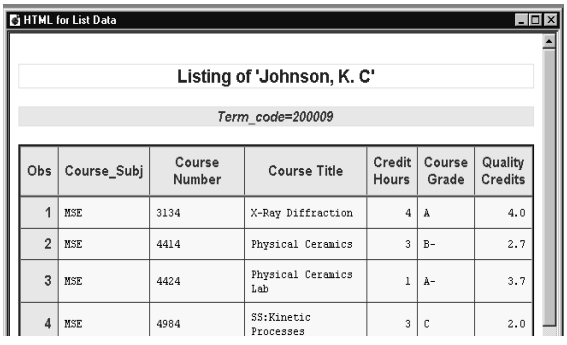


Figure 8.



An example of a profile is shown in Figure 9. An advisor can set up a filter for each advisee and resubmit the List Data or Summary during a consultation to view the current data. If the faculty member wants all of the advisees in one report, then instead of filtering the data and creating a task for each advisee, he/she may group by last name of student. It becomes a matter of personal preference. Since the advisor accesses the saved project and the data is updated, the advisor gets the most recent information for a student by re-running the task from the project window. In addition advisors can join Grade Extracts over several terms to maintain a complete course activity profile for their students.

Figure 9.



Obs	Course_Subj	Course Number	Course Title	Credit Hours	Course Grade	Quality Credits
1	MSE	3134	X-Ray Diffraction	4	A	4.0
2	MSE	4414	Physical Ceramics	3	B-	2.7
3	MSE	4424	Physical Ceramics Lab	1	A-	3.7
4	MSE	4984	SS:Kinetic Processes	3	C	2.0

Summary of Grades

Quality credits are the numeric equivalent of a letter grade. To calculate quality credit average (QCA = [∑ quality credits x credit hours]/ Total credit hours), I selected Summary under Descriptive in the Tasks box, chose quality credits for analysis variables, and chose credit hours as the relative weight variable (Figure 10). See Figure 11 for sample output.

Figure 10.

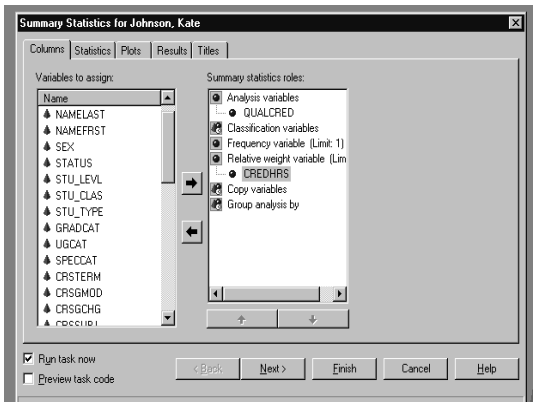
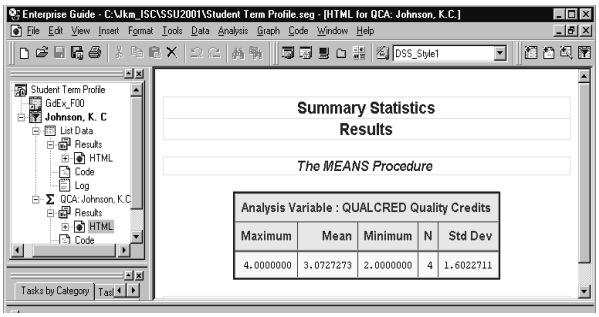


Figure 11.



Prediction

Not only is it valuable to be able to view a student's profile, but it is possible to evaluate the effectiveness of a new initiative with Enterprise Guide. In this study, the advisor wanted to know if attendance in the two offered study retreats had a beneficial effect on quality credit hours (QCA). Fall 2000 was the first time that the study retreats were offered. Students could attend both study retreats; most students chose to attend one of the sessions. The merged dataset [Fall 2000 Study Retreats] was the merge of Fall 2000 student census data with client study retreat data and Fall term grade and is listed in the Student Profile Project. The data was filtered to capture only those freshmen having major 'AZ' and having actually registered

for classes in the fall. Retreats 1 and 2 were coded 1 for attending and 0 for not attending. Correlations with scatterplots were run in Guide to check for outliers. The independent variables, Retreat1, Retreat2, high school grade point average, SAT verbal score, SAT math score, percentile rank in high school, sex (dummy coded: 0=Female, 1=Male), and honors student (dummy coded: 0=No, 1=Yes) were used to predict freshman fall QCA. Students missing data were excluded from the analysis.

The correlations of Retreat 1 and Retreat 2 with Fall QCA were not significant at the .05 level of significance. Results from the initial correlations indicated the correlation of those who attended Retreat 2 were slightly higher with Fall QCAs than the correlation with attending Retreat 1. As expected, high school grade point average had the highest correlation (.34) with freshmen Fall QCA. Although the regression model was significant ($p < .0001$), neither Retreat 1 nor Retreat 2 contributed significantly ($p < .05$) to the prediction of the Fall QCA of students in 'AZ'. The model explained only 21 percent of the variance of Fall QCA . Results from the regression may be viewed in Figures 13 and 14.

Figure 12.

Effect of Study Retreats--Fall 2000
Program = AZ

The REG Procedure
Model: Linear_Regression_Model
Dependent Variable: Fall_QCA QCA

Analysis of Variance

Source	DF	Sum of Squares	Mean Square	F Value	Pr > F
Model	9	124.61771	13.84641	32.32	<.0001
Error	1045	447.75446	0.42847		
Corrected Total	1054	572.37217			

Root MSE	0.65458	R-Square	0.2177
Dependent Mean	2.58498	Adj R-Sq	0.2110
Coeff Var	25.32242		

Figure 13.

Parameter Estimates

Variable	Label	DF	Parameter Estimate	Standard Error	t Value	Pr > |t|
Intercept	Intercept	1	-0.95571	0.29717	-3.22	0.0013
Retreat1		1	-0.00413	0.05161	-0.08	0.9362
Retreat2		1	-0.00540	0.05303	-0.10	0.9189
SATV	Verbal SAT Score	1	-0.00003309	0.00033075	-0.10	0.9203
SATM	Mathematical SAT Score	1	0.00225	0.00037206	6.05	<.0001
HSGPA	High School GPA	1	0.47555	0.06909	6.88	<.0001
PERCENT	Percentile High School Rank	1	0.38442	0.21904	1.76	0.0795
TRANHRS	Transfer Credit Hours	1	0.00446	0.00243	1.84	0.0668

CONCLUSION

The SAS version 8.2, coupled with Enterprise Guide 1.2, is a versatile tool. Clients can custom design their projects by renaming datasets and joining data from various sources. Someone very familiar with the structure of the Oracle tables that one is using should either create or assist in the development of tables/views. When data changes or is updated the project links to the updated data. Faculty advisors can merge departmental data to existing university data and create their own prediction models and make their own comparisons. One can save the source code, make modifications and run additional analyses.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Janice K. McBee
Decision Support Services
Virginia Tech
1700 Kraft Drive, Suite 2000
Blacksburg Virginia 24060
Work Phone: 540/231-3601
Fax: 540/231-2429
Email: jmcbee@vt.edu

Paper P508

Sending E-Mail From a Mainframe Using SAS® in an MVS Environment

Michelle Gillespie, Louisiana State University, Baton Rouge, LA
Doug Pacas, Louisiana State University, Baton Rouge, LA

ABSTRACT

Over 75,000 financial aid checks are disbursed to LSU students each year. In an effort to better serve the student body, a SAS program has been created to send e-mails that notify students when financial aid is posted to their accounts. This automated communication has significantly reduced the heavy volume of calls previously received by campus offices regarding the timing of incoming financial aid moneys. The simple process uses check detail information and e-mail addresses to create an output dataset which is transferred to an SMTP mail server on MVS.

INTRODUCTION

Businesses and universities are often faced with the need to inform large numbers of clients quickly and cost efficiently. On the LSU campus, over 75,000 financial aid and scholarship checks are disbursed each year through a mainframe system. The timing of these disbursements is dependent upon LSU first receiving money from the source that granted the award. Although the anticipated dates of these transactions are published, the actual posting dates may vary. Previously, large numbers of students phoned the university multiple times each semester to check on the status of their accounts. In an effort to better serve the student body and to reduce the work load for LSU employees, SAS is now used to send a personalized e-mail notification to each student when financial aid is posted to his or her account. Automated communication allows information to be disseminated efficiently, significantly reducing the number of students contacting the university by phone. This paper presents details of two SAS programs which can be used together to generate a mass e-mailing of personalized messages.

WHY USE SAS®?

SAS is a natural choice for addressing an e-mail generation problem such as the one just described. The data necessary for such a mass e-mailing (e.g., social security numbers (SSN), e-mail addresses, data specific to each individual) may be obtained from a variety of sources. The power of SAS to input, sort, merge and output this data makes it preferable to other mainframe programming languages.

The programs described in this paper were coded several months ago using SAS® version 6. There are new and exciting capabilities for a direct e-mail system interface coming in SAS® version 8. We look forward to exploring the new technology upon implementation of this newest version at our site.

THE CONCEPT

The concept behind the mass e-mailing of personalized messages is quite simple. SMTP (Simple Mail Transfer Protocol) is a started task in MVS which reads in a dataset of batched e-mails and consequently transmits each e-mail message to the appropriate address. The first SAS program described in this paper outputs a dataset of batched e-mail messages. Because it is possible to overload SMTP, a second program is used to divide this large dataset into multiple small datasets before sending the e-mails to SMTP.

PROGRAM I – CREATE A DATASET OF BATCHED E-MAIL MESSAGES

IDENTIFY THE POPULATION

As in any mass mailing, it is necessary first to identify the population to which the correspondence will be sent. In this application, a dataset of financial aid grants and loans that have been awarded to LSU students is read by the program. Multiple awards per individual may exist. Therefore, the SAS program sorts this dataset by social security number.

```
DATA SSNS;
   INFILE SENDTOS;
      @1       SSN            $CHAR9.
      @10      PGMCODE        $CHAR4.
      @14      PGMDESC        $CHAR30.
      @44      SEMESTER       $CHAR13.
      @57      CHKAMT         ZD7.2;

PROC SORT DATA = SSNS;
   BY SSN;
```

RETRIEVE THE E-MAIL ADDRESS

LSU maintains a DB2 table of current e-mail addresses for students, uniquely keyed by social security number. A DB2EXT procedure is executed in the SAS program to extract current e-mail addresses for all students. The extracted list is then sorted by social security number and a SAS MERGE is performed to associate each student's e-mail address with his or her financial aid data. This merged dataset will serve as the driver in generating the dataset of batched e-mail messages.

```
PROC DB2EXT OUT=PWSDATA;
   SELECT      DESKTOP_ID,
               PRIMARY_ACCESS_ID
     FROM      PWS.DESKTOP;
   RENAME      1=SSN
               2=ACCESSID;

PROC SORT DATA = PWSDATA;
   BY SSN;

DATA SSNSPWS;
MERGE PWSDATA (IN=A)
      SSNS    (IN=B);
   BY SSN;
   IF A AND B;
```

INPUT COMMON HEADER AND TRAILER DATA

The body of the message to be sent to each student in this process consists of three components. A common header paragraph informs the student that financial aid activity has been posted to his or her account. This is followed by personalized information regarding each financial aid award. A common trailer paragraph concludes the body providing information on when the student can expect to receive a mailed check. The header and trailer paragraphs are stored in separate datasets and are read into the SAS program, while the financial aid data specific to the individual is stored in the merged dataset previously described.

```
DATA HEADER;
   INFILE HEADER END=EOF;
   INPUT @1 MSGLINE $CHAR80.;
   NUMHDR + 1;
   IF EOF THEN

DATA TRAILER;
   INFILE TRAILER END=EOF;
   INPUT @1 MSGLINE $CHAR80.;
```

```
      CALL SYMPUT('LASTHDR',NUMHDR);
RUN;
```

```
   NUMTLR + 1;
   IF EOF THEN
      CALL SYMPUT('LASTTLR',NUMTLR);
RUN;
```

ASSEMBLE THE E-MAILS

The objective of the SAS program is to generate a dataset that can be fed to an SMTP mail server. Therefore, the records in this dataset must conform to the standards understood by SMTP within the MVS environment. Accepted format standards for sending SMTP e-mail have been published as "Request for Comments" numbered RFC821 and RFC822. Below is a sample of data records which would be output to a dataset by this SAS application and then consequently fed to SMTP.

```
HELO LSUMVS.SNCC.LSU.EDU
MAIL FROM:<BURSAR@LSUMVS.SNCC.LSU.EDU>
RCPT TO:<student1@lsu.edu>
DATA
Date: 26MAY2001 0:15:14 CDT
From: <BURSAR@LSUMVS.SNCC.LSU.EDU>
ReplyTo: <BURSAR@LLSU.EDU
To:<student1@lsu.edu>
Subject: Financial Aid
(blank line)
(body of letter goes here)
.
RSET
HELO LSUMVS.SNCC.LSU.EDU
MAIL FROM:<BURSAR@LSUMVS.SNCC.LSU.EDU>
RCPT TO:<student2@lsu.edu>
DATA
Date: 26MAY2001 0:15:14 CDT
From: <BURSAR@LSUMVS.SNCC.LSU.EDU>
ReplyTo: <BURSAR@LSU.EDU>
To:<student2@lsu.edu>
Subject: Financial Aid
(blank line)
(body of letter goes here)
.
QUIT
```

Each e-mail message may be thought of as a message body surrounded by an inner and outer envelope. The outer envelope contains information needed to accomplish transmission and delivery including the HELO, MAIL FROM, RCPT TO, DATA, RSET and QUIT commands as outlined by RFC821. The inner envelope portion should follow format standards specified by RFC822, and consists of the required header fields DATE, FROM and TO and several optional fields including REPLY TO and SUBJECT.

The SAS program generates a file of batched e-mail messages similar to those described above by reading and processing the newly created dataset of merged financial aid information and e-mail addresses. Since this dataset is sorted by SSN, the program uses FIRST.SSN and LAST.SSN to determine when the first and last records for a given SSN are being processed. When FIRST.SSN is true, all of the commands referred to in RFC821 and RFC822 that must precede the body of the message are output. The RCPT TO: and To: commands are followed by the e-mail address associated with that SSN from the merged dataset. A blank line is then output, followed by lines of data retrieved from the header input dataset. After the common header text, one line of student award detail per award is output. When LAST.SSN is true, the common trailer portion of the body is output, followed by a line of output containing a single period in column one. If an additional e-mail is to follow, a RSET line is written out to mark the end of the current mail transmission. RFC821 commands for the next e-mail then begin. Otherwise, a QUIT line is output which signals SMTP to terminate connection and close the transmission channel.

```
                    ELSE READLEN = THELEN;
   INPUT @1 LINE $VARYING.  READLEN;
   IF LINE = 'RSET'
   OR LINE = 'QUIT'
      THEN NUMRSET = NUMRSET + 1;
```

COMMENTS ON RFC821 AND RFC822

It is recommended that one review RFC821 and RFC822 in their entirety before writing SAS code to create the output file. However, a few specifics are worth mentioning.

- Commands, replies and host names are not case sensitive; however, mailbox user names may be case sensitive for some hosts.
- All commands and replies should be composed of characters from the ASCII character set.
- Although headers in the inner envelope must precede the message body, they are not required to appear in any particular order.
- If e-mails are being sent from a region that practices Daylight Saving Time, logic should be included to determine the proper time zone for the DATE field.
- A line containing only a period is output to indicate the end of the body. If for some reason it is desired to actually send a period in column one as part of the body, it is necessary to append a second period so that the period *within* the body will not be interpreted as the *end* of the body. The double period will be converted by the receiver host back to a single period.

PROGRAM II – DIVIDE THE E-MAIL DATASET FOR SMTP

WHY MULTIPLE DATASETS?

Theoretically, the dataset of batched e-mail messages generated by the Program I can be read and processed directly by the SMTP mail server. In reality, however, a single feed of a few hundred e-mails during peak times of the day can result in an overload. To avoid this problem, Program II is used to divide the large dataset into multiple smaller ones before being sent to SMTP. Although the SAS code in this program is more complex than in the one just described, the concept behind "dividing and conquering" the large dataset is quite simple.

DETERMINE THE OPTIMAL NUMBER OF E-MAILS PER SMALL BATCH AND THE TOTAL NUMBER OF SMALL BATCHES

How small are the smaller batches? The answer to this question may vary for each application and is dependent on internal parameters and available disk space at the time of execution. Using trial and error, we have determined that executing during late night hours and reducing the size of single feeds to 30 e-mails is sufficient to avoid overloading SMTP in our application. You may need to experiment to find an optimal batch size.

The large dataset of all batched e-mail messages is read in one line at a time. Using the RSET and QUIT commands as markers, the total number of e-mails is calculated. This total is then divided by the predetermined number of e-mails to be sent in a single batch. The result is rounded up to the next whole number and represents the total number of small batches needed.

```
%LET NUMBMAIL = 30;

DATA _NULL_;
   RETAIN NUMRSET   0;
   LENGTH LINE    $200;
   INFILE PASSONE LENGTH=THELEN END=LASTONE;
   INPUT @1 PEEK $CHAR1. @;
   IF THELEN > 200 THEN READLEN = 200;


   IF LASTONE THEN DO;
      NUMBAT = CEIL(NUMRSET / &NUMEMAIL);
   CALL SYMPUT('NUMBATCH',PUT(NUMBAT,5.));
   END;
RUN;
```

ALLOCATE FILE NAMES
A DO loop is utilized to dynamically allocate file names OUT1 through OUTnn, where nn is the total number of small batches needed to send the entire set of e-mails to SMTP. Each file name has a destination of SMTP and SYSOUT=A for immediate processing by the mail server.

```
%MACRO XXFNAMES;
   %DO I=1 %TO &NUMBATCH;
      FILENAME OUT&I PRINTER SYSOUT=A
         DEST=SMTP RECFM=FB %STR(;)
   %END;
%MEND XXFNAMES;
```

DIVIDE E-MAILS AMONG THE SMALLER BATCHES
The input dataset of all e-mail messages is read in a second time. A macro is used to determine the appropriate output file for each line of data. RSET and QUIT lines are again used as markers to keep track of which e-mail is being processed. The e-mail number is compared to the range of numbers to be included in each of the smaller datasets. For example, e-mails 1 through 30 are output to file OUT1, e-mails 31 through 60 are output to file OUT2, and so on.

As mentioned previously, RFC821 requires that batched e-mails be separated by a RSET command. The final e-mail being sent in a transmission must be concluded by a QUIT command to close the transmission channel. Each of the smaller datasets is sent as an independent batch to SMTP. For this reason, the RSET command following the final e-mail in each batch is converted to a QUIT. A 'trick' is used to ensure that the RSET commands being evaluated are in fact RFC821 commands and not part of the RFC822 message body.

```
%MACRO XXDOFILE;
   %DO I=1 %TO &NUMBATCH;
      %LET J = %EVAL((&I-1)*&NUMEMAIL);
      %LET K = %EVAL(&I*&NUMEMAIL);
      IF &J <= RSETNUM <=&K THEN
         FILE OUT&I NOPRINT NOTITLES %STR(;)
   %END;
%MEND XXDOFILE;

DATA _NULL_;
   RETAIN      RSETNUM        00
               NUMBAT         00
               RFC821         1;
   LENGTH      LINE           $200;

   INFILE EMAILIN END=LASTONE LENGTH=LINELEN;
   INPUT @1    PEEK    $CHAR1. @;
   IF LINELEN > 200 THEN THELEN = 200;
      ELSE THELEN = LINELEN;
   INPUT @1    LINE $VARYING.  THELEN;

   %XXDOFILE

   IF RFC821 THEN DO;
      IF LINE = 'QUIT' THEN DO;
         DORSET = 1
         LINE = 'RSET';
         END;
      IF LINE = 'RSET' THEN DO;
         RSETNUM = RSETNUM + 1;
         IF MOD(RSETNUM,&NUMEMAIL) = 0 THEN
            DOQUIT = 1;
         IF LASTONE THEN DOQUIT = 1;
         END;
      IF LINE = 'DATA' THEN RFC821 = 0;
      END;
   ELSE IF LINE = '.' THEN FRC821 = 1;

   IF DOQUIT THEN DO;
      NUMBAT + 1;
      PUT 'QUIT';
         END;
      ELSE IF DORSET THEN PUT 'RSET';
                     ELSE PUT _INFILE_;
   RUN;
```

RELEASE THE BATCHES TO SMTP
A final statement deallocates all of the dynamically allocated files allowing the messages to go to SMTP.

```
FILENAME _ALL_ CLEAR;
```

CONCLUSION
The basic functions of the two SAS programs described in this paper can easily be applied to other applications requiring the capability to send a large number of personalized e-mails. At LSU, a similar batch process has been developed to contact employees when supplemental checks have been directly deposited into their bank accounts. Another batch process sends e-mail notifications to individuals responsible for submitting procurement card accounting requests. In both of these cases, Program I was modified to create the master e-mail dataset. Program II was then used to divide the datset before sending the e-mails to SMTP.

The ultimate success or failure in developing such an e-mail application is dependent upon the ability of the program to access a stable, reliable source of current e-mail addresses for the population being contacted. Therefore, every effort should be made to ensure that this data is current. Due to the fact that individuals change e-mail addresses frequently and e-mail messages are sometimes retrieved on an irregular basis, it may be advisable to develop applications that send messages as a courtesy to clients, rather than messages of a critical nature.

REFERENCES
The two SAS programs presented in this paper can be viewed in their entirety by visiting the following web site:

http://www.lsu.edu/ocs/conference/ssu2001

ACKNOWLEDGMENTS
Program II and key elements of Program I were designed by Frank O'Quinn, Associate Director, Technology Support Center, Louisiana State University. He has also been an invaluable resource during the writing of this paper.

CONTACT INFORMATION
Your comments and questions are valued and encouraged. Contact the authors at:

Michelle Gillespie
Louisiana State University
Baton Rouge, LA 70803
Work Phone: (225) 578-4727
Email: mgilles@lsu.edu

Douglas Pacas
Louisiana State University
Baton Rouge, LA 70803
Work Phone: (225) 578-3718
Email: doug@lsu.edu

Using SAS to Create Presentation Quality Spreadsheets in Excel

By Joyce R. Hartley, Infineon Technologies – Richmond

Abstract

How often have you been asked to produce a report that has subtotals here, grand totals there, ratios two lines down, bold this label, change that font… in short, a report better created in Excel? But all your data are in SAS data sets? Using PC SAS, DDE (Dynamic Data Exchange), and a little bit of cleverness, it's now easier than ever to produce nicely formatted Excel spreadsheets that are populated with SAS data. This technique is particularly useful for the report that needs to be repeated for multiple units (i.e. departments), where each sheet needs to be customized. An intermediate skill level in both base SAS and Excel is recommended.

Background

Imagine a project that requires you to produce displays with fairly similar data, grouped by management unit or division, but with slightly different categories on each display. And you want your lines in a specific order, with subtotals and percentages interspersed throughout the display. DATA _NULL_ is a fairly good option, although it can be rather tedious to go through all the iterations, especially when it comes to handling missing data. For example, a university budget office might require analysis of data by "budget unit" or "school" categories. These categories will be similar in some ways—they would all have budget for salaries—but very dissimilar in other ways. A School of Medicine would have a rather different budget from Facilities Management, as would a School of Business from Academic Technology. Wouldn't it be nice to write a single program to handle all the various cases and be able to create a spreadsheet with special formatting (fonts, headers, etc.) as well?

Method

The first step is to create an Excel template that is later copied as a blank "form" to be completed by the inclusion of your data output from SAS. This template needs to include all special formatting, formulae, and links to the worksheet that will contain the SAS data. Once you've created the template, the next step is to get your SAS data in order, all in one dataset, in the final form to be output to the Excel file. Then the program will execute the command to open Excel, send the SAS data to the workbook, and save and close the workbook.

Create an Excel Template

The first step in this process is to create your Excel template. This file needs to include all possible rows and columns that would appear on any of the spreadsheets that you plan to populate. These will simply be hidden in the spreadsheets where they are not applicable. Leave all the data cells blank, but apply any desired formatting to those cells; the data that come from SAS will be unformatted. Also select any page breaks, if applicable. My sample template includes rows for resource data such as degrees conferred and headcount; these rows would be hidden for a budget unit such as Facilities. Because my spreadsheet includes subtotals, percentages, and ratios, these formulae were also entered in the template file.

Create a second sheet in your workbook; keep the default sheet name of Sheet2. This is the sheet that will contain the data that are output from SAS, but it will be hidden from the user. Decide in what row and column your data will begin. Plan to populate all the cells in a rectangular area to match corresponding area in your main spreadsheet. That is, if you have blank columns as spacers in your main sheet, allow for blank columns in your SAS

sheet. You will be linking this sheet to the main sheet, and it is much easier to do if you map large sections of one to the other. To do this, select the cells on Sheet2 you want to link, and Copy; go to the main sheet, right click on the cell where you want the data to appear; choose Paste Special; then choose Paste Link. Plan your rows and columns well, so that you can highlight whole areas of your data sheet, and paste the whole unit to the formatted sheet.

You can protect the workbooks you create so that no one can change the displays you've created, while still allowing SAS to update them. To do this, make Sheet2 the active sheet. Select Format > Sheet > Hide to make this sheet invisible to the viewer. Then select Tools > Protection > Protect Sheet to put password protection on your main sheet, and Tools > Protection > Protect Workbook to protect the workbook itself (so that no one can unhide Sheet2). SAS can still write to the hidden sheet, but without the password, changes can't be made in Excel.

Once you have finished all the testing and debugging, make a copy of the template for each spreadsheet you want to create.

Create and Output the Data

Your SAS data may be coming from several different datasets. The goal is to get all the data combined into one dataset, ordered in such a way that there is a single sort column and the variable names, regardless of what kind of data they contain, all reference the columns in which

Template Before SAS Data

Anywhere University
Template

	1994-95	1995-96	1996-97	1997-98	1998-99	1999-00	6-Yr Ave.
Headcount	0	0	0	0	0	0	0
Degrees Conferred	0	0	0	0	0	0	0
Annual Credit Hours	0	0	0	0	0	0	0
Expenditures by Ledger							
E&G	$0	$0	$0	$0	$0	$0	$0
Sponsored Programs	0	0	0	0	0	0	0
Facilities	0	0	0	0	0	0	0
Auxiliary Enterprises	0	0	0	0	0	0	0
Local Funds	0	0	0	0	0	0	0
State Funds	0	0	0	0	0	0	0
Aquarium Services	0	0	0	0	0	0	0
Grand Total	$0	$0	$0	$0	$0	$0	$0
Expenditures by Category							
Teaching/Research	$0	$0	$0	$0	$0	$0	$0
Graduate/Teaching Assistant	0	0	0	0	0	0	0
Adjunct Faculty	0	0	0	0	0	0	0
Other Personal Services	0	0	0	0	0	0	0
Total Personal Services	$0	$0	$0	$0	$0	$0	$0
Contracts	0	0	0	0	0	0	0
Travel	0	0	0	0	0	0	0
Equipment	0	0	0	0	0	0	0
Other Nonpersonal Services	0	0	0	0	0	0	0
Total Nonpersonal Services	$0	$0	$0	$0	$0	$0	$0
Grand Total	$0	$0	$0	$0	$0	$0	$0
E&G Expenditures per Headcount	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!
Total Expenditures per Headcount	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!
% of E&G Expenditures to Total	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!	#DIV/0!

they will be placed. In my example, my sort column is called “order”, and my column variable names are c95, c96, c97, c98, c99, and c00. Also include a dataset like this one when you combine all your data, to ensure that every row in your template Sheet 2 will get populated:

```
data dummy;
  do i=1 to 30;
    order=i;
  output;
  end;
run;
```

I try to match rows to rows in my two spreadsheets, that is, row 6 to row 6, which often leaves space in the first few rows for label variables that can be linked to the main sheet as well. A main title could be output to row 1, so that the titles don’t have to be typed into each individual spreadsheet.

Putting It All Together

The final step in the process is for your program to send the command to open Excel, create a filename to write the data to Sheet2, and use DATA _NULL_ to populate the sheet, and then close Excel. If you are doing multiple sheets, you can enclose part of the code within a macro, passing the different filenames with each macro call. Then you get to sit back and watch while SAS fills in all your spreadsheets for you.

In order to send the command to open Excel, you need to know what directory excel.exe is in. I do this by exiting to the DOS prompt, and poking around until I find the right directory. It is usually in something like c:\Program Files\Microsoft Office\Office\Excel, but you need to find the right 8-character name for the directory, i.e. c:\progra~1\micros~4\office\excel. The code to open Excel will look something like this:

```
options noxwait noxsync;
/* this opens Excel */
x c:\progra~1\micros~4\office\excel';
data _null_;
  x=sleep(2);
/* this pauses SAS to
     give Excel time to open */
run;
```

Before submitting this code, be sure that Excel is not already open.

Here’s the code to create the needed filenames and open the proper workbook:

```
filename cmds dde 'excel|system';
%let newfile=c:\my documents\my sas
files\v8\ssupaper\wkb2.xls;

/* create macro var with name of
   excel file, since you can't use
   macro vars directly w/in an OPEN
   statement w/DDE  */

data _null_;
  call symput
          ("exdata","""||'[OPEN("'||
          "&newfile"||'")]'||""");
run;

/* open the file */
data _null_;
  file cmds;
  put &exdata;
run;
```

Next, create a filename to reference sheet2 and specify the rows and columns that you will be writing to. In the example below, I’ve created an additional variable called sp (for space) that simply writes blanks to map to my blank columns in my template worksheet.

```
/* output data to the temp1data
   worksheet */
filename worksht dde
   'excel|Sheet2!r1c1:r30c15' notab;
data _null_;
  set temp1data;
  file worksht;
  sp=' ';
```

SAS Output formatted in Excel

Anywhere University
School of Marine Biology

	1994-95	1995-96	1996-97	1997-98	1998-99	1999-00	6-Yr Ave.
Headcount	123	118	120	124	129	132	124
Degrees Conferred	20	21	18	19	20	23	20
Annual Credit Hours	814	850	864	833	861	851	846
Expenditures by Ledger							
E&G	$2,415,339	$2,538,813	$2,242,864	$2,453,342	$2,864,638	$2,776,644	$2,548,606
Sponsored Programs	13,172	29,191	3,650	2,959	17,125	50,592	19,448
Facilities	28,352	13,773	185,578	41,207	80,701	88,151	72,960
Local Funds	435,104	597,610	783,491	490,266	356,959	475,495	523,154
State Funds	263,858	301,926	161,254	301,760	314,551	432,642	295,999
Aquarium Services	1,329	2,847	6,247	80,071	85,492	15,378	31,894
Grand Total	$3,157,153	$3,484,160	$3,383,084	$3,369,605	$3,719,466	$3,838,903	$3,492,062
Expenditures by Category							
Teaching/Research	$1,142,762	$1,084,864	$870,751	$889,878	$1,233,944	$1,188,039	$915,748
Graduate/Teaching Assistant	236,388	319,235	338,948	473,161	494,795	505,053	338,226
Adjunct Faculty	798,249	792,401	738,633	872,478	1,000,191	1,068,892	752,978
Other Personal Services	301,549	402,256	400,039	378,054	297,886	273,433	293,317
Total Personal Services	$2,478,948	$2,598,756	$2,348,371	$2,613,571	$3,026,815	$3,035,418	$2,300,268
Contracts	61,235	52,252	127,088	89,780	86,463	102,247	74,152
Travel	129,476	283,483	352,524	125,657	96,183	104,361	155,955
Equipment	487,108	518,231	559,395	545,505	510,409	596,877	459,646
Other Nonpersonal Services	386	31,438	(4,295)	(4,907)	(404)	0	3,174
Total Nonpersonal Services	$678,205	$885,404	$1,034,713	$756,035	$692,651	$803,485	$692,927
Grand Total	$3,157,153	$3,484,160	$3,383,084	$3,369,605	$3,719,466	$3,838,903	$2,993,196
E&G Expenditures per Headcount	$19,637	$21,515	$18,691	$19,785	$22,206	$21,035	$20,478
Total Expenditures per Headcount	$25,668	$29,527	$28,192	$27,174	$28,833	$29,083	$28,079
% of E&G Expenditures to Total	76.50%	72.87%	66.30%	72.81%	77.02%	72.33%	72.98%

```
  put order '09'x sp '09'x
          c95 '09'x sp '09'x
          c96 '09'x sp '09'x
          c97 '09'x sp '09'x
     c98 '09'x sp '09'x
          c99 '09'x sp '09'x
          c00 '09'x sp '09'x cavg ;
run;
```

And finally, close Excel:

```
/* close Excel */
data _null_;
  file cmds;
  put '[SAVE()]';
  put '[QUIT()]';
run;
```

Finishing Touches

Once all your workbooks are filled in, all you need to do is hide any rows that you choose to. You may find that some cells containing formulas come up with Excel errors, such as division by zero. In these cases, you can apply conditional formatting to these cells to replace the error with “N/A” or some other

indicator. It will probably take a few runs before you get your template file perfected, but once you do, you can copy it over for as many units as you need, and re-use it as often as you need. Nicely formatted monthly or weekly reports can be produced in minutes now, just by copying over the template file and changing the input and output filenames. The possibilities are endless.

Acknowledgments

Thanks to Mike Newsome for encouraging me to push my limits, and to Andre Walker, Jim Moyar, Bennie Fiol, Delores Anderson, Greg Vaeth and Michelle Vucci for all their help.

Contact Information

The author can be reached via e-mail at joyce.hartley@infineon.com, or by mail at
Joyce R. Hartley
Infineon Technologies Richmond
6000 Technology Boulevard
Sandston, VA 23150
(804)952-8262

V6 to V8 Applications: To Web or Not To Web?

Sharon Muha, SAS® Institute, Cary, NC
Elizabeth Malcom, SAS® Institute, Cary, NC

ABSTRACT

When upgrading a V6 SAS/AF® application to V8, what are the possibilities? Should the application be web-enabled? If not, why not and what are the other options? Should it make use of Java™ technology? What are the platform restrictions? What are the interface issues? Are there data-integrity concerns? These questions are just some of the issues that must be addressed when retooling applications. This paper steps you through the decision-making process and highlights decisions made by developers who were tasked with upgrading a SAS Publications Division V6 SAS/AF FRAME application to a web-enabled V8 application.

INTRODUCTION

When an application must be upgraded to use a new SAS software release, the process occasionally involves only a simple change to point to the executable file of the new release rather than the executable file of the old release. However, application upgrades are unfortunately rarely that simple. Over the life of the application, user requirements frequently change, as do the business problems that the application was originally created to solve. In such cases, upgrading to a new software release often provides an opportunity to re-examine the application to determine what enhancements can or should be made.

As developers in the Business Applications Department of the Publications Division at SAS, we were recently tasked with upgrading a Release 6.09 application to Version 8. When upgrading an application from Version 6 of SAS software to Version 8 of SAS software, one of the key decisions that we faced and that you will face is whether to web-enable the application. In the current business atmosphere, where all things web and wireless are hot, it is tempting to assume that your application should automatically go to the web. However, there may be compelling reasons not to make that jump.

To determine whether web-enabling your application is appropriate, we recommend first stepping back and examining the scope and requirements of your application. What are the user requirements? What constraints do the user requirements put on your system? What are the system requirements? Are there new applications that will need to interface with your existing application? If so, what platforms are these new applications on and should your application be on the same platform? Only after examining such issues can you make an educated decision about which implementation (stand-alone application, Java applet, JavaServer Pages™ technology, and so on) to pursue.

DEFINING USER REQUIREMENTS

Knowing who your users are and what your users need to accomplish by using your application is critical to successfully upgrading the application. Especially if your system has not been upgraded or enhanced recently, your user base may have shifted, and the problems that your users are trying to solve may have changed.

IDENTIFY THE USER BASE

Do you know who your users are? It may sound like a ridiculous question, but it can be easy to overlook segments of your user population. For example, sales staff may use your application to record sales data, but do executives, middle management, or project managers need reports from that data? If so, those people are also indirect users of your application, and you must be certain that your application allows for gathering the data that are needed for their reports. In short, as you prepare to gather user requirements, be sure that you have identified at least one representative from each significant group in your user population.

IDENTIFY THE BUSINESS PROBLEMS

Do you know what your problem is, or more accurately in this case, do you know what business problems your application addresses? Are those problems, in fact, the problems that still need to be addressed? As business models change, your user base may shift and the problems that your application was designed to address may become obsolete. For example, if your application tracks sales by region, but your sales regions have been redefined since the application was updated, you will need to redefine the regions for which data can be entered.

Ideally, the information that you are looking for when identifying the business problem is what your application can do to help your users do their jobs more effectively and efficiently. Keep in mind that different users may need different things from the same application.

IDENTIFY CURRENT APPLICATION DEFICIENCIES

Although we strive to create intuitive, bug-free, user-friendly applications, the truth is that there is some shortcoming in almost every application. Find out from your users if there are aspects of your current application that are problematic to them. For example, are your users entering data on one platform and then having to port that data to a different platform for reporting or for use in another application? Is there a button whose label makes no sense to them? Is there a required selection buried in a secondary window when it should be in a more obvious location that is easily accessible in the application? Is it obvious how to save data that have been entered and then exit the application or move to a different area in the application? It is obviously impossible to include here an exhaustive list of questions you could pose to your users about your application. However, if you simply ask, "What do you find difficult about this application?" most users will be more than happy to tell you.

DETERMINE THE NECESSARY LEVEL OF ENHANCEMENT

Once you have identified your users, determined what business problems your application is working to solve, and identified any current deficiencies in your application, you will be in a position to determine how much enhancing you need to do to your application as you upgrade to the new version of SAS software. If your application is meeting your users needs and addressing the problems they need it to, congratulations -- you are likely in the rare and fortunate position that you simply need to port your application to the new version with no significant changes. On the other hand, if there are new users or new problems that your application needs to address or if the usability of the application needs to be improved, yours is the far more common situation, and you can now embark on determining further system requirements before selecting which implementation option is most appropriate when you convert your application.

USER REQUIREMENTS CASE STUDY

As applications developers in the Business Applications Department of the Publications Division at SAS Institute, we were recently tasked with converting a Release 6.09 SAS/AF time-tracking application to Version 8.

Following our recommended user-requirements definition strategy, we examined our user population and determined that we had several distinct groups of users: writers and editors, project managers for individual documentation projects,

department and group managers of the various departments and groups internal to the Publications Division, and the vice president of the Publications Division. Additionally, while in the past only writers and editors were required to track their time using the application, it was decided that after the first of the year, all Publications Division employees would be tracking their time, which meant that we had the needs of additional departments to consider. We selected representatives from the various employee groups and questioned all of the managers to determine how they were using the application and what they needed from it.

As we tried to identify the business problems our application needed to solve, we determined that the writers, editors, and other employees, while they were required to use the application to enter data, rarely needed to report on the information that they entered. Project managers did not need to enter data (at least not in the context of being a project manager); however, they did want to be able to get reports on the data related to their projects. Group and department managers also needed reports related to their employees (overtime reports, time away from work, and so on). Finally, the vice president needed to be able to report on the data for business-reporting purposes (to provide IRS data and to determine cost of books, for example).

Our users had no trouble at all identifying deficiencies in the 6.09 application. The reporting system was sorely lacking to create adequate human-resources management reports and made no allowance for project-management reports. The interface that writers and editors used to enter data was antiquated by Version 8 standards and in many cases was non-intuitive or slowed down what should have been a fairly quick data-entry process. Finally, tracking codes needed to be created to support data entry by employees other than writers and editors.

After gathering our user requirements, it was clear that we needed to overhaul the time-tracking application in the process of upgrading it to Version 8, and we moved on to determining what our system requirements would be for the new version of the application.

DETERMINING SYSTEM REQUIREMENTS

Now that you have gathered your user requirements, your focus can shift to the application itself and to the system in which that application must function. You will likely find that your user requirements constrain how you implement your application. Additionally, you will need to consider issues such as platform availability, user location (anywhere from the same floor of a building to multiple international offices), and data integrity and security.

All the issues that must be addressed and the questions that must be asked to determine system requirements for any given system cannot realistically be included in this paper. However, keeping in mind that our ultimate goal is to convert an application from Version 6 to Version 8 and to determine whether the Version 8 application should be a web implementation, this section poses general questions in some of the major areas that you should address as you plan your conversion.

IDENTIFY PLATFORM ISSUES

Should your application continue to reside on its current platform? Have the user requirements indicated that there may be a need to change platforms? For example, you could move your mainframe application to a PC platform and, leaving the application data on the mainframe, use SAS/CONNECT® software to access the data.

Consider whether your application currently interacts with or will need to interact with other applications. Are the applications on the same platform, and if they are not, should they be? Because SAS software provides numerous tools for cross-platform communication, running related applications on different platforms is a realistic scenario; however, there may be additional considerations that make a different implementation a better choice.

IDENTIFY YOUR USERS

The location of your user population may greatly influence the implementation option you ultimately select for converting your application to Version 8. Are your users centrally located so that you have immediate access to the machines from which they will run your application? Will your application simply be installed on a network so that users throughout the office complex, city, state, country can run it from the network?

IDENTIFY DATA CONCERNS

The form, integrity, and security of the data created and accessed by your application are obviously key issues with respect to application development. As you consider upgrading, note whether your application is to be used for simultaneous data updates. Is there already code in place to ensure data integrity (i.e., record locking to ensure that one update does not incorrectly write over another)? Is access to some or all of your data restricted to a particular user or groups of users? Does your application access or create data that must be kept secure? For example, does your data contain financial information or account numbers that must be kept private?

IDENTIFY YOUR DEVELOPMENT STRENGTHS

As you consider your implementation options, also consider your development strengths. What are your areas of expertise? Can you write SCL code easily but have no experience with Java technology, or conversely, do you have Java experience but no experience creating SAS/AF applications?

If you have more than one implementation option and development time is critical, play to your strengths and use the implementation option with which you have the most experience. If you have several implementation options and you have enough time in your development cycle to absorb a learning curve, consider selecting an implementation strategy that will enable you to develop your skills in a new area.

SYSTEM REQUIREMENTS CASE STUDY

Once the user requirements for the Publications Division's updated time-tracking application had been determined, we turned our attention to defining system requirements. The application we were upgrading was originally a stand-alone application; however, as one of our first requirements, we knew that the new application would need to interface with a new, undeveloped system that would be used to track the life-cycle of Publications Division products. Although the new tracking system was not yet developed, we had to consider ways in which the new system and the time-tracking system would interact and model our data tables accordingly. Additionally, we were told that the application needed to be web-based. Ideally, your users will give you, the developer, the opportunity to determine based on all the available information what platform and implementation is most appropriate. However, in many cases, as in this case, you will simply be told which strategy must be used.

In examining our user base, we determined that our users were primarily local users and users from one regional office. However, we also noted that the users of the larger tracking application into which the time-tracking application would be integrated are located worldwide. Given that we knew that the two tracking applications would potentially share data, we felt that we needed to locate the data where the larger application (and the worldwide users of the larger application) could access that data.

With respect to data concerns, we did not have to concentrate greatly on this area. The applications that we develop run completely behind a firewall and, at least with respect to the time-tracking application that we were immediately concerned with, do not contain sensitive, private data. Our primary concern in this

area was being certain that records were not being updated simultaneously, but those concerns were alleviated by constructing the data tables to avoid this problem.

The strengths and areas of expertise in our development group were largely in the SAS/AF software area. None of us had significant Java experience though all of us were familiar with object-oriented programming principles.

Having determined the user and system requirements, we were now in a position to examine all of our data and determine an implementation strategy for the conversion of our application.

SELECTING AN IMPLEMENTATION OPTION

Now that you have identified your requirements, you can begin to tackle the big question: will your application be a stand-alone application or a web-enabled application?

Because the application we were tasked with converting is an internal application that runs completely behind a firewall and does not contain particularly sensitive data, we had the luxury of not having to address data and server security issues; therefore, because these issues are very much outside the realm of our experience, we do not make recommendations here for which implementation is best suited to deal with data and server security. We recommend that you discuss such concerns with your systems-support staff and if necessary with your SAS representative or a SAS consultant.

STAND-ALONE APPLICATION

Despite the fact that the web is hot and everyone seems to want to use it, there are some advantages to using a stand-alone application instead. If your application depends heavily on users navigating through it in a particular way, you may want to consider a stand-alone application. In a web-based application, there is no guarantee that your user will step through the application as you expect. For example, a user could simply shut down the browser (intentionally or accidentally) halfway through executing an application task. A user could return to a previous application web page without saving data from the current page. There are ways to exert some control over a wandering user; for example, you could use JavaScript to identify and respond to particular events. However, if you find yourself thinking that it would be great if a user could bring up a separate browser completely dedicated to your application, then consider going with a stand-alone application because users typically cannot be counted on to use web browsers that way.

Just as you avoid the wandering-user problem by implementing a stand-alone application, so too do you avoid dealing with data-caching issues. Generally speaking, you can count on knowing the settings of a stand-alone application window because you programmed them to act the way that they do. Knowing the settings, you can program the window to refresh after particular events. You cannot count on all users to have data-caching options set the same way on their browsers, and making sure that users are looking at the most up-to-date data on any given web page can be a challenge.

Another advantage is that it is easier to control simultaneous access to the same record in a stand-alone application. Record locking is difficult when implementing a web application but is relatively straightforward to manage in a stand-alone application.

With respect to user-interface considerations, stand-alone applications have the advantage of allowing you more control over the application window, for example, the size of the window and where it will appear when it comes up. Also, you can more easily create complex graphical user interfaces (GUIs) that change based on selections the user makes as he or she makes them. For example, a checkbox the user selects may change which items are available on a selection list in the same window.

If you are working in a PC networked environment, a considerable disadvantage to a stand-alone application is that either you have to have the application installed on each user's machine and then any time a patch is needed, everyone has to install the patch or upgrade individually, or you have to have an application server that is powerful enough to handle your number of users (which can be a challenge if you have a significant number of users and a complex application). If applications are installed locally, it is difficult to guarantee that all users have installed the most recent version. In the application-server scenario, in the event of an update, you can at least put an icon on the desktop that would point to the latest version of the application.

In a Unix environment, the issues seem to be more ease of maintenance and performance considerations. The farther away your users are from where your application is running, the slower the stand-alone application runs because data related to both the compiled catalog entries and the data sets need to be transmitted so the processing can be done at the client. One way around this problem is to install the application locally, which can require a heavy maintenance load. Another possibility is to surface the application to regional offices on the web via a Hydra application server. In this scenario only small amounts of data are transmitted, and all the real work is being done on an application server at your headquarters. This approach still requires a second installation of the catalogs on the Hydra application server, but maintenance is not quite so heavy as installing the application locally. The advantage to using a web implementation, specifically a JSP solution, in this situation is that theoretically there is only one copy of the application and the data to maintain and performance should be relatively the same regardless of the access location.

WEB IMPLEMENTATION

Hip, hot, here, and now - it's the web and your application can be on it. In addition to the "hipness" factor, though, it is true that web applications are generally viewed by users as being more easily accessible. They also have the distinct advantage that there is no need to install an application on everyone's desktop. Also, using tools provided with SAS software, you can access your application data on virtually any platform. For example, you could use the ROCF classes provided with webAF™ software to access your data, or you could use JDBC calls. So, even if your application data are stored on a mainframe, that in itself should not keep you from putting your application out on the web. An additional advantage to pursuing a web-based implementation is that you can use the functionality already built into the browser for basic searching capabilities and for common tasks such as printing.

A disadvantage to a web implementation is that if you are dealing with a complex GUI, it can be difficult or impossible to reproduce your stand-alone application interface in a web page. For example, if you need to vary the available selections in a list based on information that the user provides elsewhere in the GUI, you may not be able to do that in a single web page. In addition, you will need to address data-caching issues to be sure that pages are generated appropriately (or not generated at all) when a user attempts to revisit a page. Finally, you must determine where your web server is going to reside and make sure that all the necessary servers to run your application can communicate with each other where they are.

There are two obvious choices for a web implementation: Java applets and JavaServer Pages (JSP™) technology. Each option, as you might expect, has advantages and disadvantages.

JAVA APPLET

If you use a Java applet rather than JSP technology, you have more control over the GUI than you do if you use the available HTML entities. The Swing classes, in fact, enable you to create fairly elaborate interfaces.

A major disadvantage to pursuing an applet over a JSP solution is that you have to be sure when using an applet that everyone has the correct plug-in. Alternatively, if users do not install the expected plug-in, the rendering of your application GUI may vary widely relative to the browser configuration. Also, download time for applets can be significant. Will your users be patient enough to wait for the download?

Data processing is another area where there is some advantage to using a JSP solution rather than an applet because data processing in an applet is confined to the client. If you have to write back to the server using an applet, you will need to implement a SAS middleware technology solution, and while that is certainly possible, writing data can be much more straightforward with JSP technology.

JAVASERVER PAGES TECHNOLOGY

Using JSP technology, you can process your data on the server side, which allows you a very thin client. Also, you do not have to deal with download issues in the same way you do with applets. There can be an initial download hit the first time a user runs a JSP solution, but after that, the process speeds up considerably.

With respect to GUI development, because most developers are generally familiar with the existing HTML entities, the learning curve to implement a JSP GUI will likely be lower than the learning curve to create an applet GUI. However, your JSP GUI may not be as snazzy unless you are willing to branch out and implement some JavaScript or pull in a graphic designer to assist with livening things up with graphics (keeping in mind that graphics will affect load time).

IMPLEMENTATION OPTION CASE STUDY

Deciding which implementation to pursue was frankly the most difficult decision we dealt with in converting our application. We had been instructed to pursue a web-enabled solution, so that decision was easy. However, figuring out which web implementation to pursue on what platform, was a challenge.

Our current application data lived on Unix, but we had the option of moving it to an NT environment for the new time-tracking application. The future Publications tracking application that our time-tracking application will interact with will have data on numerous platforms, so the location of the data we might need to access in the future was not really a deciding factor when choosing our data platform. The current development platform in our department is primarily HP-UX reached via an emulator running on an NT workstation. There were indications that the department would move to the NT platform for future development. That being the case, we decided that our data would live on the NT platform.

The server issue, which servers we needed and what platforms they would live on, was another one that dogged us for longer than we would have wished. We determined that based on our number of users (about 150 occasional users and 10-15 constant users), either a Unix or an NT application server would be reasonable to meet our needs. With respect to web servers, we found that we were more familiar with NT web servers and could more easily run and test web servers on our NT workstations than on our Unix network. Additionally, we found that we could get an NT application server cheaper and faster, so we concluded that an all-NT solution would best suit our needs.

We ultimately chose to pursue a JSP implementation over a Java applet. Although in either case we had to learn how to program in Java, a JSP implementation seemed easier to understand, and we had more JSP sample programs available to use as models. We also wanted to avoid plug-in issues and hoped to capitalize on data-processing efficiency by taking advantage of the server-side processing that JSP technology provides.

CONCLUSION

Unfortunately, there simply is not a unique set of steps you can follow to convert your application from Version 6 to Version 8. The response "Well, that depends..." followed closely by yet another question will persist through most of your conversion-implementation analysis. However, patient and detailed examination of your current application and the goals of your new application will ultimately yield a workable implementation strategy that will in turn ultimately yield a V8 application best suited to the needs of your users.

ACKNOWLEDGMENTS

Thanks go to Leah Redfern for her contributions to this paper and to the project on which the paper was based.

CONTACT INFORMATION

If you have any comments or questions, please feel free to contact either author at:

Sharon Muha / Elizabeth Malcom
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
Phone: 919-677-8001
Fax: 919-677-4444
Email: sharon.muha@sas.com
liz.malcom@sas.com

Point and Click Web Pages with Design-Time Controls and SAS/IntrNet®

Vincent DelGobbo, SAS® Institute Inc., Cary, NC
John Leveille, iBiomatics LLC™, Cary, NC

ABSTRACT

SAS® Design-Time Controls (DTCs) are a powerful and exciting addition to SAS Web Technologies. These add-in software components integrate back-end SAS® software functionality with your desktop HTML editor. You can now create Web pages that contain SAS content in a WYSWIG, point-and-click fashion using any one of several popular HTML editors. In most cases, you do not have to write a single line of code.

INTRODUCTION

This paper provides a brief overview of the SAS/IntrNet Application Dispatcher. You do not need to know how to use this product in order to understand this paper. However, you need a working knowledge of the Application Dispatcher to use DTCs to generate Web pages.

This paper explains SAS Design-Time Controls and how you can use them to ease the task of building attractive Web pages. It also explains how to generate Web pages that use Java Server Page (JSP) and Active Server Page (ASP) technology.

SOFTWARE AND HARDWARE REQUIREMENTS

In order to use Design-time Controls, you must meet the following software and hardware requirements.

- You must license SAS/IntrNet software Version 6.09E, Version 6.12, Release 8.0 or later and have SAS installed and ready to run.
- You must license SAS/GRAPH® software if you wish to use Design-Time Controls that produce graphics.
- You must license SAS/EIS® software to use the MDDB Report control in Version 6 of the SAS System. If you are using Version 8, you must license either the SAS/EIS® software or SAS OLAP Server software. If you wish to build a SAS Multidimensional Database (MDDB) in Version 6 or 8, you must license SAS/MDDB® Server software.
- You must have a Web server in order to store and serve the Web pages you create. If you wish to use Design-Time Controls to create JSP-based or ASP-based Web pages, the Web server you choose must support these application types.
- You are not required to have the SAS System installed on your development machine in order to use SAS Design-Time Controls. However, SAS must be installed and configured to run as a SAS/IntrNet Application Server on some machine on your network. You must also have a Web server installed on some machine on your network.
- You must be running the Microsoft Windows operating system on the machine that you use to create the Web pages. Microsoft Internet Explorer Version 4.0 or later and one of the following HTML editors must be installed on this machine.
    - Microsoft FrontPage 98 or 2000
    - Softquad HoTMetaL Pro 5 or 6
    - Microsoft Visual InterDev
    - Macromedia Dreamweaver 2, 3, or 4
    - Macromedia Dreamweaver UltraDev
    - webAF™ 2.0 (part of AppDev Studio™ 2.0)
    - Macromedia Drumbeat 2000

  Note that the Microsoft Windows requirement applies only to the machine used to develop the Web pages. The machine that serves the pages and the machines that browse the pages can be any operating system.

APPLICATION DISPATCHER OVERVIEW

The Application Dispatcher is a component of SAS IntrNet software. The Dispatcher allows you to execute SAS programs from a Web browser. These programs can consist of any combination of DATA step, PROC, MACRO, or SCL code, allowing you to immediately leverage your SAS skills and deploy a truly thin-client Web application.

APPLICATION DISPATCHER ARCHITECTURE

The Application Dispatcher is comprised of three components, as shown in Figure 1:

- Thin Client
- SAS Application Broker
- SAS Application Server

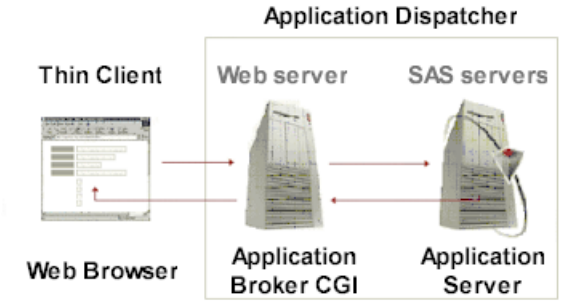


Figure 1. SAS Application Dispatcher Architecture and Execution Flow

The thin client is simply your Web browser. It is used to interact with any SAS programs that execute on the SAS Application Server. The user interface is typically an HTML page with a number of HTML form elements, and possibly some client-side script such as JavaScript. SAS software is not required on the client machine.

The SAS Application Broker is a very lightweight program that must be installed on your Web server machine. It is used to "broker" communication between the Web browser and the SAS Application Server using the Common Gateway Interface (CGI). You do not need CGI programming experience to use the Application Dispatcher.

The final component of the Application Dispatcher is the SAS Application Server. The server executes the SAS programs that you write and the results are returned to the Web browser via the Application Broker. Later you will see that Design-Time Controls alleviate this requirement to write your own SAS programs.

TYPICAL APPLICATION DISPATCHER EXECUTION FLOW

Figure 1 illustrates the typical execution flow of an Application Dispatcher request. As previously mentioned, the user interface is usually an HTML page containing HTML form elements. A user presented with this page will make appropriate choices and then submit the form for processing.

At this point, any data that are contained in the form are passed to the SAS Application Broker. The Application Broker determines which SAS Application Server is to handle the request and forwards all data to that server. The data are then converted to SAS macro variables and are also stored in an SCL list. Thus, the data contained on the HTML form are made available to your SAS program.

You should write your SAS program so that it uses the data passed from the Application Broker. After the SAS program finishes execution, you can display in your Web browser, the content that is generated by the program. HTML generated by the SAS Output Delivery System (ODS) or by the SAS HTML Formatting Tools are examples of appropriate content. A typical Application Dispatcher program is responsible for creating an entire page of content. This content is passed to the Application Broker, which in turn sends it back to the Web browser.

WHAT ARE DESIGN-TIME CONTROLS?

Design-Time Controls are add-in components that can be used with your Windows-based HTML editor. DTCs allow you to easily add SAS content to your Web pages. Using the interface of your HTML editor, you can insert a Design-Time Control into your Web page. The control assists you by presenting a dialog of choices and settings. After you make selections and close the dialog, the Design-Time Control will cause an Application Dispatcher program to execute, and the appropriate SAS-based content will be written into your Web page. Design-Time Controls act like page component wizards that help you build parts of your Web page.

After you publish your Web page anyone can browse the content that you have created. The Web browser user does not have to have Design-Time Controls installed on his/her machine in order to see the pages you have created, just as other people do not need your HTML editor to see pages you have created. Thus, the Windows restriction is imposed at design time, but not on clients that wish to view the page.

Design-Time Controls are very powerful. They can generate many forms of Web page content including HTML, JavaScript, Java applets, ActiveX controls, ASP, and JSP code. They present a user-friendly, intuitive interface that insulates you from much of the complexity that comes along with sophisticated Web content design. The SAS Design-Time Controls allow you to access the power of SAS Software, surface the results in your Web page, and still control the look and feel of your pages in a WYSIWYG editor environment.

Design-Time Controls contrast with Application Dispatcher programs in several ways.

- They generate only part of a page.
- You typically do not have to write the SAS program.
- The HTML editor handles HTML layout.
- They integrate with ASP and JSP.

As previously noted, typical Application Dispatcher programs generate a full page of content. Since the HTML editor is responsible for handling layout, the Design-Time Controls are solely responsible for generating fragments of SAS content. As will be discussed later, you can use Design-Time Controls to generate ASP- and JSP-based Web pages.

WHO USES SAS DESIGN-TIME CONTROLS?

SAS Design-Time Controls are intended to be used by someone who has knowledge of both the desired Web pages and the underlying data structures on which those pages are based. This may or may not be the same person in charge of the aesthetic design of the Web pages. It is possible for one person to lay out a series of Web pages including backgrounds, color schemes, and graphics, and a second person to use DTCs to insert the SAS content in the appropriate locations on these pages. Each Design-Time Control encapsulates its content, making it safe for anyone to edit the Web page without disturbing either the layout or the SAS content.

Of course, the page designer and SAS content builder may be the same person. In this case the SAS Design-Time Controls are still of considerable benefit. In the past, creating Web pages with a SAS server often meant having to write HTML and SAS code by hand. SAS Design-Time Controls make both tasks easier. Writing HTML is easy when you use a WYSIWYG HTML editor. Additionally, writing code is much simpler when you use the control property dialogs because they write it all for you.

When combined with the SAS/ACCESS® software products, the SAS Design-Time Controls have the added benefit of being able to reach from your Web page editor through SAS to external data sources, such as Oracle, DB2, and Excel. This means that a person in charge of creating Web-based reports for these other databases can also use SAS Design-Time Controls.

When your task is to analyze, construct reports, and surface that content to the Web, SAS Design-Time Controls make life easier for everyone involved.

HOW SAS DESIGN-TIME CONTROLS WORK WITH SAS/INTRNET SOFTWARE

You are not required to have the SAS System installed on your development machine in order to use SAS Design-Time Controls. However, SAS must be installed and configured to run as a SAS/IntrNet Application Server on some machine on your network. You must also have a Web server installed on some machine on your network. The Web server must be accessible from the development machine where you are creating the Web pages and from your users' machines where they will view the pages using Web browsers. The following diagram illustrates the required infrastructure to run Design-Time Controls with SAS/IntrNet.

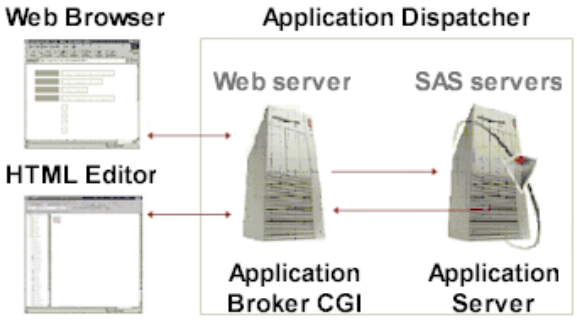


Figure 2. Design-Time Control and SAS/IntrNet Infrastructure

It is possible for the development, Web server, and SAS server machines to be three different machines, two different machines, or a single machine. It is also likely that the development machine (with the HTML editor) will function as a Web browser machine.

When you insert a SAS Design-Time Control into a Web page it reads the Windows registry and loads your default SAS server connection information. This information is set when the Design-Time Controls are installed on your machine. The control does not connect to the SAS server at this time. When you invoke the properties dialog for the control and make selections on the various tabs of the dialog, the control makes connections to the SAS server. The control queries the server for information such as a list of available data sets or a list of variables, and presents this information to you in the properties dialog.

A series of events occur when you save the Design-Time Control and when the Web page is browsed. The specific events that occur depend upon the current processing mode of the control. The processing mode can be set so that the control generates static content or dynamic content. Static content is content that does not change after construction of the page is complete. Conversely, a page that contains dynamic content, which is achieved through the use of ASP and JSP, will change if the underlying data changes.

If the control is set to generate static content, certain events occur, as shown in Table 1.

If the control's processing mode is set to static	
When the control is saved	**When the Web page is browsed**
• Control property settings are serialized into the Web page as an HTML comment. • Control connects to the SAS server and sends the current property settings. • SAS server runs a program stored on the server using the current control settings as input. • SAS server returns static content (usually HTML). • Control deposits content into the Web page	Static content is returned to the browser each time the page is viewed. No SAS processing is performed at this point.

Table 1. Static Processing Mode

If the control is set to generate dynamic content you have a choice of generating ASP code or JSP code. In either case, the events occur as shown in Table 2.

If the control's processing mode is set to dynamic	
When the control is saved	**When the Web page is browsed**
• Control property settings are serialized into the Web page as an HTML comment. • Control generates a URL string containing all the property settings. • Control wraps the URL string in ASP or JSP code and deposits the code into the Web page. No SAS processing is performed at this point.	• ASP or JSP code connects to SAS server and sends the current property settings. • SAS server runs a program stored on the server using the current control settings as input. • SAS server returns content (usually HTML) • ASP or JSP code deposits content into the Web page.

Table 2. Dynamic Processing Mode

When viewed in a Web browser, the resulting page should initially look identical for both the static and dynamic cases. However, over time the dynamic output may change if the underlying data changes. For static pages, the SAS processing is performed once, at the time when you construct the page. This is advantageous when your data is not changing because you only pay the processing cost once. It is not as beneficial if you need to see the latest data values and your data source is changing. For dynamic pages, the SAS processing is performed for each user that visits the page. This is advantageous because it shows up to the minute data values, which is very important for volatile data sources. However, this can become prohibitive if the processing takes a long time or if the page gets a lot of visitors. It is recommended that you consider the pros and cons of dynamic publishing before deploying your Web pages.

Sometimes you may find it necessary to blend the publishing modes of static and dynamic. For more information how to get the best of both techniques, see the "Advanced Topic: Scheduled Publishing" section of this paper.

AVAILABLE CONTROLS

Currently there are seven controls that you can use. Table 3 on the following page lists the controls and shows the minimum required release of the SAS/IntrNet Application Dispatcher software that is required for each control.

Control Name	Output Description	SAS Release or Version
SAS Critical Success Factor	Critical Success Factor Java Applet	6.09E, 6.12, or 8
SAS MDDB Report	Multidimensional database report in HTML form	6.12 or 8
SAS Stored Program	Output from your SAS/IntrNet program	6.09E, 6.12, or 8
SAS Table	HTML table from a SAS data set	6.09E, 6.12, or 8
SAS Tabular Report	HTML table from PROC TABULATE	8
SAS Thin Client Graphics	SAS graphic Java applet or ActiveX control	6.09E, 6.12, or 8
SAS TreeView	Java applet to visualize hierarchical data	8.2

Table 3. Available Design-Time Controls

These controls are described as follows:

- The SAS Critical Success Factor control is a point and click interface to the SAS RangeView HTML Generator (DS2CSF Macro).
- The SAS MDDB Report control is modeled after the Layout page of the SAS MDDB Report Viewer.
- The SAS Stored Program control is used to run any standard SAS/IntrNet Application Dispatcher program. It provides a way for you to integrate your existing or new program output into a Web page.
- The SAS Table control is a point and click interface to the SAS HTML Data Set Formatter.
- The SAS Tabular Report control surfaces some of the functionality of the SAS TABULATE procedure.
- The SAS Thin Client Graphics control is a point and click interface to the SAS Graph Applet HTML Generator.
- The SAS TreeView control is a point and click interface to the SAS TreeView HTML Generator.

BUILDING YOUR FIRST WEB PAGE

In this section you will learn how to use a SAS Design-Time Control to add some SAS-based content to your Web page. For the purpose of illustration, the SAS Table control will be used. You can adapt the steps outlined below to work with any of the Design-Time Controls.

Once you install the SAS Design-Time Controls on your development machine and set up the SAS Application Dispatcher, you are ready to incorporate SAS content into your Web pages. You use the functionality of your HTML editor to do the entire layout of your page. In most cases, you can create an attractive Web page without having to write any HTML code.

Start your HTML editor and spend a few minutes adjusting the layout and appearance of your page. You may want to change the background color or add some text and images to make the page more attractive. Once all the layout tasks are complete, you are ready to insert a SAS Design-Time Control into the page.

The act of inserting a Design-Time Control is different based on the HTML editor you are using. Typically, you can do this by finding the correct menu, such as **Insert**, and then selecting the proper submenu item, such as **Design-Time Control**. In Microsoft FrontPage, for example, the menu path is:

Insert ➨ Advanced ➨ Design-Time Control

Dreamweaver and Dreamweaver UltraDev users can insert Design-Time Controls through the **Insert** menu or via the SAS panel of the object palette. Other editors may have slightly different mechanisms for inserting controls. Please consult the documentation for your HTML editor if you are unsure how to insert an object.

Choose to insert a SAS Table DTC. At this point a thumbnail icon representing the Design-Time Control is placed into the page. Note that this is simply a placeholder to indicate that there is some piece of content at this point in your HTML page. You then must display the properties dialog of the Design-Time Control in order to make appropriate selections.

Some HTML editors may automatically display this dialog. Others will require you to invoke the dialog by double clicking the control, accessing a right mouse button menu, or some other technique. Access to the properties dialog in Dreamweaver and Dreamweaver UltraDev is provided through the property inspector for that object. If you are unable to access the properties dialog for the Design-Time Controls please consult the documentation for your HTML editor.

In Microsoft FrontPage, you can access the properties dialog either by double clicking the thumbnail icon or by selecting the icon and then choosing **Design-Time Control Properties…** from the popup menu. Figure 3 shows the **Dataset** tab of the SAS Table control properties dialog.

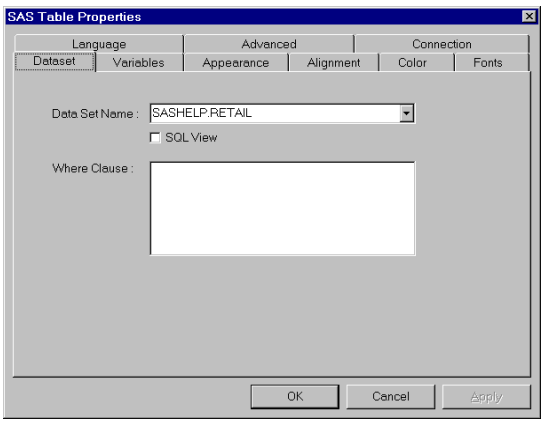


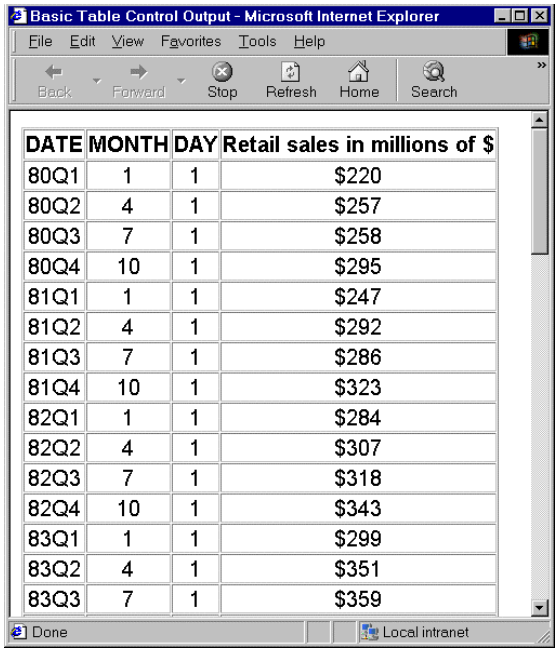
Figure 3. Dataset tab of the SAS Table control properties dialog

Select the data set SASHELP.RETAIL from the **Data Set Name** dropdown list.

You should now choose the variables that you wish to display in the HTML table. Go to the Variables tab and click the **Select** button next to the **Display variables** field. When the variable selector dialog opens, select the variables MONTH, DAY and SALES, in that order. Then click OK to close the Variable Selector dialog. Click the **Select** button next to the **ID variables** field. Select the variable DATE and click OK to close the variable

selector dialog. Type SALES in the **Sum variables** field to indicate that you want to sum the SALES column. Note that all of the fields on the **Variables** tab are text boxes. If you know the name of a variable you can type it in the box instead of opening the Variable Selector dialog.

Click OK to close the Table control properties dialog. At this point a SAS server program executes and returns an HTML table that contains the columns DATE, MONTH, DAY and SALES from the data set SASHELP.RETAIL. Save this HTML page and view it using your Web browser or using the preview window of your HTML editor. Figure 4 shows that the HTML table that was generated is pretty bland looking, so we will now set some other properties to make the table more attractive.



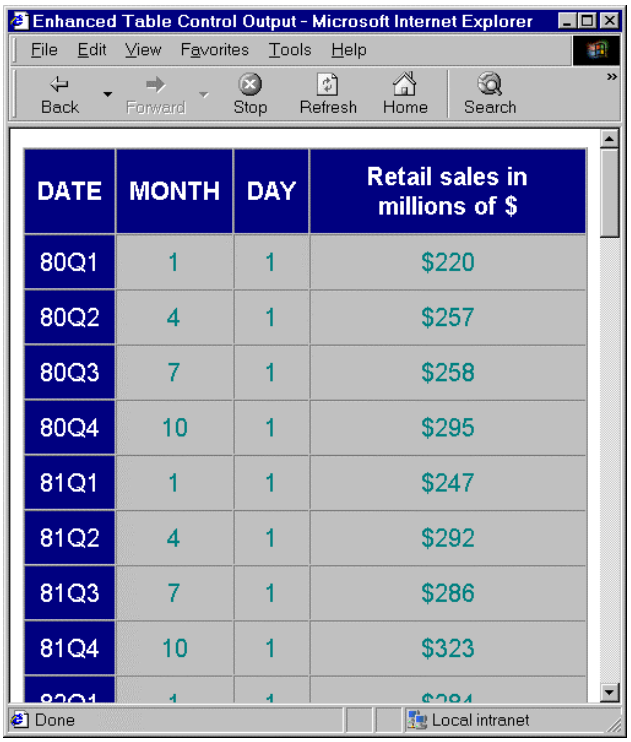
DATE	MONTH	DAY	Retail sales in millions of $
80Q1	1	1	$220
80Q2	4	1	$257
80Q3	7	1	$258
80Q4	10	1	$295
81Q1	1	1	$247
81Q2	4	1	$292
81Q3	7	1	$286
81Q4	10	1	$323
82Q1	1	1	$284
82Q2	4	1	$307
82Q3	7	1	$318
82Q4	10	1	$343
83Q1	1	1	$299
83Q2	4	1	$351
83Q3	7	1	$359

Figure 4. Basic HTML table generated by the SAS Table control

Access the properties dialog of the Table control and go to the **Appearance** tab. Supply a value of 10 for **Cell Padding** and a value of 0 for **Cell Spacing**. Go to the **Color** tab and supply the following colors for the various elements.

Element	Foreground Color	Background Color
Variables	Teal	Silver
Sum Variables	White	Red
Column Labels	White	Navy
ID Variables	White	Navy

Click OK to close the Table control properties dialog and save the HTML page. To see the enhanced table appearance (shown in Figure 5), view the page using your Web browser or using the preview window of your HTML editor.



DATE	MONTH	DAY	Retail sales in millions of $
80Q1	1	1	$220
80Q2	4	1	$257
80Q3	7	1	$258
80Q4	10	1	$295
81Q1	1	1	$247
81Q2	4	1	$292
81Q3	7	1	$286
81Q4	10	1	$323

Figure 5. Enhanced HTML table generated by the SAS Table control

STATIC PROCESSING VS. DYNAMIC PROCESSING

The SAS Design-Time Controls can operate in a few different processing modes. These processing modes produce either static output or dynamic output. If you examine the **Connection** tab in the properties dialog for each control you will see that the first field has the following options.

- Perform SAS processing once when building this page.
- Perform SAS processing when Java Server Page is invoked.
- Perform SAS processing when Active Server Page is invoked.

Note: The MDDB Report control does not have this field, but rather has a field named “Show MDDB Report.” Refer to the Design-Time Control documentation for information on this field.

The first option **“once when building this page”** is the static processing mode. If you select this option, the DTC connects to SAS when you save the control properties or when you save the Web page. A program runs inside the Application Server, and HTML is deposited into the Web page that you are editing. The output from the control is static when you use this mode. It will not change until the next time you modify the control or save the page (depending upon your HTML editor behavior).

Static processing mode may be appropriate for a lot of pages. This mode can be used to construct static content for HTML, ASP, or JSP pages. It also means that each time a user browses this page on your Web site you are not incurring the cost of running a SAS program. However, if you are using static processing mode, the Web page may not display the latest data. This may be just fine if you are displaying a table of last year’s revenue. Since that data will not change, it makes sense to use static processing mode. However, if you want to display a report based on some underlying data source that is rapidly changing then you probably want to consider a dynamic processing mode.

The second and third options, **"when Java Server Page is invoked"** and **"when Active Server Page is invoked,"** are dynamic processing modes. SAS Design-Time Controls use JSP and ASP in a similar manner. Therefore, consider statements about dynamic processing to apply equally to either ASP or JSP.

To use dynamic processing you must first make sure that your Web server is ASP or JSP enabled. Then, you must make sure that the file name for the Web page you are creating ends in .asp, if you intend to use ASP for dynamic processing, or .jsp if you intend to use JSP. Next, set the processing mode on the **Connection** tab to match the file name extension you are using.

Dynamic processing mode is powerful, but the differences from static mode can be subtle. The power of dynamic processing mode surfaces when you want to display the latest data in your Web page. When the Web server serves the dynamic page, the code generated by the SAS Design-Time Control will connect to SAS and run a program. This program will return HTML output representing the very latest data values. This is invaluable if you have a data source that is changing frequently. By using dynamic processing your Web page evolves and stays current without your intervention. From the perspective of editing the Web page content, dynamic seems almost exactly like static processing mode. However, there are some differences to consider. It is worth noting that you will probably be unable to use the preview feature of the HTML editor with dynamic processing mode. The best technique is to construct the page using static processing mode. Then, switch the processing mode on the control and change the file name extension when you are finished creating the page. When you are ready to view the Web pages, you must make sure to use an address that starts with "http" to see the result of the dynamic controls.

BUILDING A DYNAMIC WEB PAGE USING ASP OR JSP

The process of building dynamic Web pages is very similar to the process of creating static pages. The major difference is that for dynamic pages, you defer the execution of the Application Dispatcher program, and hence, generation of the SAS content, until you view the page in your browser. For the sake of brevity we will start with the static Web page that was constructed earlier, and shown in Figure 5.

Make a copy of the HTML file and open it in your HTML editor. Access the Design-Time Control properties dialog and go to the **Connection** tab. Note that the value specified in the **Perform SAS processing** dropdown list is **once when building this page**. This indicates static processing mode. To convert to a dynamic processing mode, select either **when Java Server Page is invoked** or **when Active Server Page is invoked**, depending on the type of dynamic page you wish to generate. Close the properties dialog and save the Web page.

Using your HTML editor, view the HTML code that was generated. You will see that the Design-Time Control did not create static HTML. Rather you will see VB Script or Java code, depending on the processing mode you chose. This code will run when a Web browser views the page. Thus, if the data that are in the SAS data set SASHELP.RETAIL change over time, the Web page will reflect that change when viewed in a Web browser. This is the difference between static and dynamic processing.

You must now give your Web page the correct file extension based on the dynamic processing mode that you chose. Use an extension of .asp for Active Server Pages and .jsp for Java Server Pages. You are now ready to view the page via your Web browser. Note that as mentioned earlier, you must use a Web browser to view dynamic pages, because the preview feature of your HTML editor will likely fail.

The result you see should be exactly the same as the page shown in Figure 5. Again, if the data had changed, then what you are viewing now would look different from the static page.

ADVANCED TOPIC: SCHEDULED PUBLISHING

If you take advantage of the dynamic publishing features of the SAS Design-Time controls, the ASP or JSP pages you create can be run on a scheduled basis. That is, the dynamic pages can be run periodically and the HTML that is generated can be stored as static pages on your Web server. This is useful if you have data that changes on a periodic basis, and you want to refresh the HTML pages that contain the time-sensitive content.

The remainder of this section discusses how you can accomplish this using the SAS System. Additionally, you will need scheduling software that is capable of running a program at a specified time. Examples include **cron** on Unix systems, **at** on Windows systems and **Control-M** on IBM mainframe systems. Alternatively, the SAS/Warehouse Administrator® software can be used to schedule your job.

Suppose you have data that is updated at 6:00 a.m. each day. You would like to generate reports that reflect the updated data and then store those reports on your Web server.

First, use the SAS Design-Time Controls to generate either ASP or JSP pages that represent your report, and store the pages on your Web server. Then, modify the following SAS program and schedule it to run at a specific time, for example, 6:10 a.m. This sample code uses the SAS software URL access method to retrieve the dynamic DTC page and store it as a static page on a Web server using the SAS software FTP access method.

```
*;
*  Provide the path to the dynamic DTC page
*  using the SAS URL access method.
*;
filename dtcpage url "http://myserver/path-
                      to-dynamic-dtc-page";
*;
*  Provide the name of the static HTML page
*  to be created, the domain of the Web
*  server machine where the static page is to
*  be stored, the user name a password to be
*  used to log into the Web server machine
*  and any other host commands using the SAS
*  URL access method.
*;
filename static ftp "path-to-static-page"
         host="myserver"
         user="userid" pass="password"
         rcmd="ascii"
         rcmd="site umask 002";
*;
*  Run the dynamic DTC page by accessing it
*  via the SAS URL access method. Then
*  publish the resulting HTML to the
*  Web server using the SAS FTP access
*  method.
*;
data _null_;
  infile dtcpage;
  file static;
  input;
  put _infile_;
run;
```

Note that if the SAS log contains the error message "ERROR: Service httpd not found" or "ERROR: Service ftp not found", please refer to the Troubleshooting section of the SAS Design-Time Controls documentation.

CONCLUSION

SAS Design-Time Controls are a powerful and easy way to add SAS-based content to your Web pages. They allow you to build attractive Web pages with a minimal amount of effort. SAS/ACCESS software also provides powerful data access capabilities in this easy-to-use environment. By supporting Active Server Pages and Java Server Pages, Design-Time Controls bring a new dimension to the types of applications you can develop with the SAS Application Dispatcher.

REFERENCES

SAS/IntrNet Web Site, SAS Institute Inc., Cary, NC. http://www.sas.com/rnd/web/intrnet/

SAS Application Dispatcher documentation, SAS Institute Inc., Cary, NC. http://www.sas.com/rnd/web/intrnet/dispatch.html

SAS HTML Formatting Tools Web Site, SAS Institute Inc., Cary, NC. http://www.sas.com/rnd/web/intrnet/format

SAS MDDB Report Viewer Web Site, SAS Institute Inc., Cary, NC. http://www.sas.com/rnd/web/intrnet/mddbapp.html

SAS/IntrNet Papers and Presentations, SAS Institute Inc., Cary, NC. http://www.sas.com/rnd/web/intrnet/papers/

SAS Design-Time Controls Web Site, SAS Institute Inc., Cary, NC. http://www.sas.com/rnd/web/dtc/

"SAS Design-Time Controls Now Available as Downloadable Component for SAS/IntrNet® Software," SAS Institute Inc., Cary, NC, 2000. http://www.sas.com/service/news/feature/03jul00/sas_dtc.html

"SAS Design-Time Controls Add Support for Macromedia Dreamweaver," SAS Institute Inc., Cary, NC, 2000. http://www.sas.com/service/news/feature/18dec00/dtcs2.html

"SAS Design Time Controls: New Tools for Information Delivery in the Enterprise," SAS Institute Inc., Cary, NC, 2000. http://www.sas.com/rnd/web/intrnet/papers/dtcDec00.pdf

“Microsoft Design-Time Controls Overview,” Microsoft Corp., Redmond, WA, 1997. http://msdn.microsoft.com/workshop/components/dtctrl/doc/overview.asp

CONTACT INFORMATION

Your comments and questions are valued and encouraged. If you have specific needs for controls and/or use an HTML editor or other application that does not support DTCs, let me know and I will evaluate your request as a future enhancement.

Contact the author at:

Vincent DelGobbo
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
Phone: (919) 677-8000
Fax: (919) 677-4444
Email: Vincent.DelGobbo@sas.com
Web: http://www.sas.com/rnd/web/dtc/

AppDev Studio® Release 2.0

Carl LaChapelle, SAS Institute Inc., Cary, NC

Presented By Scott Leslie

ABSTRACT

Version 2 of AppDev Studio is shipping! What is AppDev Studio you ask? AppDev Studio provides a single interface for the development of thin- and power-client business intelligence applications. It is the first suite of application development products tailored specifically for developing information delivery applications. It draws upon the proven strengths of SAS software, including an open architecture for developing solutions on every major Web standard on both the server and the client side.

This presentation will highlight many of the enhancements that have been made to this release of the software with live demonstrations of new features available in webAF® and webEIS™, two of the components that make up the AppDev Studio bundle.

A copy of the finished paper can be found under the Reference link on the AppDev Studio® Developer's Site (www.sas.com/rnd/appdev).

INTRODUCTION

With AppDev Studio you have everything you need to create Java client applications and applets; Java server applications (servlets, JavaServer Pages and Enterprise JavaBeans); CGI/HTML applications; Active Server Pages applications and traditional full-client applications.

The remainder of this paper will focus primarily on enhancements made to two of the Java-based technologies available with AppDev Studio: webAF and webEIS software.

webAF software is an integrated visual programming environment that enables you to rapidly build Java applications, applets, servlets and classes using a drag-and-drop object-oriented interface that helps reduce the amount of programming needed. webAF software helps you build applications that are easy to manage and that instantly connect to SAS software. New features found in V2 of webAF include:

Enhancements to the development environment

- Support for Java 2 and JFC/Swing technologies
- Support for drag-and-drop building of both AWT and Swing dialogs
- Improved packaging support including JAR signing
- A new Enterprise JavaBean (EJB) Wizard
- A pure Java implementation of SASNetCopy (called JSASNetCopy) that uses JDK 1.3 extensions mechanism for downloading required extension jars

Improved server-side Java development capabilities

- Improved support for servlet projects, including full support for the Servlet 2.2/JSP 1.1 specification
- Extended the ability for JSP and Servlet execution in the built-in Apache/Tomcat Web server, or any local Web server that you specify
- Provided new "WebAppDev" project area to build J2EE-compliant Web applications that can be easily deployed
- Added Visual Builder tool for JavaServer Page (JSP) projects
- Support for JSP tag libraries (also referred to as "custom tags")

New Java components and enhanced TransformationBeans

- New "Intelligent Page" (iPage) TransformationBeans for wireless/handheld device tasks that render an appropriate markup (WML, HDML, or HTML) based on the user request
- Enhanced MDTable TransformationBean for display of multidimensional data, including several new selector components enabling you to build complete OLAP applications
- New Chart TransformationBeans (both 2-D and multi-dimensional)
- New MenuBar TransformationBean for XHTHML/DHTML output
- Other enhanced TransformationBeans include the Table and TreeView
- New components for JDBC support
- Added components for IOM support

webEIS is an OLAP (Online Analytical Processing) application for the Web. An easy-to-use, drag-and-drop environment enables business analysts to build their own Web-based documents for sharing and viewing multidimensional data structures (MDDBs). OLAP documents can be published as either Java applets or using JSP technology without having to write a single line of Java code. The applets or JSPs are generated automatically and can be viewed with a familiar browser such as Microsoft Internet Explorer or Netscape.

Version 2 enhancements for webEIS® include:

- Improvements made to both the build-time and run-time pulldown menus, dialogs and toolbars to be more intuitive and to better follow GUI standards
- Run-time toolbars can be created via a drag-and-drop GUI from a full-featured suite of defined actions
- Charting improved to include independent specification of category and group variables, as well as formatted labels
- Formats can be applied to computed values
- Styles can be applied to totals
- NUNIQUE support
- Packaging process rewritten and simplified as "Save as Applet" and includes better support for images
- Documents can be deployed as JavaServer Pages using "Save as JSP…" option

This paper will discuss these and more of the features that Version 2 of AppDev Studio brings to help you build your enterprise applications.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Carl LaChapelle
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
Work Phone: (919) 531-7712
Email: Carl.LaChapelle@sas.com

A Modular Approach to Portable Programming

Michael A. Litzsinger, Quintiles Inc.
Lisa Kaye Brooks, Quintiles Inc.

*There is a balance between programming in a flexible **modular** manner **versus** a direct **standalone** manner. Segmenting programs into components aids flexibility but also introduces complexity. Creating standalone programs from redundant and repetitive components makes applying widespread changes tedious and time consuming. Consistency and version issues are raised as a program library grows.*

*This paper presents SAS® programming techniques that allow operating system and project assignments to be made centrally. **Components** generally include anything that can be best defined in a single location, such as LIBNAMEs, TITLEs, FOOTNOTEs, or system options. More advanced uses include single-source definition of macro variables in a conditional manner, allowing numbering or key elements to be controlled from one place.*

*This paper also details the use of **MFILE,** in conjunction with **MPRINT,** to direct executed program statements to a standalone output program. MFILE is a relatively unknown system option that was known as RESERVEDB1 in SAS 6.12. It is a simple way to port code using the output directing capabilities of the MPRINT system option. The resulting SAS program is stripped of all traces of modularity and underlying macros, and can be executed as a standalone program. The combination of these two techniques results in truly portable SAS programs while retaining the flexibility.*

Modular versus Standalone:

Modular programming is the breaking down of a program into parts that are of more manageable size and have a well-defined purpose. Modules should fit intuitively together into a system of components. In any development environment, including SAS, modular programming is how larger, more complicated programs are constructed. The first step is to break the task into its basic parts, which leads to defining intermediate steps, and ultimately devising a comprehensive and efficient solution. Modules can be developed individually, validated, and then used throughout an organization, promoting teamwork, efficiency, and innovation.

The advantages of modularity are obvious. Code only needs to be written once, which allows quick modifications. It provides a framework that dictates how subsequent programming should be incorporated. Modularity fits an environment where several programmers share work. The drawbacks include a slightly higher learning curve, a moderate level of complexity (even for simple tasks), and some required management of the macro libraries of SAS programs. An important issue is how to deliver simple portable programs to achieve contractual client obligations without providing proprietary SAS programs. Lastly, programs must adhere to corporate or departmental standards to maximize work sharing.

Standalone programs however are inherently portable. Consultants and service industry programmers create code that may be used across platforms or systems, and are often asked to provide executable SAS programs to satisfy documentation or audit trail requirements. Standalone programs are also the obvious method for starting a task from scratch. There is no reason to build in any unneeded complexity as the development unfolds. This works best for a single programmer, or in an environment where specifications do not change. De-macrotized standalone programs are the way to provide code without disclosing proprietary work.

The main drawback of standalone programming can be quite problematic. Without the planning intrinsically provided by modular programming, it can be difficult to quickly apply unexpected changes. In fact, the redundant nature of standalone programs can make applying modifications very tedious.

Presented on the next page is an example of a modular program and a standalone program side-by-side for easy comparison followed by a table that contrasts the advantages of each method:

Modular Program (demo_itt.sas):

```
*System options;
%include “settings.sas”;

*Define project libraries;
%include “libnames.sas”;

*Define output settings;
%include “titles.sas”;
%include “tnum.sas”;
%include “foots.sas”;

*Define macro library;
%include “age.sas”;
%include “freq.sas”;
%include “npct.sas”;
%include “maketbl.sas”;

*Main program;
data agedata;
  set datalib.demo;
  age=%age(birthdt,visitdt);
run;
%freq(indsn=agedata,colvar=pop,total=Y,
          npct=”000 (000%)”,out=freqout1);
%freq(indsn=agedata,depvar=age,
          colvar=pop,rowvar=sex,total=Y,
          npct=”000 (000%)”,out=freqout2);
data tbldata;
  set freqout1
       freqout2;
run;
%maketbl(indsn=tbldata,
               method=ProcReport,
               style=Standard3,source=Y)
```

Standalone Program (demo_itt.sas):

```
*System options;
options nocenter ls=150 ps=55 yearcutoff=1910;

*Define project libraries;
libname datalib “c:\datalib”;
libname library “c:\datalib”;

*Define output settings;
title1 “My Project”
title2 “Table 10.1.1 (Draft)”;
title3 “Demographics Summary”;
title4 “Intent-to-Treat Population”;
footnote1 “Note: Only randomized subjects
are presented.”;
footenote2 “Source: DEMO_ITT.SAS,
&sysdate9 &systime”;

*Main program;
data agedata;
  set datalib.demo;
  age= (year(visitdt)-year(birthdt))-
(month(visitdt)<=month(birthdt))+
(month(visitdt)=month(birthdt)&
day(visitdt)>=day (birthdt)));
run;
[..some code has been skipped .].
proc freq data=agedata;
tables pop/out=freqout1;
run;
proc freq data=agedata;
tables pop*sex/out=freqout2;
run;
[..some code has been skipped .].
data tbldata;
  if _n_=1 then set pct;
  set freqout1
       freqout2;
  array n n1-n3;
  array pct pct1-pct3;
  array val $ val1-val3;
  do val=1 to pop;
    val=put(n,3.)||” (“||put(pct,3.)||”%)”:
  end;
run;
proc report data=tbldata;
[..some code has been skipped .].
run;
```

Modular:	**Standalone:**
1. Best for medium and large solutions, whether static or dynamic. 2. Facilitates solving complex problems by breaking them down. 3. Global modification is made easy by only having to update one location. 4. Flexible structure encourages innovation. 5. Eliminates/reduces re-inventing the wheel. 6. Creates environment conducive to sharing workload and farming out programming tasks.	1. Best for small static solutions. 2. Large problems are not clearly defined and thus can be overwhelming. 3. Widespread changes are likely repetitive and tedious. 4. Reluctance to try new ideas because changes are difficult to implement. 5. Code is generated as many times as needed. 6. Only one person can work on a program at a given time.

Primary Components:

For this presentation, the modules have been grouped together into five primary functions (these are the first five listed below). A sixth module type, really a technique to automate elements of the other five modules, will be discussed briefly.

Modules discussed in this presentation:
1. System options, draft stamps
2. Library names and formats
3. Titles, numbering of tables, footnotes
4. Macros for data step manipulations and customized procedures
5. Main program body
6. (Advanced) Conditional execution of code based on macro variable (automatic or explicit)

System Options
Placing system options in their own programs facilitates making project-wide system changes. For example, during SAS Y2K remediation each project had to incorporate the appropriate cutoff year. It was only necessary to add the YEARCUTOFF option to the existing system option module.

Library Names and Formats
A single location for LIBNAMEs and FORMATs is a natural solution for portability. Projects can easily share programs with colleagues on different platforms or directory structures. Referencing unique LIBNAME modules makes it unnecessary to update individual programs.

Titles, Tables Numbers, and Footnotes
Placing project specific details in one location facilitates making quick changes. If an additional table is inserted, this technique makes it easy to renumber subsequent tables. A central location for table components allows code to be easily reused across similar projects. Differences can be clearly specified, aiding understanding as well.

Macros
The macro language can do some slick tricks in controlling program execution. But macros do not have to be complicated to be extremely useful. The primary use of macro language at all programming levels is the ability to use code repeatedly. Modularizing macros makes them more reusable with availability to all programs within a project or even an organization.

Main Program Body
This is the primary module. It contains code that is unique to the purpose of the program. It usually brings in data and manipulates and/or presents it. The main program brings together all of the sub-modules by calling or passing parameters to them.

Conditional Execution of Code
This applies mainly to the assignment of titles, table numbers and footnotes. Efficiencies can be gained by uniquely setting a macro variable within each of the main program bodies (for example: %LET PGMNAME=DEMO_ITT). Code can then be conditionally executed based on which program is accessing the module. This is a time saver when a global change has a domino effect, like the change of table numbers. A detailed example can be found later in this paper. The SAS system can automatically set program specific macro variables (SYSPARM), but this method is outside the scope of this paper.

Generating Code Using MFILE:

In SAS 6.12, "MFILE" debuted as system option RESERVEDB1. It became MFILE in SAS 7.

The MPRINT option is traditionally used as a technique in debugging or to display all executed statements to the log. MPRINT sends macro generated code to the log with the prefix MPRINT beginning each line:

```
MPRINT(PGMCODE):   libname datalib "c:\datalib";
MPRINT(PGMCODE):   data newdemo;
MPRINT(PGMCODE):   set datalib.demo;
MPRINT(PGMCODE):   dsvar=12;
MPRINT(PGMCODE):   run;
```

This code has all macro variables and references resolved and can be thought of as source or "compiled" code reduced to its simplest form. A great feature of MPRINT is that this code can be directed to an external file.

SAS program code is redirected to an output destination when:

- FILENAME MPRINT *path* is defined
- MPRINT and MFILE are in effect
- Code is executed through a macro

The required parts are bolded below. Using this technique, the macro facility saves every executed statement to a file for you.

Running this code:

```
filename mprint "c:\pgmlib\newdemo.pgm";
options mprint mfile;

%macro pgmcode;
   libname datalib "c:\datalib";
   %let var1=12;
   data newdemo;
      set datalib.demo;
      dsvar=&var1;
   run;
%mend pgmcode;
%pgmcode;
```

creates this executable code (newdemo.pgm):

```
libname datalib "c:\datalib";
data newdemo;
   set datalib.demo;
   dsvar=12;
run;
```

By now you may have realized an obstacle: SAS will automatically resolve *all* macro elements in the code it redirects to your saved file. You can't have it selectively pass code that contains desired macro elements, such as %include. Because our goal here is to create standalone code containing *some* modular elements (the ability to change system options and LIBNAMEs/FILENAMEs location), we have to be creative in how and when MFILE is used. Generally, we want to use it to create a standalone file of the proprietary elements, and then tweak this file so it can run on a new system. There are several ways to do this, and most methods have a "burden" of knowing the new operating system and location.

The best way to end the redirection of statements is by ending the macro. Other methods write an unwanted remnant to the redirected program:

- OPTIONS *NO*MFILE; or OPTIONS *NO*MPRINT; will become the last statement written to the output program.
- FILENAME MPRINT *path*; becomes the last statement written to the first output program. Then the second output program picks up where the previous one left off.

An important decision that must be made upfront is which type of comment text should be used in your source programs. Not all types of comments are passed to the standalone program:

Comment type	**Result**
*Comment;	Passed through as is: *Comment;
*Comment1 *Comment2;	Text will flow to long lines: *Comment1 *Comment2;
/*Comment*/	Ignored
%*Comment;	Ignored

Thus, only "*" comments are passed unchanged through to the MPRINT file. To avoid long comments on a single line, always end each line with a semicolon. Documentation blocks need to use "*;" comments to neatly pass them through to the standalone program. Comments in macros should not use "*" comments as these are not desired in the standalone program (proprietary elements should be kept transparent).

Outlined in the following table are solutions that make standalone programs created by MFILE more portable. They range from the simple "Text Replacement" method to the most portable "Append" method. The method of choice depends on the knowledge of the target operating system, the degree of desired portability, and the client's ability to alter and run SAS programs.

MFILE Methods to Create Portable Standalone Programs:

Method	Issues for provider	Issues for client	True Standalone?
Text replacement	Create upfront if OS/location known	Client must replace token text in each program (Operating system may have a text replacement utility to make this easier)	Yes
Portable Media	Define on portable media (Zip disk or diskette)	Client must leave work on media	Yes, restricted – must run directly on media (rules out CDs)
Comprehensive Program		Client must modify a single program in a directory	No – runs all programs in directory
Interactive Program		Client must set LIBNAMEs/ FILENAMEs once per session	No – requires interactive mode
Append	Create upfront if OS/location known	Client must run macro to create	Yes

Text Replacement Method:

The most straightforward method for creating portable standalone programs is to place a token string at the beginning of each source program. This token can then be replaced or swapped out prior to running the program on the new system once it is determined where datasets and included programs will reside. However, for this token text to be benignly passed through, it must be defined as a "*comment;". Remember, you cannot pass "%include …" through because it is resolved by the MPRINT facility. Other types of comments are not passed through at all.

Running this code:

```
filename mprint "c:\pgmlib\newdemo.pgm";
options mprint mfile;

%macro pgmcode;
   **** new libnames go here ****;
   %let var1=12;

   data newdemo;
      set saslib.demo;
      dsvar=&var1;
   run;
%mend pgmcode;
%pgmcode;
```

creates this code (newdemo.pgm):

```
**** new libnames go here ****;
data newdemo;
   set datalib.demo;
   dsvar=12;
run;
```

A swap or replacement in each program will need to be made.

Replace the text:

```
**** new libnames go here ****;
```

with the text:

```
%include c:\pgmlib\libnames.sas";
```

and define LIBNAMEs centrally in libnames.sas.

This process can be automated using operating system specific utilities, or each program can be modified individually. For example, a DOS com file or Visual Basic macro can make such changes automatically. And on an OpenVMS system, a SWAP com file can easily make large-scale replacements from one command.

Thus, if the target operating system and file locations can be determined in advance, the standalone programs can be readily created prior to shipment. Alternately, if the client is able to make the changes, they could determine the locations.

Portable Media Method:

Packaging all programs and data onto read/write media (zip drive, diskette) allows programs to be run in place. The programming environment is dependent on the media. There must be ample space available to write back to the disk when required (creating files, table output, SAS system writes, etc.). If LIBNAME and operating system definitions are known prior to delivery it is possible to prepare the programs ahead of time.

MFILE Methods, continued:

The resulting code is "ready to run" and would need no manipulation at run time. The programmer could develop the code directly onto the delivery media. Or it can be created in a testing area and then the text replacement or append method can be used to prepare the code for porting to the new media. This method is really only an option for smaller projects unless an external hard drive is delivered, or higher-density portable read/write storage becomes available.

Prologue to the Comprehensive Program and Append Methods:

The comprehensive and append methods require the program created by MFILE to be stripped of all operating system specific code (such as LIBNAMEs). A program is then run to attach code necessary for running on the new system.

The following example shows how you can modularize specific elements of the source program to make it more portable. Specifically, libnames.sas resides outside of the macro that redirects code to the standalone program. The *librefs* are available for running on the original system but are not included in the output standalone program.

Running this code (newdemo.sas):

```
*Code not directed to the output program;
%let pgmname=newdemo;
%include "c:\pgmlib\libnames.sas";

*set up program for mfile;
options mprint mfile ls=230 ps=80;
filename mprint "c:\pgmlib\&pgmname..pgm";
```

```
*Code directed to the output program;
%macro pgmcode;
%let var1=12;
  data newdemo;
     set datalib.demo;
     dsvar=&var1;
  run;
%mend pgmcode;
%pgmcode;
```

creates this standalone executable code (newdemo.pgm):

```
data newdemo;
   set datalib.demo;
   dsvar=12;
run;
```

Comprehensive Program Method:

For this method, a comprehensive program is built in addition to each individual program provided. The logical arrangement is to link similar tasks or group all the programs from a subdirectory. So if you had 20 summary table programs in a subdirectory, you would create another program that called the other 20 programs sequentially. If the operation system and location were known in advance, the comprehensive program could be built upfront. If not, the client user would have to fill in information for file locations (LIBNAMEs/ FILENAMEs) and the location of the included program library.

This method is not a true standalone because all programs are launched from one program. But the simplicity of this approach means that the LIBNAMEs only have to be defined once at the top of the comprehensive program. This approach may be clear enough to the client that they are willing to fill in the location information themselves.

For individual program execution, just comment out unwanted macro calls. For the same result, submit only a highlighted selection of code in an interactive session.

To run this code below, either the program and data locations need to be defined upfront or the client must provide this information in each comprehensive program:

```
%let srcpath=c:\pgmlib\;
%let datpath=c:\datalib\;

options nocenter ls=150 ps=55 yearcutoff=1910;

libname datalib "&datpath";

%macro runall(&pgmname=);
  proc printto file="&srcpath.&pgmname..tbl";
  run;
  %include "&srcpath.&pgname..pgm"
%mend runall;

%runall(newdemo);
%runall(neweff);
[… et cetera …]
```

MFILE Methods, continued:

An alternate to the comprehensive program solution is the **Interactive Program** solution, where LIBNAMEs define temporary work files, and individual programs merely call work files and are run subsequently. LIBNAMEs must define work files at the start of every session. The Interactive Program method isn't really a viable solution in a validated program environment, so an example is not provided, but it could be an option in some situations.

Append Method:

In the append method, a library name module is appended to each standalone program generated by MFILE. The following code (append.sas) attaches libnames.sas to newdemo.pgm to create the new program newdemo.sas.

The append.sas program makes use of put, input, and file statements to build the portable program that contains both the libnames.sas component and the standalone program created by MFILE. It is only necessary to update the macro variables DATPATH and PGMPATH in append.sas. The libnames.sas program uses these macro variables to set the *librefs*.

Run this program (append.sas):

```
%let pgmpath=c:\pgmlib;**target programs;
%let datpath=c:\datalib; **data directory;

%macro append(pgm=,srcpath=);
filename pgmout "&srcpath.&pgm..sas";
data _null_;
    file pgmout;
    put @1 "%" "let pgmname=&pgm;";
    put @1 "%" "let srcpath=&srcpath;";
    put @1 "%" "let datpath=&datpath;";
run;
data _null_;
    infile "&srcpath.libnames.sas" pad;
    input @1 line $256.;
    file pgmout mod;
    put line;
run;
data _null_;
    infile "&srcpath.&pgm..pgm" pad;
    input @1 line $256.;
    file pgmout mod;
    put line;
run;
%mend append;
%append(pgm=newdemo,srcpath=&pgmpath);
```

along with this code (c:\pgmlib\libnames.sas):

```
libname datalib "&datpath";
libname library "&datpath";
```

to create this executable code (newdemo.sas):

```
%let pgmname=newdemo;
%let srcpath=c:\datalib\;
%let datpath=c:\pgmlib\;
libname datalib "&datpath";
libname library "&datpath";
data newdemo;
   set datalib.demo;
   dsvar=12;
run;
```

Note that a single append.sas program can be constructed to process multiple programs. The append method is probably the best all-around solution. It can be set up to prompt for dataset and program locations in an intuitive manner.

Advanced Components:

One of the most helpful ways to automate modular programs is the use of single-source macro variables. Passing the program name in a macro variable can be used to conditionally select code to be executed in other modules. In the following example, table attributes are assigned based on the name of the program (on the next page):

Section of code (tnum.sas):

```
%macro tnum;

%if pgmname=DEMO_ITT %then %do;
  %let tnum=10.1.1;
  %let tbltitle=Demographics Summary;
%end;
%else if pgmname=DEMO_SAF %then %do;
   %let tnum=10.1.2;
   %let tbltitle=Demographics Summary;
%end;
[… more definitions have been skipped …]

%if %index(&pgmname,_ITT)>0 %then %let
   pop=Intent-to-Treat Population;
%else %if %index(&pgmname._SAF)>0
   %then %let pop=Safety Population;

%mend tnum;

title2 "&tnum";
title3 "&tbltitle";
title4 "&pop";
```

Advanced Components, continued:

Running this section of code (demo_itt.sas):

```
%let pgmname=DEMO_ITT;
%include tnum.sas;
%tnum;
```

The resulting macro variables assigned:

```
%let tnum=10.1.1;
%let tbltitle=Demographics Summary;
%let pop=Intent-to-Treat Population;
```

The resolved code:

```
title2 “Table 10.1.1”;
title3 “Demographics Summary”;
title4 “Intent-to-Treat Population”;
```

Whenever you want to plug in the values 10.1.1, Demographics Summary, or Intent-to-Treat Population, you need only use the tokens &tnum, &tbltitle, or &pop respectively.

Lastly, such conventions make it much easier to rename the delivered programs to number-based names. During development, it is preferred to have function-based program names (program demo_itt produces the Demographic Summary for the ITT population). However, the client may prefer to receive number-based program names (program t10_1_1 produces table 10.1.1) or even both (program t10_1_1_demo_itt produces Table 10.1.1 - Demographic Summary for the ITT population).

Conclusion:

The optimal solution combines the flexibility of modular programming and the straightforwardness of standalone programming. This solution must provide comprehensive code that still allows proprietary work to be preserved. Providing system options and LIBNAME project declarations in a modular format allows true portability to another platform/system. Other components are then delivered as “compiled” non-proprietary code, void of all macros, so that the entire set of programs serves as documentation. This permits easy replication of work. This approach is the most intuitive and is fairly straightforward.

An alternative is to also provide the project specific details (table numbers, titles, and footnotes) to the client in a modular format. This would allow renumbering and other cosmetic modifications to be made by the client, but this depends on the purpose of the ported code and the roles of those involved. This risks overwhelming the client. The system option and LIBNAME components also can be largely pre-built should that information be determined as a project is being set up. Changing the degree of modularity provided in the portable programs "after the fact" is significantly more time-consuming.

Version Note:

“MFILE” first appeared as a system option named RESERVEDB1 in SAS version 6.12. Beginning with SAS version 7 this system option was renamed MFILE. The functionality remained the same. Companies running both 6.12 as well as a more recent version will need to remember this and to change the system option name when moving programs across versions.

Contact Information:

Michael A. Litzsinger
Lisa Kaye Brooks
Quintiles, Inc.
Statistical Programming
P.O. Box 13979
Research Triangle Park, NC 27709-3979
(919) 998-2888
mike.litzsinger@quintiles.com
lisa.brooks@quintiles.com

OOP Needs OOA and OOD

Andrew Ratcliffe, Ratcliffe Technical Services Limited

Abstract

So SAS/AF® supports object-oriented programming (OOP), but does that mean that all applications developed using SAS/AF® are object-oriented? Not so. This paper provides an outline of an object-oriented approach to producing true object-oriented applications. The paper emphasises a focus on objects, not processes, combined with proper object-oriented analysis (OOA) and design (OOD).

Techniques and practices covered in more detail include the use of CRC (Class, Responsibility, and Collaboration) cards, the UML (Unified Modelling Language), software modelling, and an iterative approach to OO projects.

Introduction

Since reading Grady Booch[1] in the early 90's, I have been convinced of the merits of the OO approach. I have also been convinced that it takes a lot of work for those like me, brought-up with a procedural and modular training, to learn and truly understand the new approach.

The introduction of Frame entries to SAS/AF® at around the same time was accompanied by a large amount of talk about SAS software applications being object-oriented. The truth was that a) the programmer's interface with the SAS system had gained a large amount of object orientation, and b) programmers now had the **opportunity** to write object-oriented applications. However, not all programmers took that opportunity.

To this day, the majority of my new clients are not producing object-oriented applications. Many of them believe that they are, but a five-minute chat about the classes that they have in their applications soon convinces them otherwise.

The objects in object-oriented applications are represented by classes. Objects are items in the business-world that have attributes, states, and behaviour. Examples of objects include accounts, patients, users, and production lines. Actions like run, load, and measure are not objects, they are behaviours of one or more objects. I often get questioned "but running has attributes such as speed, so surely it's an object." Think carefully: the attribute of speed more accurately pertains to the object that is running. Running is behaviour, and speed is an attribute of the thing that is running.

With this brief paper I aim to guide the reader towards a clearer understanding of the object-oriented approach. I will do this with brief descriptions of some key areas of interest and with copious pointers to further reading.

Objects and Processes

In some ways, the OO approach is an evolution of the modular school. Before I get flamed by groups of OO enthusiasts, let me explain. One of the key concepts of modularity is that of breaking the problem down into pieces, defining the interfaces between the pieces, and then solving the individual problems without a great deal of cross-reference between the individual pieces and their solutions. It's called loose-coupling.

Loose-coupling is a key concept of OO too, but the loose-coupling is combined with other concepts such as abstraction, polymorphism, and hierarchies.

Using the traditional modular approach, data would be analysed almost independently of the tasks (processes) to be performed upon it by the application. OO bundles data and processes together into objects. Thus, individual items of data and

related processes are tightly-coupled, whilst different data items are loosely-coupled.

One of the key tasks in building an OO application is finding the right classes in your analysis phase. This is distinct from the traditional programming methods where one would be looking for algorithms. A class should provide a representation of something in the vocabulary of the problem domain (or solution domain). It should be simple (small) yet flexible (and extendable). It should provide a good abstraction whilst hiding its implementation. Each class should have a clearly identifiable responsibility to fulfil, just like each person in a team.

In your search for candidate classes, look for nouns not verbs. Interview users and sweep any documentation you can find (requirements documents and sample reports, for example).

Things such as 'run,' 'load,' and 'validate' are typically not good classes. The fact that these words are verbs is a major clue. In these cases the more appropriate classes are likely to be things such as a database or a transaction.

When implementing your classes, keep the principal of abstraction in your mind. The consumer of your class (the programmer who calls your methods) should be ignorant of how your class is implemented. A clear example is that your consumer should not need to know whether you're storing your data in a data set, an SLIST, or an external file. Your consumer should focus on what your class delivers, not how it is delivered, and your consumer should **never** access your class's data directly.

For further reading on the subject of good classes and responsibility-driven design, see Booch[1] or Rebecca Wirfs-Brock[2].

CRC Cards

As I said, the majority of SAS software application developers have adopted a procedural approach to their work. This has come about from a number of reasons, not least being that the DATA step and PROC approach is very sequential and procedural in nature. To change one's entire mind-set on application design can be very challenging. Class, Responsibility, and Collaborations (CRC) cards provide an excellent, natural means of adjusting that mind-set.

First introduced by Kent Beck and Ward Cunningham in 1989, CRC cards are easy to use and have additional benefits arising from their interactive nature. In simple terms, the idea is that you create a 4" x 6" index-card for each proposed class in your application. On each card, you identify the name of the class, its responsibilities, and the classes with which it collaborates to fulfil its responsibilities. Then you run through scenarios of activities that the application must perform ('use cases' in UML terminology).

In running-through the scenarios, participants in the exercise take hold of the cards and act-out the responsibilities of their given classes. The cards act as a central part of an iterative exercise: running a scenario, analysing the outcome, and adjusting the classes and responsibilities.

Responsibilities are high-level descriptions of the purpose of the class. The size of the card prevents the creation of over-loaded classes, i.e. those that have too many responsibilities.

The simplicity of the CRC card concept (and its practical implementation in index cards) results in a concentration on the analysis and design rather than the implementation. It is ideal! Participants focus upon the objects in the system and are divorced from platform-dependence and language-dependence. By physically holding the cards and acting-out their classes' responsibilities, participants are forced to identify with their classes, to see the design from the perspectives of their classes.

The cards form a communication medium that is acceptable to both business-focused and technically focused participants. I find them to be an excellent facilitator when trying to get users, designers, and developers to sit around one table.

The physical interaction between participants is unavoidable and desirable. It contributes a) to the understanding of the design, and b) to the effectiveness of the team both during and after the exercise. CRC card activities are good team-building exercises.

In addition to using CRC cards as part of analysis and design phases, I have found them to be tremendously useful as training tools for those who are new to OO concepts. Students are taken away from all that they have learnt about traditional techniques and take to the class-focused approach very quickly.

For further reading on the subject of CRC cards, see Beck[3] and David Bellin[4]. Wirfs-Brock's[2] approach is responsibility-driven and supports the use of CRC cards.

QuickCRC is a software product produced by Excel Software. It permits CRC cards to be created and stored on MS-Windows and Macintosh. Further, it permits scenarios to be created, stored, and acted-out. Information is available at http://www.excelsoftware.com. Note, however, one of the big benefits of the CRC card approach is its interactivity. This benefit is lost with the use of computer software. As a means of storing the outcome of a CRC exercise, QuickCRC does have benefits.

Methods of Control

The interactions that occur between objects should also be the subject of your design thoughts. There are several different patterns of collaboration and control that might be used.

All objects are not created equal. Some are tasked with cajoling others into doing detailed work, in a pattern similar to the manager-staff relationship most of us work with in our own lives. Objects might be classified as controllers, information holders, or interfaces. Those objects that are more active (such as controllers) warrant more of our attention.

Methods of control range from 'uncontrolled' through to 'centralised' and 'de-centralised.' Applications that use an uncontrolled style usually result from lack of design. Thus, control and co-ordination is performed in many different parts of the application without any structure to it.

I have found that novice OO designers tend toward the centralised approach. In this approach, one (or a small number) of the objects takes a large amount of responsibility for interfacing with other objects. The central object acts like a traditional procedural solution, "running" one object after another in order to run through a sequence of tasks.

The centralised approach tends to result in a complex controller object that is difficult to maintain because of its complex code. In addition, it can be difficult to assign work to teams of developers when the bulk of the application's logic is contained in one single class.

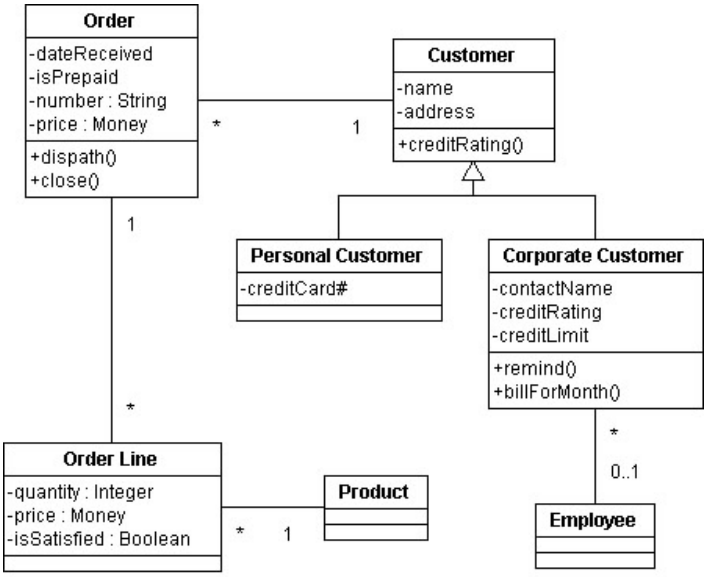
I much prefer the de-centralised method of control. Applications designed using this style tend to have clusters of objects. Each cluster has a clearly defined purpose and role within the application. Each object in a cluster co-operates and interacts with its partners rather than being centrally controlled. Similarly, a sub-set of classes in each cluster will co-operate and interact with classes in other clusters.

On the down-side, the de-centralised approach can make it harder for a maintenance programmer to follow the flow of events within the application. But, on the plus-side, responsibilities are spread more evenly (and to a shallower depth) amongst the application's classes. Each and every class is easier to maintain and enhance because of this.

For further reading on the subject of control styles, see Wirfs-Brock[5].

The UML

Drawing diagrams to describe your OO application needs something more than simple flow-charts. The UML (Unified Modelling Language) is the definitive notation for describing OO applications.

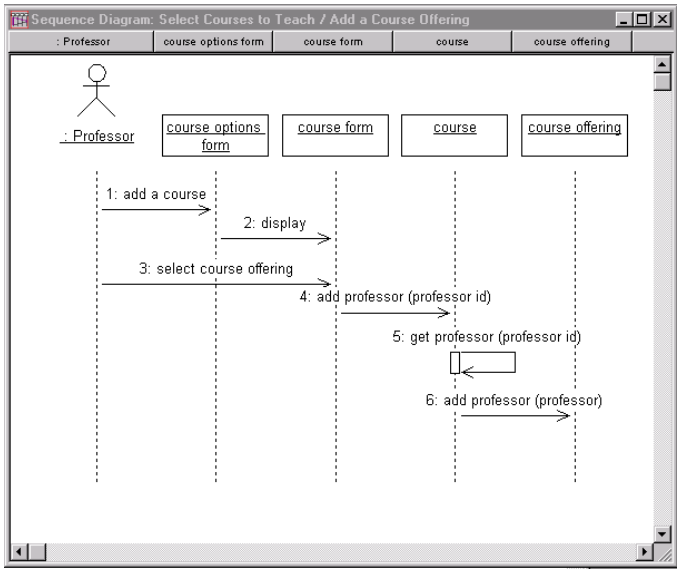


The UML is just a notation - a series of symbols and diagrams. The set of symbols (and associated semantics) that comprises the UML is designed to optimise the information conveyed by any diagram that you should draw. The UML defines seven types of diagrams: use case, class, package, sequence, collaboration, state, activity, and deployment. The scope of the diagrams runs all of the way from high-level analysis down to individual class's methods and attributes (instance variables) and the ways in which they interact.

The OMG (Object Management Group) are the independent standards group for all things object-oriented. They have accepted the UML as the standard modelling language. Thus, diagrams drawn using the UML notation will be more easily

understood by your peers than those drawn using any other notation or style.

The class diagram is the one that most people instantly associate with. It represents the static



relationships between the classes: association ("has a") and subtypes ("is a"). It also documents the attributes and operations of the classes. The figure above shows an example class diagram.

Sequence diagrams have similarities with flow-charts. They demonstrate the sequence of communication between objects at run-time. All of the objects involved in the activity are listed across the top of the diagram, and the sequence runs down the page. Horizontal lines represent communication between objects. The figure alongside shows an example sequence diagram.

They say a picture paints a thousand words. Drawing diagrams as part of your design is almost a necessity. If you use UML diagrams then the "language" of your diagrams is universal, so your diagram will be understood by a greater number of people.

There are many sources for further reading on the subject of the UML. For those new to it I recommend the OMG Primer[6] and Martin Fowler[7].

Software Modelling

Drawing UML diagrams can be done with pencil and paper, but in many cases it is preferable to use some computer-based drawing package to ease storage and maintenance of the diagrams. A number of specialised packages are available for drawing and modelling.

Modelling packages are superior to plain drawing packages because they build a model behind the scenes that collates all the information from all of your diagrams. If you've already drawn and defined a class on one diagram and then choose to show it on another, the information you entered on the first diagram will automatically be echoed to the second. If you change the information in one diagram, all others are automatically updated. Thus, your diagrams remain consistent with each other.

A number of packages are available commercially. A selection of these were given a review[8] in the newsletter of the UK independent SAS User Group (VIEWS). The two figures shown in "The UML" were drawn with MagicDraw (http://www.nomagic.com) and Rational Rose (http://www.rational.com) respectively.

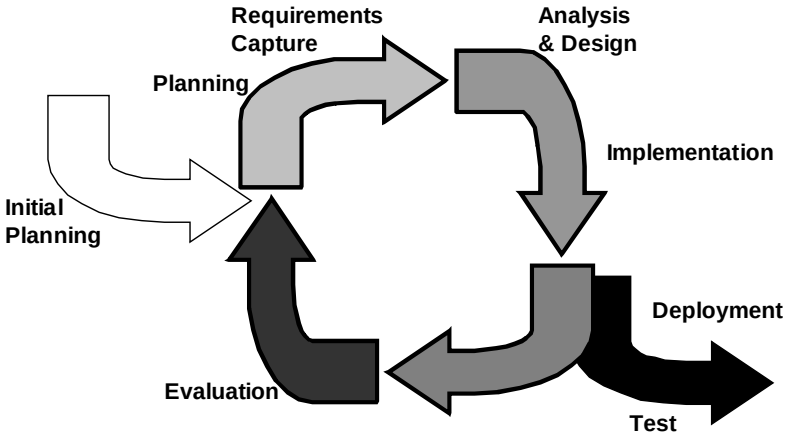
Iterative Development

There is no hard-and-fast reason why they should be but iterative development seems to be associated with OO and waterfall with traditional approaches. Certainly, the vast majority of OO practitioners use the iterative development method.

In my experience, iterative development pleases sponsors, users, and developers alike. Sponsors like it because it decreases the inevitable risk involved in any sizeable project. Users like it because a) they do not feel bound-in to a large complicated specification from the outset, and b) they see results quickly.

Traditional waterfall development consists of three key phases: Define, Build, and Test. These are run sequentially. Iterative uses the same phases but they are used for small parts of the project, and those parts of the project run in parallel.

As the project progresses, the delivered article at the end of each phase grows in size. The risk in the project is attacked at earlier stages because the deliverables are functional, integrated articles.



One of the deliverables of the initial planning phase is often an architecture that will provide a solid, resilient baseline for the subsequent detailed design and development. Achieving a solid architecture is crucial. That architecture will have been built using many of the techniques that I have discussed earlier, such as CRC cards for analysis and UML modelling software for documenting and communicating the architectural design.

Conclusion

Object-oriented techniques are an established means of developing applications. SAS software supports the development of object-oriented applications, but it does not do the object-oriented analysis and design. Without an object-based analysis and design, the resulting application will not be taking advantage of the merits of object-orientation.

Whilst CASE tools continue to grow in their abilities, they show no signs yet of replacing the analyst and designer! The tasks of analysing and designing continue as a requirement for successful application development. If

- CRC cards are used with the analysis;
- an appropriate method of control is chosen at the design stage; and
- the architecture and design are documented and communicated with the UML,

then the developers will be better able to create a truly object-oriented application (and achieve some or all of the associated benefits).

References

[1] Booch, Grady 1994. *Object-Oriented Analysis and Design With Applications.* The Benjamin/Cummings Publishing Company, Inc.

[2] Wirfs-Brock, Rebecca 1990. *Designing Object-Oriented Software.* PTR Prentice-Hall, Inc.

[3] Beck, Kent 1989. *A Laboratory for Teaching Object-Oriented Thinking.* http://c2.com/doc/oopsla89/paper.html

[4] Bellin, David 1997. *The CRC Card Book.* Addison, Wesley, Longman, Inc.

[5] Wirfs-Brock, Rebecca. *Characterising Your Application's Control Style.* http://www.wirfs-brock.com/characterizing_object_control_style.htm

[6] OMG 1998. *What Is OMG-UML and Why Is It Important?* http://www.omg.org/news/pr97/umlprimer.html

[7] Fowler, Martin 1997. *UML Distilled.* Addison, Wesley, Longman, Inc.

[8] VIEWS 1999. *VIEWS News, issue 5.* http://www.views-uk.demon.co.uk/issue5.pdf

Biography

Andrew Ratcliffe is a freelance SAS software consultant with over 15 years experience of SAS software. He specialises in object-oriented application development. Through his company (Ratcliffe Technical Services Limited), he is able to offer services including analysis and design, consultancy, and mentoring. Andrew is editor of the NOTE: free e-newsletter.

Email: andrew@ratcliffe.co.uk

Web: www.ratcliffe.co.uk

Newsletter subscription:

www.ratcliffe.co.uk/note_colon

Optimizing Data Extraction from Oracle® Tables

Caroline Bahler, Meridian Software, Inc.

Abstract

The SAS/ACCESS® product for Oracle® offers a fast and easy interface for reading Oracle tables - access views or SQL Pass-Thru facility. However, if your extraction requirements include joining Oracle tables or extracting date ranges then you need to know Oracle's "flavor" of SQL. In addition, efficient use of SAS/ACCESS methods may require some knowledge of how the Oracle database was designed and what Oracle operational options are in effect. The objective of this paper is to discuss some of the potential pitfalls and efficiencies when working with SAS/ACCESS to extract information from Oracle. This paper assumes the reader is familiar with the SAS/ACCESS product and specifically targets SAS/ACCESS and Oracle operational and SQL options that can be used to optimize data retrieval.

Extracting Data from Oracle

ACCESS Views

An Access view creates a "road map" to the location of a table and permits data extraction to occur. Access views are created using PROC ACCESS. A typical invocation of the ACCESS procedure would be:

```
proc access dbms=oracle;

/* create access descriptor */
 create work.sales.access;
   user=MyUserid;
   orapw=MyPassword;
   path='@ORAPATH.WORLD';
   table=acme_na.Sales_NorthAmer_1999;
   assign=yes;
   list all;

/* create view descriptor */
 create work.sales.view;
   select all;

run;
```

(Note - @ is an Oracle specific term used to designate the correct Oracle SQLNET® path. Whether the @ is utilized is dependent upon how Oracle is set-up at your site.)

In the example above a view is created instead of a data set. The advantage of this is that the view only needs to be created once and then can be reused. A new view needs to be generated only when the Oracle table is modified (new columns added or new indexes NOT new rows). Therefore, a permanent set of views can be created that can be used over and over without worrying about new rows within the table.

Tip 1 – Assess when to allow SAS to automatically assign variable name to Oracle table columns. Note: this tip is now specific for those users who still work with SAS version 6.12 and below.

In the example above, the use of the ASSIGN statement automatically creates a SAS variable name for each Oracle column. The new variable name consists of the first eight (8) characters of each column name[1]. The problem with allowing SAS to automatically assign variable name is illustrated below:

Oracle Column Name	SAS Variable Name
Product_Number	Product
Product_Type	Product1
Product_Serial_Number	Product2
Product_Sales	Product3

In each case the first eight (8) characters is product_. SAS® handles this situation by adding a number to the end of each variable name. Unique variable names are created, but the names are not indicative of the data stored within each variable. To assign unique variable names to these columns, use the RENAME statement within the CREATE statement to individually assign variable names to columns. So from the previous example:

```
proc access dbms=oracle;
/* create access descriptor */
 create work.sales.access;
  user=MyUserid;
  orapw=MyPassword;
  path='@ORAPATH.WORLD';
  table=acme_na.Sales_NorthAmer_1999;
  assign=yes;

  /* rename oracle columns to meaningful */
  /* SAS variable names                  */
  rename Product_Number = prodno
         Product_type = prodtype
         Product_Serial_Number=serialno
         Product_Sales = prodsale;

  list all;

/* create view descriptor */
 create work.sales.view;
   select all;
run;
```

Note: The RENAME statement must be used when the access descriptor is created. RENAME and ASSIGN=YES are mutually exclusive and cannot be used together.

Therefore, if the column names are eight (8) characters or less or uniquely named, then having SAS assign the name is appropriate.

Tip 2 – Efficiency considerations

The amount of data within a table will directly affect the time and resources needed in both SAS and Oracle to extract the information requested by the view. An ACCESS view reads and extracts from the Oracle table every time it is utilized. This is not a problem if the table from which you are extracting is a small lookup table containing a few hundred rows. However, if the table contains many thousands of rows, it may be more efficient to utilize the view to create a data set[1]. This data set is then available for use. The key factors that should be evaluated when considering this strategy are:

1. How many times is the table accessed within a program or application? If the table is accessed several times within a program or application, then creating a data set is a good choice.
2. How much temporary space is available for SAS data?
3. Can the data extracted from the table be subset when the table is accessed by using a WHERE statement? Use of an efficient where clause can reduce the amount of data and the amount of time needed for data extraction.
4. Can you reduce the number of columns you are selecting? SAS must convert the data within each column selected from Oracle to SAS format. So, by decreasing the number of columns selected, the resources and time needed to extract data is decreased. You can select columns when a view is created by using a KEEP or DROP statement within a DATA step or a VAR statement within a procedure.

You may need to benchmark to determine whether creation of a data set or a view is the more efficient option.

For example:

```
proc access dbms=oracle;
/* create access descriptor */
 create work.sales.access;
  user=MyUserid;
  orapw=MyPassword;
  path='@ORAPATH.WORLD';
  table=acme_na.Sales_NorthAmer_1999;
  assign=yes;

  /* rename oracle columns to      */
  /* meaningful SAS variable names */
  rename Product_Number = prodno
         Product_Type = prodtype
         Product_Serial_Number=serialno
         Product_Sales = prodsale;

  list all;

/* create data set  */
 create work.sales;
  select     Product_Number
             Product_Type
             Orderno
             Product_Sales;
   subset where Product_Sales > 100000;
run;
```

Note: The where clause of the SUBSET statement is sent "as is" to Oracle, so it must contain correct Oracle syntax.

When to Use Access Views

- Information from only one table at a time is required.
- You are extracting data from a table with a small number of rows.
- You are accessing table information one row at a time or allowing a user to update data within a table row.

SQL Pass-Thru

PROC SQL allows you to execute Oracle[1] commands or create SAS data sets/views from Oracle queries through the SQL Pass-Thru facility.

Connection options - SQL Pass-Thru requires that a connection to Oracle first is established before an Oracle command or query can be executed. The CONNECT statement requires the same information as in PROC ACCESS user id, password, Oracle path. For example the following statements will connect PROC SQL to Oracle with the same parameters specified for PROC ACCESS.

```
proc sql;
  Connect to oracle(
       userid='MyUserid'
       orapw='MyPassword'
       path="@ORAPATH.WORLD");

...

quit;
```

Tip 3 – Other CONNECT options. SQL Pass-Thru has two (2) other connection options, buffsize and preserve_comments, that can decrease the amount of time required to extract data from the results of an Oracle query.

[1] All Oracle® comments refer to Oracle version 7, 7.1, 7.2

BUFFSIZE – this option specifies the number of rows that will be transferred from Oracle to SAS when a fetch occurs, i.e. the number of rows that go into a buffer which moves the Oracle table rows from Oracle to SAS[1]. The default BUFFSIZE is 25 rows and the maximum is 32,767 rows. Increasing the number of rows within the buffer can enhance extraction performance.

Depending upon your particular hardware and Oracle setup using a BUFFSIZE between 5,000 and 10,000 will give the best performance in most cases. Somerville and Cooper in their SUGI 23 paper ran tests with Oracle tables containing 5.8 million records and found that a BUFFSIZE of 5000 gave them their best results[3]. Note that BUFFSIZE can not be set within PROC ACCESS or DBLOAD.
An example of using BUFFSIZE:

```
proc sql;
  Connect to oracle(
       userid='MyUserid'
       orapw='MyPassword'
       path="@ORAPATH.WORLD,buffsize=5000")
       ;

 create table sales as
  select *
    from connection to Oracle
     (select *
        from acme_na.Sales_NorthAmer_1999);
quit;
```

When to Use SQL Pass-Thru

- The output data set requires information from more than one Oracle table.
  - Subqueries – this is a technique that uses the information from one table to subset another. Subquerying can be a very efficient way of selecting only the information needed from a table. For example –

```
proc sql;
 connect to oracle(
    userid='MyUserid'
    orapw='MyPassword'
    path="@ORAPATH.WORLD,buffsize=5000")
    ;

create table sales as
 select *
   from connection to Oracle
    (select *
       from acme_na.Sales_NorthAmer_1999
      where CustomerID in
         (select CustomerID
            from acme_na.Customer
      where region like 'NORTH%')
    );
quit;
```

In the example above, a subquery made sense because we wanted just the customers residing in the northern regions. It is also efficient since only the required information was selected from both tables.

  - Joins – Two or more tables are joined using either an inner or outer join. Note: It is a good idea to evaluate whether it is more efficient to join the tables within Oracle or SAS.
- Extracting data from a table with a large number of rows. The use of the BUFFSIZE option within the path connect parameter greatly speeds up extracting data from a large table.
- Need to perform an Oracle command. There are instances where the ability to submit an Oracle command first is important. For example you need to load a table from a SAS data set using PROC DBLOAD, but first you need to drop the table so that new columns can be added.

Dynamic DBMS Engines

Dynamic DBMS Engines – this essentially replaces SAS/ACCESS views and descriptors! In version 7 and 8 the LIBNAME statement now connects to the DBMS server. For example:

```
Libname oralib oracle user=MyUserId
   password=MyPassword
   path="@ORAPATH.WORLD"
   readbuff = 5000
   ;
```

This means that all Oracle tables are accessible through the use of LIBNAME statement! Oracle tables are accessed in this case identically to SAS data sets.

Querying the Oracle Data

Oracle Meta Data

Oracle meta data consists of an extensive set of tables containing information about all database tables, user privileges, etc. The main meta data tables of interest are those containing information about the tables you need to access to obtain the information needed. Note: While the term table is used for descriptive purposes, Oracle considers all data dictionary tables to be views. Table 1 lists all Oracle dictionary tables[4] of interest.

Tip 4 – Listing Oracle table columns. A quick way to get basic information about the columns in an Oracle table is to use the DESC command (it does not need to be capitalized) in SQLPLUS®. The DESC command prints a listing of all columns in alphabetical order with their associated format and length.

Table 1. Oracle Data Dictionary Tables[4].

Table Name*	Description
USER_ : these are views containing information about tables owned by a particular user id	
USER_TABLE	Table listing
USER_TAB_COLUMNS	Column listing by table
USER_CONSTRAINTS	Listing of constraints defined for a table (primary, foreign, unique).
USER_CONS_CONSTRAINTS	Used with USER_CONSTRAINTS to define a table's primary and foreign keys.
USER_INDEXES	Listing of indexes defined for a table.
USER_IND_COLUMNS	Used with USER_INDEXES to defined which columns within a table are used within an index. Note – Oracle automatically creates an unique index for all unique and primary keys.
ALL_ : these are views containing information about all tables accessible by a particular user id	
ALL_TABLE	Table listing
ALL_TAB_COLUMNS	Column listing by table
ALL_CONSTRAINTS	Listing of constraints defined for a table (primary, foreign, unique).
ALL_CONS_CONSTRAINTS	Used with ALL_CONSTRAINTS to define a table's primary and foreign keys.
ALL_INDEXES	Listing of indexes defined for a table.
ALL_IND_COLUMNS	Used with ALL_INDEXES to defined which columns within a table are used within an index. Note – Oracle automatically creates an unique index for all unique and primary keys.

*Note: In addition, Oracle is case sensitive when using these table(view) names in queries the table name MUST be capitalized.

Why do you need meta data?

In many cases, a programmer can go to the dba and get a printed copy of the data dictionary including the entity relationship diagrams. However that may not always be the case, so the programmer is forced to dig for the information that is needed (i.e. the dba for table names and meta data for table information). Also remember Oracle is a relational database and a single database can contain hundreds (even thousands) of tables! Relationships (primary and foreign keys) between tables become extremely important when joining these tables. Finally, meta data can be used to write queries on the "fly" by utilizing the data dictionary within query–writing macros or SCL.

Where to do joins

In general, Oracle tables should be joined within Oracle, because all optimization features are available to enhance the performance of a join. To evaluate where to join Oracle tables, you should ask the following questions:

- What type of join is needed to provide the required information in the resulting data set?

Determining what the result data set should look like will identify the type of join needed. Oracle has two types of joins available – an inner and an outer join[5,6].

- The inner join produces a result data set that contains only those rows that match exactly in all parent tables.
- The outer join will produce a result data set that contains all rows from all parent tables (similar to using a MERGE statement with a BY statement in a DATA step)[5].

Make sure that using either type of SQL join will provide the needed results. Remember, within SQL (SAS and Oracle) both types of joins can result in Cartesian products, so check the parent tables carefully to prevent unintended extra rows[7].

However, there are cases where Oracle does not have the tools to allow for the type of join necessary. Table 2 lists all of the types of joins available within SAS and Oracle. Oracle cannot perform joins B or D but SAS can[8].

To perform an inner join within Oracle, use the following syntax:

```
create table sales as
 select *
   from connection to Oracle
    (select b.CompanyName, a.Sales
      from acme_na.Sales_NorthAmer_1999 a,
           acme_na.Customer b
     where a.CustomerId=b.CustomerId);
```

Table 2. Types of joins and their availability within SAS and Oracle.

Result Data Set	Type of Join	Availability	
		SAS	Oracle
A. All data values from all parent* tables.	Full Outer Join	X	X
B. All data values from a single parent table (base) and all data values from the other table(s) that match the data values of the joining variables within the base.	Non-base parent data set(s) are used as • "Look-up" table(s) • Right or Left outer join.	X	
C. Only those data values from all parent tables that contain the same data values within the joining variables.	Inner Join	X	X
D. Placement of parent tables side by side	One-to-one Merge	X	
E. Expansion of the result data set to include all levels of a non-common variable (Cartesian products).	Many-to-many Join (Inner or Outer Join)	X	X

* Parent table = one of the tables joined to produce the result data set.

To perform an outer join, use the following syntax:

```
create table sales as
 select *
   from connection to Oracle
    (select a.Region,
            a.Territory,
            b.Sales_Rep
       from acme_na.NorthAmer_Region a,
            acme_na.Sale_Representative b
      where a.Region=b.Region(+));
```

- Can the table be joined by primary or foreign keys? Are indexes available?

Using the primary and foreign keys within tables improves the performance of joins. This is because Oracle automatically creates indexes on the tables for these keys and the optimizer within Oracle will use these indexes during any join[4]. In addition, all other indexes on a table are available so performance of the join will be enhanced. By moving the tables into SAS data sets, these indexes are lost and the join is performed on unindexed variables.

- What are the sizes of the tables to be joined?

The decision to join a set of tables within Oracle or SAS can be affected by table size. To join the tables within SAS, all tables must first be extracted to SAS and then joined. The larger the table, the more time it takes to extract it to SAS. CPU time is expended before the join takes place. In comparison, by joining the tables within Oracle, with indexes in place, the join takes less time and only one extract needs to take place.

However, there is a circumstance where joining the tables within SAS may have to occur – when Oracle does not have enough temporary tablespace (work space) for the joins to occur. Finally, if one or more of the tables is extremely large, (1 million plus records) then the join may have to occur in smaller subsets with the total result data set being concatenated within SAS.

Is the database optimized for OLTP (on-line transaction processing) or as a data warehouse?

A database that is optimized for OLTP will be extremely fast at getting all of the information required about a single customer for example. Other types of joins may be much slower within that type of database than within SAS. In addition, your DBA may not approve of any large CPU drain during "peak" usage times. In contrast, a database optimized for data warehouse usage will be designed to optimize joins requiring large amounts of data.

- What does your DBA say?

Finally, the DBA is your best source of information about the database. They can suggest "best" practices for joining tables within the database.

Oracle SQL Tips

Dates

Oracle Date Formats[9]

MM	Number of month for example 12
MON	Three letter abbrev for example NOV
MONTH	Full month name
DD	Day of month
YYYY	Four digit year
YY	Two digit year
HH	Hour 1-12
HH24	Hour 1-24
MI	Minutes
SS	Seconds

Some useful Oracle date functions[9] are

- SysDate – this is equivalent to the DATE() or TODAY() functions within SAS and returns the current date and time.
- TO_CHAR(*date*,'*format*') – used in queries to change an Oracle datetime value into a formatted value. Similar in function to using the following SAS syntax - put(*date*,*mmddyy*10.). This format can be used within a query as follows:

```
create table sales as
 select *
   from connection to Oracle
    (select b.CompanyName, a.Sales
       from acme_na.Sales_NorthAmer_1999 a,
            acme_na.Customer b
      where a.CustomerId=b.CustomerId and
 to_char(Updt_date,'MM/DD/YYYY') =
 '01/01/2000');
```

  Using TO_CHAR can be tricky since you are specifically turning a numeric datetime value into a character. You need to be careful how you use this function, in general TO_CHAR is more useful in an equality situation.

- TO_DATE('*date*','*format*') – from my own experience and talking to knowledgeable Oracle programmers this is the preferred function. For example:

```
create table sales as
 select *
   from connection to Oracle
    (select b.CompanyName, a.Sales
       from acme_na.Sales_NorthAmer_1999 a,
            acme_na.Customer b
      where a.CustomerId=b.CustomerId
        and Updt_date between
 to_date('01/01/2000','MM/DD/YYYY') and
 to_date('02/15/2000','MM/DD/YYYY')  );
```

Tip 5 – Feeding dates into an Oracle query.

Using the same query as above, we can generalize for dates as follows:

```
Data _null_;
 To=today();
 From=intx('week',to,-8);
 Call symput('from',"'"||
      put(from,mmddyy10.)||"'"");
 Call symput('to',"'"||
      put(to,mmddyy10.)||"'"");
Run;

Proc sql;
 Connect to oracle(
      userid='MyUserid'
      orapw='MyPassword'
      path="@ORAPATH.WORLD,buffsize=5000")
      ;

 create table sales as
  select *
    from connection to Oracle
     (select b.CompanyName, a.Sales
       from acme_na.Sales_NorthAmer_1999 a,
            acme_na.Customer b
      where a.CustomerId=b.CustomerId
        and Updt_date between
          to_date(&from , 'MM/DD/YYYY') and
          to_date(&to , 'MM/DD/YYYY')  );


  disconnect from oracle;
quit;
```

Note: Double quotes (") are not used around the macro variables as in SAS. Instead, the quotes are included as part of the macro variable value. Macro variables are extremely useful within SQL Pass-Thru in automating the Oracle query.

One use of macro variables and meta data tables is to determine the fields within a table. The resulting data set can be used to build a query by passing the fields into the query as macro variables.

Other Tricks of the Trade

- Quotes – the use of quotes can be tricky within Oracle. As stated above, it is best to pass the quotes within the macro variable value. Oracle does not handle double quotes (") within the WHERE portion of a query. Use of double quotes results in a cryptic error message. Instead, incorporate the quotes into the macro variable passed.

  In the example below, each of the passed macro variables contains the necessary quote for the character values.

```
Data _null_;
 Set param;
 To=today();
 From=intx('week',to,-8);

 Call symput('state',"'"||
      trim(state)||"'");
 Call symput('from',"'"||
      put(from,mmddyy10.)||"'"):
 Call symput('to',"'"||
      put(to,mmddyy10.)||"'");
Run;

Proc sql;
 Connect to oracle(
      userid='MyUserid'
      orapw='MyPassword'
      path="@ORAPATH.WORLD,buffsize=5000")
      ;

 create table sales as
```

```
  select *
    from connection to Oracle
    (select b.CompanyName, a.Sales
       from acme_na.Sales_NorthAmer_1999 a,
            acme_na.Customer b
      where a.CustomerId=b.CustomerId
        and Updt_date between
          to_date(&from , 'MM/DD/YYYY') and
          to_date(&to , 'MM/DD/YYYY')
        and state = &state );


  disconnect from oracle;
quit;
```

Tip 6 - Handling a quote embedded within a field value. If a field value has an embedded quote in its value, a comment field or company name for example, then you should use special care when you are within a query. In the following example a title contains a single quote:

Mr. Roger's Neighborhood

To use this title in a PROC SQL query, surrounding the title with double quotes is sufficient. In an Oracle query, the following is needed:

'Mr. Roger''s Neighborhood'

Note: the two single quotes are side by side[5]. Therefore, any text value that may contain quotes could be handled as follows:

```
Data _null_;
 Set titles end=eof;
   Where title contains "SESUG";
Quottest=index(title,"'");
If quottest > 0 then

Title=scan(title,1,"'")||"''"||scan(title,2
,"'");

If _n_=1 then titles="'"||trim(title)||"'";
 Else title=trim(title)||"',"||
trim(title)||"'";

If eof then call symput('titles'title');
Run;

Proc sql;
 Connect to oracle (
      userid='MyUserid'
      orapw='MyPassword'
      path="@ORAPATH.WORLD,buffsize=5000")
      ;

 create table proceeds as
  select *
    from connection to oracle
    (select author_lastname lastname,
            author_firstname firstnme,
            title
       from bibliography
      where title in ( &titles ) );
    disconnect from oracle;
quit;
```

- "Temporary Tables" – within Oracle, each user has personal tablespace for creating tables. This tablespace can be put to use for creating temporary query result tables depending upon their size. The idea here is to run a query subsetting a much larger table and then use the results of that query to subset other tables. For example –

```
Data _null_;
 Set titles end=eof;
   Where title contains "SESUG";
Quottest=index(title,"'");
If quottest > 0 then

Title=scan(title,1,"'")||"''"||scan(title,2
,"'");

If _n_=1 then titles="'"||trim(title)||"'";
 Else title=trim(title)||"',"||
           trim(title)||"'";

 If eof then call symput('titles'title');
Run;

Proc sql;
 Connect to oracle (
      userid='MyUserid'
      orapw='MyPassword'
      path="@ORAPATH.WORLD,buffsize=5000")
      ;

    execute
      (create table temp as
        select author_lastname lastname,
               author_firstname firstnme,
               title
          from bibliography
         where title in ( &titles ) )
       by oracle;

     create table advtut as
      select *
         from connection to oracle
        (select *
           from temp
         where title like 'SQL%' );

    execute (drop table temp) by oracle;

    disconnect from oracle;
quit;
```

Note: The temporary table needs to be dropped when you are finished! A permanent Oracle table is being created, so this house-cleaning step is important.

Tip 7 – Reserved words, "words" that cannot be used in a query except within the correct syntax for that "word". Appendix 1 contains the whole list of Oracle reserved words. For example, count is a

function both in Oracle and SAS it's syntax is count(*) or count(field_name). This is the only way that count can be used in Oracle otherwise an error message will occur.

References

1. SAS Institute Inc. 1993. SAS/ACCESS Interface to Oracle Usage and Reference. Version 6, Second Edition, Cary, NC. SAS Institute Inc.
2. SAS Institute Inc. 1994. SAS/ACCESS Software for Relational Databases: Reference, Version 6, First Edition, Cary, NC. SAS Institute Inc.
3. Somerville, Clare and Copper, Clive. 1998. Optimizing SAS® Access to an Oracle Database in a Large Data Warehouse. Proceedings of the Twenty-Third Annual SAS Users Group International Conference, Cary, NC. SAS Institute. pp 511-520.
4. Koch, George and Loney, Kevin. 1995. Chapter 24. The Hitchhiker's Guide to the ORACLE7 Data Dictionary. Oracle: Complete Reference Third Edition. Oracle Press™, Berkeley, CA. Osborne McGraw-Hill Inc. pp 540-586
5. Koch, George and Loney, Kevin. 1995. Chapter 25. Alphabetical Reference. Oracle: Complete Reference Third Edition. Oracle Press™, Berkeley, CA. Osborne McGraw-Hill Inc.
6. Koch, George and Loney, Kevin. 1995. Chapter 10. When One Query Depends Upon Another. Oracle: Complete Reference Third Edition. Oracle Press™, Berkeley, CA. Osborne McGraw-Hill Inc.
7. Koch, George and Loney, Kevin. 1995. Chapter 9. Grouping Things Together. Oracle: Complete Reference Third Edition. Oracle Press™, Berkeley, CA. Osborne McGraw-Hill Inc.
8. Bahler, Caroline. 1996. It Takes at Least Two to Tango -A Data Set Joining Primer. Proceedings of the Twenty-Second Annual SAS Users Group International Conference, Cary, NC. SAS Institute. pp 190-198.
9. Koch, George and Loney, Kevin. 1995. Chapter 7. Dates: Then, Now, and the Difference. Oracle: Complete Reference Third Edition. Oracle Press™, Berkeley, CA. Osborne McGraw-Hill Inc. pp. 173-189.
10. Gona, Vino and Van Wyk, Jana. 1998. Version 7 Enhancements to SAS/ACCESS Software. Proceedings of the Twenty-Third Annual SAS Users Group International Conference, Cary, NC. SAS Institute. pp 336-341.

Trademarks

Contact Information

Caroline Bahler

Meridian Software, Inc.

12204 Old Creedmoor Road

Raleigh, NC 27613

(919) 518-1070

merccb@meridian-software.com

Appendix 1: Oracle Reserved Words

access
add
admin
after
all
allocate
alter
analyze
and
any
archive
archivelog
as
asc
audit
authorization
avg
backup
become
before
begin
between
block
body
by
cache
cancel
cascade
change
char
character
check
checkpoint
close
cluster
cobol
column
comment
commit
compile
compress
connect
constraint
constraints
contents
continue
controlfile
count
create
current
cursor
cycle
database
datafile
date
dba
dec
decimal
declare
default
delete
desc
disable
dismount
distinct
double
drop
dump
each
else
enable
end
escape
events
exceptions
exclusive
exec
exists
explain
extent
externally
fetch
file
float
flush
for
force
foreign
fortran
found
freelist
freelists
from
function
go
goto
grant
group
groups
having
identified
immediate
in
including
increment
index
indicator
infile
initial
initrans
insert
instance
int
integer
intersect
into
is
key
language
layer
level
like
link
lock
long
maxextents
minus
mode
modify
noaudit
nocompress
not
nowait
null
number
of
offline
on
online
option
or
order
pctfree
prior
privileges
public
raw
rename
resource
revoke
row
rowid
rownum
rows
select
session
set
share
size
smallint
start
successful
synonym
sysdate
table
then
to
trigger
uid
union
unique
update
user
validate
values
varchar
varchar2
view
whenever
where
with

SERENDIPITY

SECTION CHAIRS

Derek Nguyen
DataLogic Consulting, Inc.

Ian Whitlock
Westat

SERENDIPITY

Elegant Tables: Dressing Up Your TABULATE Results

Lauren Haworth
Genentech, Inc., South San Francisco

➢ INTRODUCTION

Once you've taken the time to learn the basics of TABULATE, you'll quickly discover that creating the table is the easy part. It's making it look nice that's hard.

This paper will show you a number of tips and tricks for designing and formatting your table to make it concise, informative and attractive. The paper then goes on to show how to move your output to HTML and to a word processor so that your results can be easily distributed to others.

➢ TIP #1: MODIFYING STATISTIC LABELS

This first tip is designed to make your table easier to understand. When we put statistics in our tables, PROC TABULATE adds labels for the appropriate rows or columns that indicate which statistic has been selected. As SAS programmers, the labels N, MEAN, and STD may make perfect sense. However, what about the end-users?

To give them a hand, you can rename the default statistics. The following code shows how it's done. The gray shaded areas are the parts of the code that have been modified to change the labels. To give a statistic a new label to replace the default, place an equal sign after the statistic keyword, and follow that with the new label in quotes.

```
PROC TABULATE DATA=TEMP;
   VAR RENT;
   TABLE RENT, ALL*
      (N='Number of Observations'
       MEAN='Average'
       STD='Standard Deviation');
RUN;
```

The resulting table is shown below. Notice that since the new labels were longer than the originals, PROC TABULATE made the spaces for the labels two lines deep. PROC TABULATE will make room for whatever labels you create, but you should try to keep them as short and simple as possible.

	All		
	Number of Observations	Average	Standard Deviation
rent	126.00	1129.66	544.77

➢ TIP #2: MODIFYING VARIABLE LABELS

Just as we changed the labels for the statistics in the previous example, we can also change the labels for the variables. In the following code, a variable has been labeled in the TABLE statement by using an equal sign after the variable name and then putting the label in quotation marks.

```
PROC TABULATE DATA=TEMP;
   VAR RENT;
   TABLE RENT='Monthly Rent',
      ALL*(N='Number of Observations'
           MEAN='Average'
           STD='Standard Deviation');
RUN;
```

The new table is shown below.

	All		
	Number of Observations	Average	Standard Deviation
Monthly Rent	126.00	1129.66	544.77

➢ TIP #3: HIDING STATISTIC LABELS

The next area of cleanup we'll tackle is excessive statistic labels. Every time you display a statistic in a table, by default PROC TABULATE generates a row or column heading for that statistic.

In the example below, we have a table that calculates mean rent by city. Because there are three cities, the label "Mean" is repeated three times.

	city		
	Portland	San Francisco	Seattle
	Mean	Mean	Mean
Monthly Rent	859.67	1691.79	1010.37

This table would be much more attractive if we could get rid of the repeated labels. This can be done by using the row variable label to hold the information about the statistic, and then deleting the statistic label. The revised code is below.

```
PROC TABULATE DATA=TEMP;
   CLASS CITY;
   VAR RENT;
   TABLE RENT='Average Monthly Rent',
         CITY*MEAN=' ';
RUN;
```

The statistic label is deleted by assigning it a blank label, which is created by putting a space between two quotes. This code produces the following table. Now the output is much more attractive, but no meaning has been lost (no pun intended).

	city		
	Portland	San Francisco	Seattle
Average Monthly Rent	859.67	1691.79	1010.37

➢ TIP #4: ANOTHER WAY TO RELOCATE THE STATISTIC LABEL

If you don't want to move your statistic into the row label, or it doesn't easily fit into the row label, there's another place you can put the extra information. If you look at the table above, you can see that there's a big empty box in the top left corner of the label. This is valuable space, and TABULATE lets you use it by adding a BOX= option. The code below illustrates how to use this option.

```
PROC TABULATE DATA=TEMP;
   CLASS CITY;
   VAR RENT;
   TABLE RENT='Monthly Rent',
         CITY*MEAN=' '
         / BOX='Averages';
RUN;
```

The new table looks like this:

Averages	city		
	Portland	San Francisco	Seattle
Monthly Rent	859.67	1691.79	1010.37

➢ TIP #5: HIDING A VARIABLE LABEL

Getting rid of the repeated 'MEAN' labels helped simplify the table. However, in the previous example, there's another superfluous label.

If you look at the column headings of the above table, putting the label 'city' above the three values 'Portland', 'San Francisco', and 'Seattle' is redundant. It's obvious that they are cities. This variable label can be removed the same way the 'MEAN' statistic labels were removed, by assigning a blank label.

```
PROC TABULATE DATA=TEMP;
   CLASS CITY;
   VAR RENT;
   TABLE RENT='Monthly Rent',
         CITY=' '*MEAN=' '
         / BOX='Averages';
RUN;
```

The new table is shown below.

Averages	Portland	San Francisco	Seattle
Monthly Rent	859.67	1691.79	1010.37

Compare this to the original table at the top left of this page. Notice how much more elegant the table looks, and no information has been lost. This is your goal in creating TABULATE output: to refine and simplify the table as much as possible, to make it easier for the reader to follow.

➢ TIP #6: CLEANING UP THE ROW HEADINGS

The previous examples looked at how to get rid of excess column headings. But you can also have problems with row labels. Consider the following table. It's the same table as the previous examples, but turned on its side.

```
PROC TABULATE DATA=TEMP;
   CLASS CITY;
   VAR RENT;
   TABLE CITY=' '*MEAN=' ',
         RENT='Average Monthly Rent';
RUN;
```

All that has changed is that the row and column dimensions have been reversed. However, if you look at the output below, something is wrong.

		Average Monthly Rent
Portland		859.67
San Francis-co		1691.79
Seattle		1010.37

This table has an extra set of blank boxes in the row heading. That's because the label for city has been set to a blank label. In the column heading, the box for a blank label is removed from the table. In a row heading, TABULATE leaves behind the empty box. To fix this, you need use the ROW=FLOAT option, as shown in the code below.

```
PROC TABULATE DATA=TEMP;
   CLASS CITY;
   VAR RENT;
   TABLE CITY=' '*MEAN=' ',
         RENT='Average Monthly Rent'
         / ROW=FLOAT;
RUN;
```

Now the table looks like this:

	Average Monthly Rent
Portland	859.67
San Francisco	1691.79
Seattle	1010.37

The ROW=FLOAT option is a good thing to leave turned on all of the time. It does no harm if you have no blank variable or statistic labels in your row headings, and you will need it whenever you do have blank labels.

➢ TIP #7: MODIFYING ROW HEADING WIDTHS

By default, TABULATE divides the line width between the row headings and the table cells holding row values. It uses a simple formula, which unfortunately does not account for the width of your row variable labels. So quite often, the row headings have too much or too little space for your labels.

In this example, a narrow line size setting has caused the space available for the row heading to become too small to hold the text of the label without wrapping.

	Portland	San Francisco	Seattle
Average Monthly Rent	859.67	1691.79	1010.37

To provide more space, you can override the default width for the row heading using the RTS option. The code for this is as follows:

```
PROC TABULATE DATA=TEMP;
   CLASS CITY;
   VAR RENT;
   TABLE CITY=' '*MEAN=' ',
         RENT='Average Monthly Rent'
         / ROW=FLOAT RTS=22;
RUN;
```

The RTS setting of 22 was computed by taking the width of the label and adding two (for the borders of the row heading). The revised table is shown below

	Portland	San Francisco	Seattle
Average Monthly Rent	859.67	1691.79	1010.37

➢ TIP #8: MODIFYING COLUMN WIDTHS

To change the width of row headings, you use RTS. Unfortunately, there isn't an equivalent CTS option for columns. Instead, to adjust the width of a column, you modify the format of the data displayed in that column.

By default, TABULATE uses the format BEST12.2 for all table values. This means unless you specify otherwise, every column will be 12 spaces wide. In our sample table, 12 spaces looks okay, but if we wanted to save space, we could reduce this to 9 spaces, which is the width of the longest word in a column label.

To change the format, add a FORMAT= option to the PROC TABULATE statement as in the following code.

```
PROC TABULATE DATA=TEMP FORMAT=9.2;
   CLASS CITY;
   VAR RENT;
   TABLE CITY=' '*MEAN=' ',
         RENT='Average Monthly Rent'
         / ROW=FLOAT RTS=22;
RUN;
```

The new table is shown below: It's a bit more compact than the original.

	Portland	San Francisco	Seattle
Average Monthly Rent	859.67	1691.79	1010.37

➢ TIP #9: USING APPROPRIATE FORMATS

Using the FORMAT= option allowed us to pick an appropriate column width. However, we haven't yet taken full advantage of this option. This table displays results that are dollar amounts. We can use the FORMAT= option to display the results in a more appropriate format. The DOLLAR format adds "$" and commas to dollar amounts. In addition, since these are large dollar amounts, we can get rid of the decimal places to make the table easier to read.

```
PROC TABULATE DATA=TEMP FORMAT=DOLLAR9.;
   CLASS CITY;
   VAR RENT;
   TABLE CITY=' '*MEAN=' ',
         RENT='Average Monthly Rent'
         / ROW=FLOAT RTS=22;
RUN;
```

The new table is shown below:

	Portland	San Francisco	Seattle
Average Monthly Rent	$860	$1,692	$1,010

➢ TIP #10: RE-ORDERING THE HEADINGS

When you generate a TABULATE table, by default the values of your CLASS variable are listed based on their internal values (their unformatted values). TABULATE gives you options to change this to the order of the data (so you can sort the data into the order that you want) or the order of the formatted values.

However, sometimes the table values and their formatted values are not in the order you want. There's a trick you can use to force your headings into the order you desire. To illustrate the technique, let's say we want to change the table from the previous examples so that the cities are listed in geographic order from north to south.

To get the table in this order you create a new format with leading blanks added to force the formatted values to sort into the order that you want. Then you use the ORDER=FORMATTED option on your PROC TABULATE statement.

```
PROC FORMAT;
   value cityft 1=' Portland'
                2='San Francisco'
                3='  Seattle';
RUN;
PROC TABULATE DATA=TEMP FORMAT=9.
   ORDER=FORMATTED;
   CLASS CITY;
   VAR RENT;
   TABLE RENT='Average Monthly Rent',
         CITY=' '*MEAN=' '
         / RTS=22;
RUN;
```

In this example, since we want Seattle to come first, its format is given two leading blanks. Portland comes next, so it gets one leading blank, and San Francisco is left alone. When the table is created, TABULATE uses these blanks to create the column heading order, but strips off the blanks before displaying the table. The result is that the resulting table is in the correct order but doesn't have any extra spaces.

	Seattle	Portland	San Francisco
Average Monthly Rent	$1,010	$860	$1,692

➢ TIP #11: FORMATTING PERCENTAGES

One of the most powerful parts of PROC TABULATE is the ability to specify complex percentages to display in your table. However, the default formatting of these percentages leaves a little to be desired.

For example, the following code is used to produce the table below. It calls for a table of parking availability by city, with column percentages and totals at the end of each column.

```
PROC TABULATE DATA=TEMP
   ORDER=FORMATTED;
   CLASS PARKING CITY;
   TABLE (PARKING='Has Parking' ALL)
      *PCTN<PARKING ALL>=' ',
      CITY=' '
      / ROW=FLOAT RTS=13;
RUN;
```

	Seattle	Portland	San Francisco
Has Parking			
No	47.37	40.74	29.41
Yes	52.63	59.26	70.59
All	100.00	100.00	100.00

Instead of putting percent signs after the percentages, TABULATE uses its standard BEST12.2 format. You might think that switching to the PERCENT format would solve the problem. However, here's what you get if you assign the format of PERCENT9. in the PROC TABULATE statement:

	Seattle	Portland	San Francisco
Has Parking			
No	4737%	4074%	2941%
Yes	5263%	5926%	7059%
All	10000%	10000%	10000%

What happens is that TABULATE multiplies all calculated percentages by 100 before displaying the results. Unfortunately, the PERCENT format also multiplies values by 100, so what you get is values that have been multiplied by 1000!

To add percent signs to TABULATE percentages, you need to create your own percentage format. You can do this with a PICTURE format.

```
PROC FORMAT;
   PICTURE PCTPIC low-high='000%';
RUN;
PROC TABULATE DATA=TEMP
   ORDER=FORMATTED FORMAT=PCTPIC9.;
   CLASS PARKING CITY;
   TABLE (PARKING='Has Parking' ALL)
      *PCTN<PARKING ALL>=' ',
      CITY=' '
      / ROW=FLOAT RTS=13;
RUN;
```

The new table now has percent signs, but the values have not been altered.

	Seattle	Portland	San Francisco
Has Parking			
No	47%	40%	29%
Yes	52%	59%	70%
All	100%	100%	100%

➢ TIP #12: ANOTHER WAY TO DISPLAY PERCENTAGES

Sometimes TABULATE gives you more output than you need. For example, in the above table we can see that 52% of the Seattle apartments in our dataset have parking available. We don't really need to know that 47% do not. In fact, with the rounding error in this table, it's somewhat confusing to show the percentages for "Yes" and "No". It would be more useful to display a single percentage that shows what percentage of apartments have parking, and ignore the percentage that do not (since it is implied).

You can do this with a simple DATA step trick. You compute a dummy variable that indicates whether the building has parking. All "Yes" answers are coded to 1; all "No" answers are coded to 0. Then you take the mean of that variable in your TABULATE table, and format it as a percentage.

```
PROC TABULATE DATA=TEMP
   ORDER=FORMATTED FORMAT=PERCENT9.;
   CLASS CITY;
   VAR PARKING;
   TABLE PARKING='% With Parking'*MEAN=' ',
      CITY=' '
      / ROW=FLOAT RTS=16;
RUN;
```

In this example, the variable PARKING happens to already be created as 1=Yes, 0=No, so we don't even need a data step. Also note that we can now use the PERCENT format, because we are calculating the percent ourselves so TABULATE will not multiply it by 100. The new table is shown below.

	Seattle	Portland	San Francisco
% With Parking	53%	59%	71%

This version of table is much smaller and easier to interpret. Keep this trick in mind whenever you are displaying percentages of a dichotomous variable.

➢ TIP #13: FORMATTING MISSING VALUES

No data is ever perfect, so chances are you're going to run into the problem of missing data. TABULATE can't make your missing data go away, but it does give you some control on how it is displayed. For example, look at the following table:

Number of Observations	Seattle	Portland	San Francisco
Has Parking			
No	.	22	10
Yes	20	32	24

This table has a period in one cell where there is missing data. This convention is familiar to SAS programmers, but the end user of your table may not know what the "." stands for.

Since this is a table of Ns, it would be nice to display a zero in that empty cell instead of the period. You can request this with the MISSTEXT option.

```
PROC TABULATE DATA=TEMPMISS
   ORDER=FORMATTED FORMAT=9.;
   CLASS CITY PARKING;
   TABLE PARKING='Has Parking'*N=' ',
      CITY=' '
      / ROW=FLOAT RTS=15
      BOX='Number of Observations'
      MISSTEXT='0';
RUN;
```

The revised table is displayed below.

Number of Observations	Seattle	Portland	San Francisco
Has Parking			
No	0	22	10
Yes	20	32	24

➢ TIP #14: REMOVING REDUNDANT INFORMATION

This next example uses several tricks to simplify a table with redundant information.

```
PROC TABULATE DATA=TEMP
   ORDER=FORMATTED FORMAT=6.;
   CLASS CITY;
   VAR RENT BEDROOMS;
   TABLE RENT='Monthly Rent'
      BEDROOMS='Number of Bedrooms',
      CITY=' '*(N MEAN)
      / ROW=FLOAT RTS=14.;
RUN;
```

The table is designed to show the N and MEAN for two variables for each city. However, notice that the N is the same for both variables in each case. So space is being wasted displaying the N twice in each column.

	Seattle		Portland		San Francisco	
	N	Mean	N	Mean	N	Mean
Monthly Rent	38	1010	54	860	34	1692
Number of Bedrooms	38	2	54	2	34	2

The table can be simplified by only displaying the N for one of the two variables. It can also be made more user friendly by displaying the results in one column per city instead of two.

```
PROC TABULATE DATA=TEMP
ORDER=FORMATTED;
   CLASS CITY;
   VAR RENT BEDROOMS;
   TABLE RENT='(N)'*N=' '*F=9.
      RENT='Average Rent'
         *MEAN=' '*F=DOLLAR9.
      BEDROOMS='Average Bedrooms'
         *MEAN=' '*F=9.2,
         CITY=' '
         / ROW=FLOAT RTS=16.;
RUN;
```

The trick in this table is to move the statistics to the row dimension and to use the variable RENT twice in the table statement. The first time it is labeled as "(N)" and is used to compute the N statistic. The second time it is labeled as "Average Rent" and it is used to compute the MEAN statistic. The second variable BEDROOMS is only used to compute a mean.

Also, notice how formats have been used in the TABLE statement. If you want to use a single format for the whole table, specify it in the PROC TABULATE statement. However, if you want to use different formats for each statistic or variable, you can apply the format directly to the statistic or variable by using an asterisk and the FORMAT= or F= syntax. In this example, F= is used to format the N as an integer, the variable RENT as a dollar amount, and the variable BEDROOMS as a number with two decimal places.

The new table, shown below, now has only three columns (instead of 6) and has one new row.

	Seattle	Portland	San Francisco
(N)	38	54	34
Average Rent	$1,010	$860	$1,692
Average Bedrooms	1.50	1.50	1.50

Whenever your table has repeated information, think of using tricks like this to simplify it.

➢ TIP #15: REMOVING THE ROW DIVIDERS

In each of the examples so far, we've used the default PROC TABULATE table grid. The table grid is the horizontal and vertical bar characters that make up the borders of the table rows, columns, and cells. The default table grid puts a border around every cell, row, column, and page. However, you don't have to use the standard table grid. If you'd like a table that looks a little less "boxy," you can tell PROC TABULATE to remove the row dividers.

Average Rent	Seattle	Portland	San Francisco
1 Bedroom	$887	$621	$1,284
2 Bedrooms	$1,134	$1,099	$2,100

If you look closely at the above table, you will see that each row in the table actually requires two lines of text. One has the data that goes in each of the table cells. The second line consists of a series of characters that form the divider between the rows. The following code uses the NOSEPS option to remove these row dividers from your table.

```
PROC TABULATE DATA=TEMP NOSEPS
   ORDER=FORMATTED F=DOLLAR9.;
   CLASS CITY BEDROOMS;
   VAR RENT;
   TABLE RENT=' '*BEDROOMS=' ',
      CITY=' '*MEAN=' '
      / ROW=FLOAT RTS=16
        BOX='Average Rent';
RUN;
```

The new table retains the boxes around the column headers, but now there are no row dividers in the body of the table.

Average Rent	Seattle	Portland	San Francisco
1 Bedroom	$887	$621	$1,284
2 Bedrooms	$1,134	$1,099	$2,100

This can be a great space-saving tool if you have a table that is too tall to fit on the page, but be careful. If your table has many columns, it may be hard to see which numbers line up across a row without the help of the row dividers.

➢ TIP #16: MODIFYING THE BORDER STYLE

Throughout this paper, the tables have been displayed with high-resolution smooth lines as borders. However, when you run TABULATE, your tables may look more like this:

```
-------------------------------------------
|Average Rent  |         |         |   San   |
|              | Seattle |Portland |Francisco|
|--------------+---------+---------+---------|
|1 Bedroom     |     $887|     $621|   $1,284|
|2 Bedrooms    |   $1,134|   $1,099|   $2,100|
-------------------------------------------
```

This setup is designed so that it can be displayed on low-resolution monitors, and printed on a line printer. But these days, even if you're on a mainframe, you're probably printing to a high-resolution laser printer. If you'd like to set up TABULATE to print tables with smooth line borders, you need to change your FORMCHAR option setting.

There are two ways to do this. One is to look up the hex codes for each of the special border characters for your printer. This means digging

out the printer manual and finding the codes for characters like "┐" and "┬".

The other option is to use the following FORMCHAR setting:

```
OPTIONS FORMCHAR='828384858687888898A8B8C'X;
```

If you're running SAS on Windows, you can change your font setting to SAS Monospace, and your table will be displayed and printed with the smooth borders of the previous examples.

If you're running SAS in an environment where you can't change the display and printing font, then you'll need to save your output and open it in a word processor. You can change the font to SAS Monospace and print it from there.

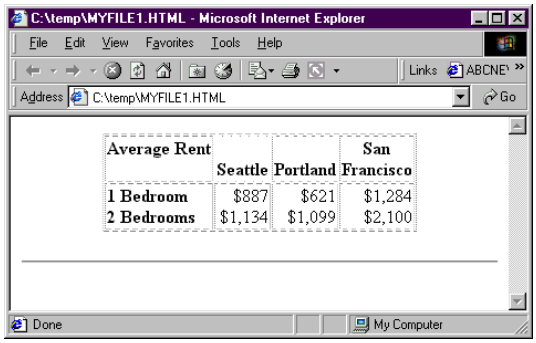
➢ TIP #17: EXPORTING TO HTML

If you plan on displaying your results on the Internet, or sharing them via the company intranet, or e-mailing the results to a client, the best thing you can do to make your output more attractive is to convert it to HTML.

In version 6, the code for creating HTML is:

```
OPTIONS FORMCHAR='828384858687888898A8B8C'X;
%TAB2HTM (CAPTURE=ON, RUNMODE=B);
PROC TABULATE DATA=TEMP NOSEPS
   ORDER=FORMATTED F=DOLLAR9.;
   CLASS CITY BEDROOMS;
   VAR RENT;
   TABLE RENT=' '*BEDROOMS=' ',
      CITY=' '*MEAN=' '
      / ROW=FLOAT RTS=16
        BOX='Average Rent';
RUN;
%TAB2HTM(CAPTURE=OFF, RUNMODE=B,
   OPENMODE=REPLACE,
   HTMLFILE=C:\TEMP\MYFILE1.HTML);
```

This example shows a minimal set of options, there are many more. In particular, note the FORMCHAR setting, which is required. The output is shown below.

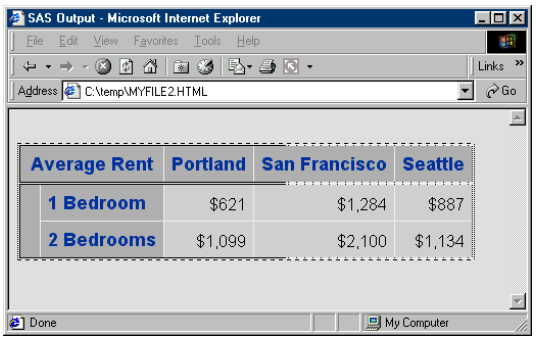


Average Rent	Seattle	Portland	San Francisco
1 Bedroom	$887	$621	$1,284
2 Bedrooms	$1,134	$1,099	$2,100

In version 8, the syntax is even easier:

```
ODS HTML BODY='C:\TEMP\MYFILE2.HTML';
PROC TABULATE DATA=TEMP NOSEPS
   ORDER=FORMATTED F=DOLLAR9.;
   CLASS CITY BEDROOMS;
   VAR RENT;
   TABLE RENT=' '*BEDROOMS=' ',
      CITY=' '*MEAN=' '
      / ROW=FLOAT RTS=16
        BOX='Average Rent';
RUN;
ODS HTML CLOSE;
```

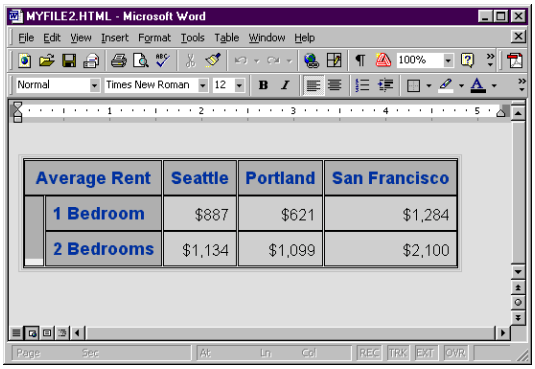
The output for version 8 is shown below:



Average Rent	Portland	San Francisco	Seattle
1 Bedroom	$621	$1,284	$887
2 Bedrooms	$1,099	$2,100	$1,134

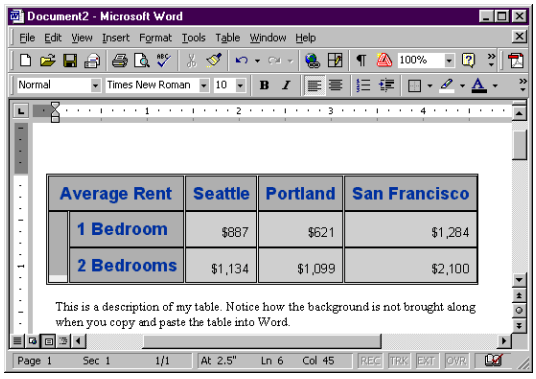
➢ TIP #18: IMPORTING INTO OTHER APPLICATIONS

Once you've created HTML output, it's easy to move your table into other applications. To move your table into a word processor, you can simple open the HTML file directly from the word processor.

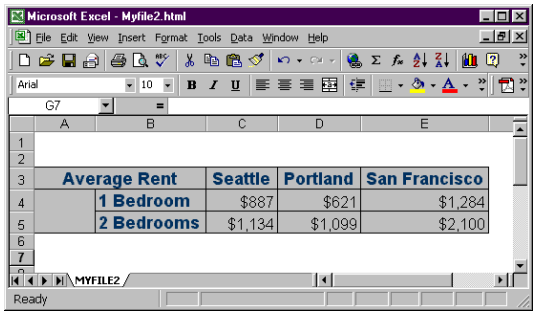


Average Rent	Seattle	Portland	San Francisco
1 Bedroom	$887	$621	$1,284
2 Bedrooms	$1,134	$1,099	$2,100

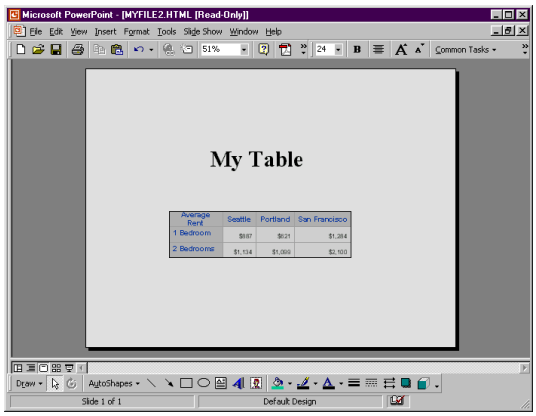
If you want to add your table to an existing document, simply open it in a browser, highlight the table, and use copy and paste to drop it into your word processor.



The same technique works well for moving your table into a spreadsheet. Just open the file directly from your spreadsheet.



You can also open an HTML file from PowerPoint. You may find that your file needs a bit of formatting once it is imported. Generally, you'll need to move the table to where you want it on the page, and you may need to resize the table and fonts.



Save this file as a PowerPoint presentation. Then you can either add the rest of your slides to this file, or import this file into another presentation.

➢ CONCLUSIONS

This paper has shown a number of tips and tricks for taking your TABULATE output to the next level. This procedure can be a powerful table generator if you take a little time to learn how to enhance your output.

There are many more ways to enhance your output. Check the SUGI proceedings for Tutorials and Coders Corner papers on the procedure. There are also more techniques in my book "PROC TABULATE by Example". Finally, you can probably figure out some new tricks on your own, based on the needs of your organization.

➢ ACKNOWLEDGEMENTS

➢ CONTACTING THE AUTHOR

Please direct any questions or feedback to the author at: info@laurenhaworth.com

Creating Adobe Pdf Files from SAS Graph Output

Patrick M. McGown, FSD Data Services, Inc., Winston-Salem, NC

Introduction

As more companies and individuals move online, the demand for reporting information electronically will continue to increase. The need to provide information electronically is confounded by the variety of hardware and software platforms in use. This variability in systems can result in files being reformatted for a particular device or resolution, preventing a standard presentation of the information to the audience. Adobe Portable Document Format (Pdf) provides a method for ensuring standard presentation of information across the different hardware and software used by the end users. This paper discusses the different methods for creating Adobe Pdf files from SAS Graph output. For this paper, all files and graphs were created on a Windows NT 4.0 system with Adobe Acrobat and Reader 4.0 installed.

PDF files, what are they good for?

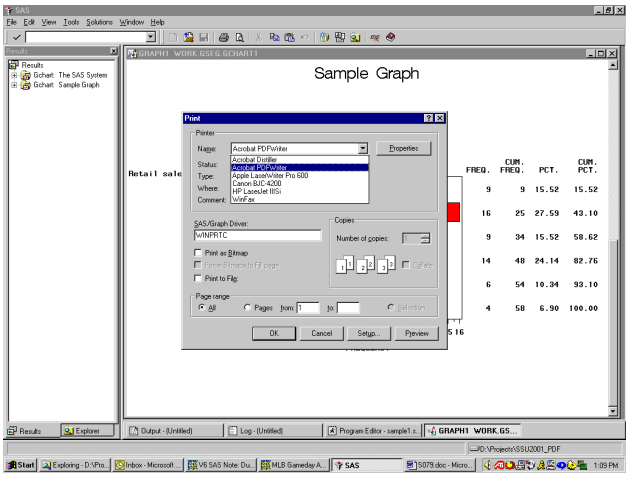
Pdf files are used with increasing frequency for on-line forms, documents, manuals and reports. They can be found in many places including the SAS Online Documentation and IRS Tax forms and publications. The Adobe Portable Document Format provides a platform for ensuring the standard presentation of information across different hardware and software systems. Once created, the Pdf file retains all of the fonts, graphics, colors and formatting regardless of the platform. By downloading and installing the free Adobe Acrobat Reader, most anyone can open Pdf files. The Adobe Acrobat Reader is available for systems such as Windows, Macintosh, Linux, Solaris, Unix, IBM AIX, OS2, Sun SPARCstation and the recently released version for Palm OS 3.1 or higher. The ability to provide individual users with the same file regardless of their system saves time and money and greatly simplifies the publishing process.

Creating Pdfs using SAS Version 6

Version 6 does not include any Pdf drivers, requiring a multi-step process for generating Pdf files from SAS Graph output. Both of the processes discussed here require the installation of Adobe Acrobat, software used to generate Pdf files from other systems. Once installed, Adobe Acrobat provides two methods for placing SAS Graphs into Pdf files, the Adobe PDFWriter print driver and the Adobe Acrobat Distiller.

Creating Single Page PDF files

For creating single page file from a single SAS Graph, the process includes running the graph, selecting File/Print from the SAS menu bar, then selecting the Adobe PDFWriter under the printer name list, and providing a path and filename when prompted.



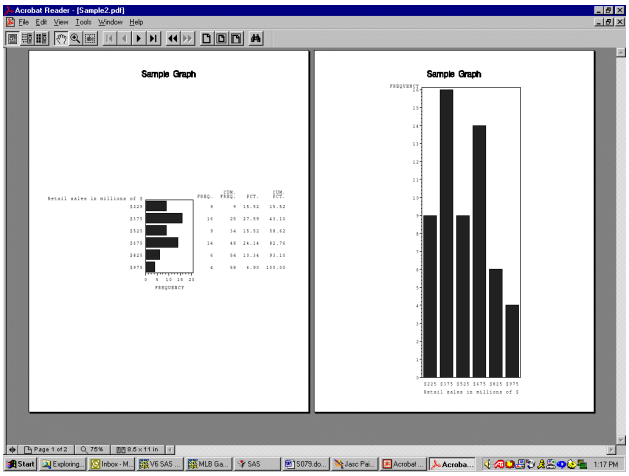
If you have several single page Graphs to process, this can take some time to do and for large scale processing, it is not an efficient process.

Creating Multi-Page PDF Files

Creating a single Pdf file from multiple SAS Graphs requires a different process. This method may also be used if the job requires numerous single page files, making the above method too time consuming. This requires the graphs to be saved as Postscript files and then translated into Pdf files using the Adobe Distiller. The following program creates two graphs from SASHELP.RETAIL with the output being saved to a Postscript file named Sample2.ps. The filename statement assigns a Graphics Stream File destination for the output and the GACCESS option directs the output to the postscript file. GSFNAME may be substituted for GACCESS. The GSF file will contain the SAS Graphs exported in the format specified by the device listed in the goptions statement. In this case, the graphs are being exported using the 300 DPI Postscript Driver. In order to export multiple graphs to a single file, GSFMODE must be set to APPEND.

```
filename gsasfile 'd:\projects\ssu2001_pdf\Sample2.ps';
goptions device=ps300 gaccess=gsasfile gsfmode=append;
proc gchart data=sashelp.retail;
title 'Sample Graph';
hbar sales;
vbar sales;
run;
quit;
```

Once the graphs have been export to the Postscript file, Adobe Distiller can be used to translate the Postscript into a multi-page Pdf file like the one below.



This method requires Adobe Distiller, part of the Adobe Acrobat package but it provides a powerful tool for generating numerous Pdfs from SAS Graph output, either single page or multi-page by dynamically creating the filerefs for storing the exported graphs. The following macro creates three two-page postscript files by using the macro variable FILENAME to assign the fileref for each file.

```
%macro graphs(filename);
  filename gsasfile "d:\projects\ssu2001_pdf\&filename..ps";
  goptions dev=ps300 gaccess=gsasfile gsfmode=append;
  proc gchart data=sashelp.retail;
    title "&filename";
    hbar sales;
    vbar sales;
    run;
   quit;
%mend;
%graphs(sample2a);
%graphs(sample2b);
%graphs(sample2c);
```

Once this program is run and the postscript files created, the Adobe Distiller can be set to watch the folder and it will process all postscript files within the folder. The advantage to this is that hundreds or thousands of postscript files can be created and the Adobe Distiller can translate them to Pdfs during off hours.

Creating Pdfs using SAS Version 7

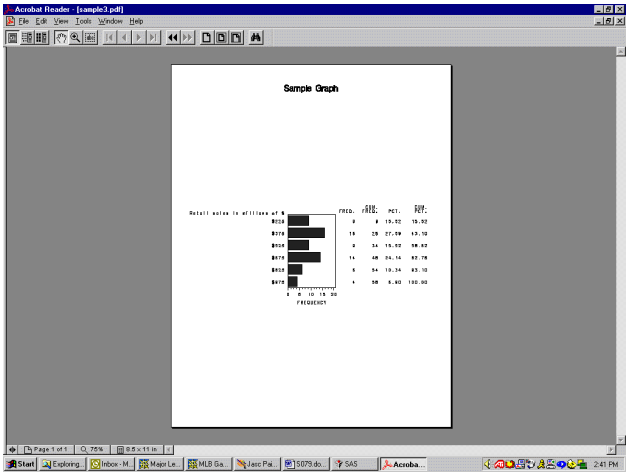
Version 7 of SAS introduces the PDF and PDFC (color) device drivers for creating Pdf output directly from SAS. Instead of specifying the postscript driver for the device, specify either of the two PDF drivers to create the file or files.

Creating Single Page PDF files

The following code creates a single page Pdf file from a single SAS Graph. The program is the same for the single postscript file but the device is set to PDF instead of PS300 and the file extension is PDF instead of PS.

```
filename gsasfile 'd:\projects\ssu2001_pdf\sample3.pdf';
goptions gaccess=gsasfile dev=pdf ;
 proc gchart data=sashelp.retail;
  title 'Sample Graph';
  hbar sales;
  run;
quit;
```

The resulting Pdf file looks like:

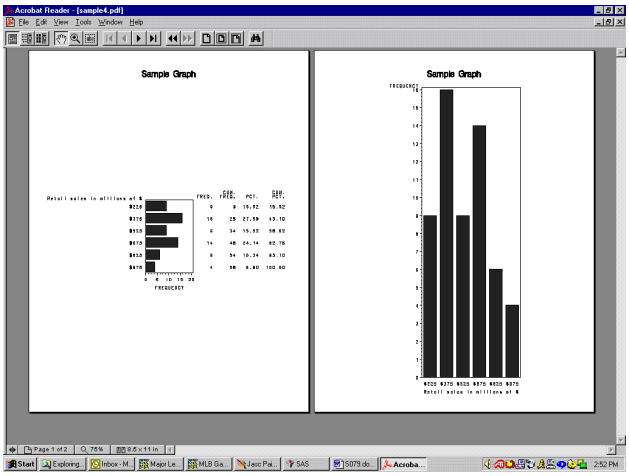


Creating Multi-Page PDF Files

The method for creating a multi-page Pdf from more than one SAS Graph is the same with one exception.

```
filename gsasfile 'd:\projects\ssu2001_pdf\sample4.pdf';
goptions reset=all gaccess=gsasfile dev=pdf target=ps300
gsfmode=append ;
 proc gchart data=sashelp.retail;
  title 'Sample Graph';
  hbar sales;
  vbar sales;
  run;
quit;
```

In this situation, a target device must be specified and it must contain a value other than Pdf. In the above example, the target device is the 300 DPI Postscript driver. If the target device is not specified or the target device is set to Pdf, an error will occur when viewing the second page and it will appear blank. Using the above code to create a two page Pdf file from the two graphs with the device target set to PS300 results in the following Pdf file. This method works when the graphs are produced under the same SAS Graph procedure.

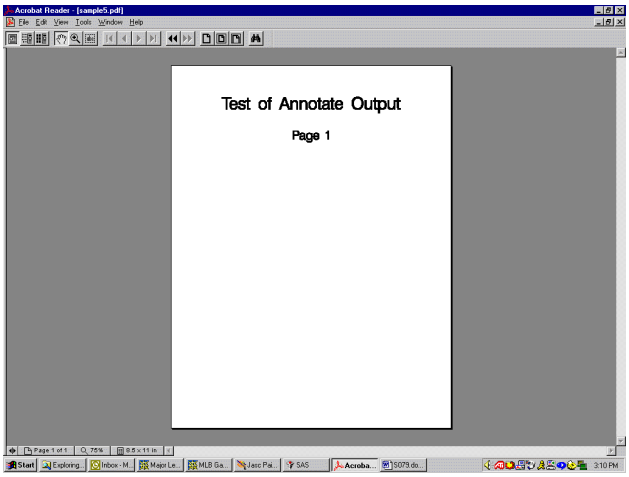


An exception to the rule: SAS Graph Output from Multiple SAS Graph Procedures

There is a problem with using the Pdf device driver to create multi-page Pdf files from multiple SAS Graph Procedures. The following program creates a single page Graph output using one Proc Gslide procedure to display simple Annotate graphics.

```
filename gsasfile 'd:\projects\ssu2001_pdf\sample5.pdf';
%annomac;
goptions reset=all device=pdf target=ps300 gaccess=gsasfile
gsfmode=append  ;

data temp;
 length text $100 function color style $8;
 retain xsys ysys '3';
 %label(50,95,'Test of Annotate Output',black,0,0,2,swiss,5);
 %label(50,85,'Page 1',black,0,0,1.5,swiss,5);
run;
proc gslide anno=temp;
run;
quit;
```

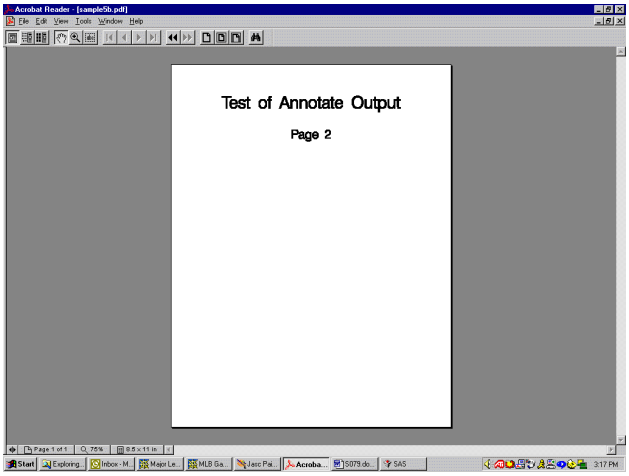
Below is the result of this program.



The problem arises when the output from more than one SAS Graph procedures are exported to the same Graphics Stream File using the PDF driver. If we put the code above into a macro and run the Proc Gslide twice to create a two page Pdf file, only the last page is created in the file.

```
filename gsasfile 'd:\projects\ssu2001_pdf\sample5b.pdf';
%annomac;
goptions reset=all device=pdf target=ps300 gaccess=gsasfile
gsfmode=append  ;
%macro test(page);
  data temp;
  length text $100 function color style $8;
  retain xsys ysys '3';
  %label(50,95,'Test of Annotate Output',black,0,0,4,swiss,5);
  %label(50,85,'Page '||left(trim("&page")),black,0,0,3,swiss,5);
  run;
  proc gslide anno=temp;
  run;
  quit;
%mend;
%test(1);
%test(2);
```

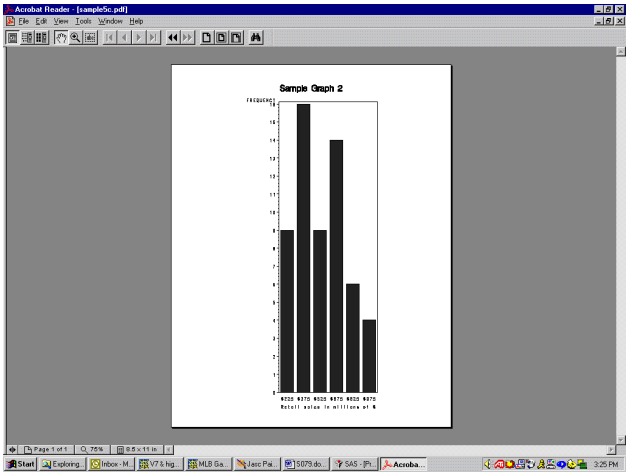
This programs creates two pages but only the second page is viewable in the Pdf file.



Likewise, running two different SAS Graph Procedures and exporting the results to the same Graphics Stream File produces the same result, only the last page is saved. For example,

```
filename gsasfile 'd:\projects\ssu2001_pdf\sample5c.pdf';
goptions reset=all gaccess=gsasfile dev=pdf target=ps300
gsfmode=append ;
 proc gchart data=sashelp.retail;
  title 'Sample Graph 1';
  hbar sales;
 run;
 quit;
 proc gchart data=sashelp.retail;
  title 'Sample Graph 2';
  vbar sales;
 run;
 quit;
```

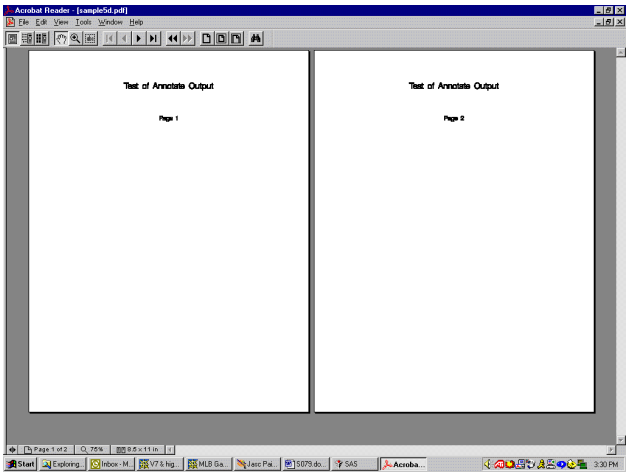
This program results in a Pdf file that only contains the vertical bar chart run in the second procedure.



SAS Note SN-000918 deals with this issue and provides a solution to the problem.

```
filename gsasfile 'd:\projects\ssu2001_pdf\sample5c.pdf';
%annomac;
goptions reset=all;
goptions nodisplay device=pdf ;
%macro annotest(page);
data temp;
 length text $100 function color style $8;
 retain xsys ysys '3';
 %label(50,95,'Test of Annotate Output',black,0,0,2,swiss,5);
 %label(50,85,'Page '||left(trim("&page")),black,0,0,1.5,swiss,5);
run;
proc gslide anno=temp;
run;
quit;
%mend;
%annotest(1);
%annotest(2);
goptions display gaccess=gsasfile gsfmode=append;
proc greplay nofs;
 igout work.gseg;
replay _all_;
run;
quit;
```

In this situation, the output from the two separate Gslide procedures are stored in the work.gseg catalog and then replayed to the Graphics Stream File using a single SAS Graph Procedure, Proc Greplay. This allows both pages to be created correctly in the Pdf file.



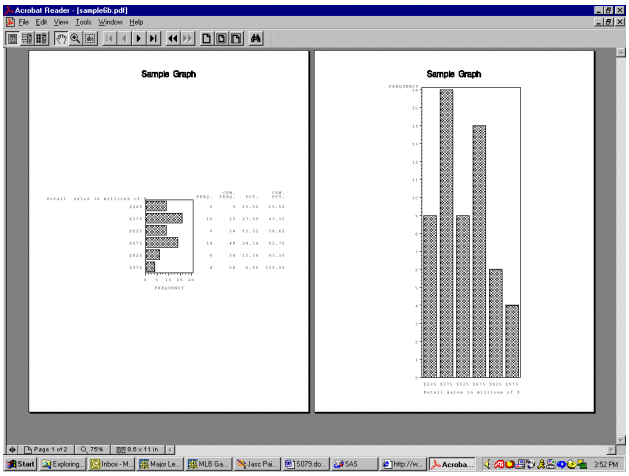
The caution here is that if multiple files are to be created using this method, the work.gseg catalog needs to be deleted before each file is created otherwise the next file will contain the graph output from the current output as well as the previous output.

Creating Pdfs using SAS Version 8.01

The major difference between Version 7 and Version 8.01 is that for creating multi page files from a single SAS Graph Procedure, the target device is no longer needed. The two page Pdf file can be created without specifying the target device.

```
filename gsasfile 'd:\projects\ssu2001_pdf\sample6b.pdf';
goptions reset=all gaccess=gsasfile dev=pdf  gsfmode=append ;
proc gchart data=sashelp.retail;
 title 'Sample Graph';
 hbar sales;
 vbar sales;
run;
quit;
```

Taking out the target=PS300 has no effect on the exporting of the graphics output except that the bars are shaded instead of solid.



As in Version 7, if the graphics output is generated by multiple SAS Graph procedures, the Proc Greplay method described above must be used.

Creating Pdfs using SAS Version 8.2

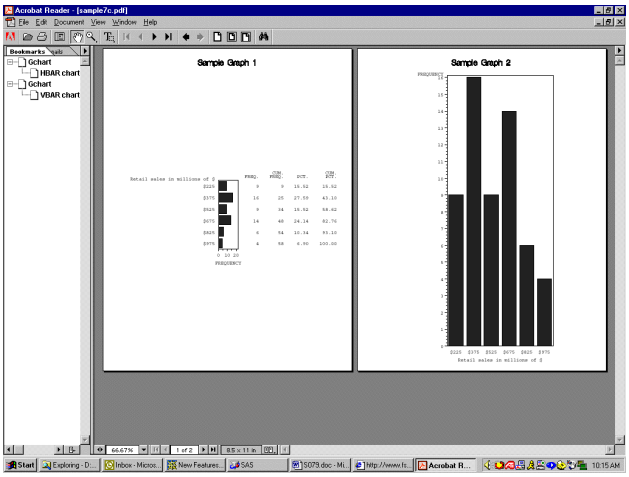
In Version 8.2, the multiple SAS Graph Procedure output to a single Graphics Stream File problem using the Pdf driver still exists. The Adobe Distiller method or the Proc Greplay method are still options in Version 8.2 using the Pdf driver.

Last but not least, ODS

In Version 8.1, ODS provided a method for exporting output to Pdf files using ODS PRINTER PDF but only under certain conditions. This was experimental in 8.1 but is production in Version 8.2. This method allows for the exporting of output from multiple SAS Graph Procedures to a single Pdf file.

```
ods printer pdf file='d:\projects\ssu2001_pdf\sample7c.pdf';
goptions target=winprtg;

 proc gchart data=sashelp.retail ;
  title 'Sample Graph 1';
  hbar sales ;
 run;
 quit;

 proc gchart data=sashelp.retail ;
  title 'Sample Graph 2';
  vbar sales ;
 run;
 quit;
ods printer close;
```

This program creates the following file.



There are a couple of details worth mentioning. In order to force the bars to be black, the goptions statement with target=winprtg was added to format the output based on the Windows gray scale print driver. The second detail to note is when ODS creates the Pdf file, it also creates bookmarks, which are located in the far left column of the file above. In Pdf files, the bookmarks allow for moving from page to page by clicking on any given bookmark. The bookmark display can be hidden in Adobe Reader under Window\Hide Bookmarks. ODS PRINTER PDF provides a method for the direct creation of multipage Pdf files from multiple SAS Graph Procedures.

Conclusion

This paper provides a simple overview of creating Pdf files from SAS Graph Output. The ability to create Pdf files from SAS Graph Output provides a powerful tool for distributing information electronically in a consistent format. The flexibility and power of SAS linked with the universal nature of the Portable Document Format provides a way to generate thousands of documents that can be made available on the web or distributed in other electronic medium.

References

Contacting the Author

Please direct any questions or feedback to the author at:

FSD Data Services, Inc.
1001 S. Marshall Street
Suite 125, Box 25
Winston-Salem, NC 27101

E-mail: patrickm@fsddatasvc.com

Behind the Scenes at SAS-L

Francis Joseph Kelley
University of Georgia, Athens, GA

Abstract: Since the Fall of 1986, there has been a Listserver-based discussion group called "SAS-L". For almost 15 years it has been an active, sometimes argumentative, group discussing everything from the conditional probability of selecting a prize-winning "door" to correct JCL to sound programming practices to search algorithms to the SAS macro language to good programming editors to ... Well, the list continues, seemingly without end. This is a discussion group that one participant once likened to "asking a colleague down the hall." This paper isn't about that. This paper is about why the mail sometimes doesn't go through, about using the SAS-L archives to help answer questions (and how to do so more effectively), about strangely-behaving mailers. It is also about some plans and thoughts for the future of SAS-L.

Dynamically Instantiating Widgets on SAS® Frames – Why, How, and When A Sample

David Ward, InterNext, Inc., Somerset, NJ

ABSTRACT

SAS/AF Frames can be a quick and powerful way to build SAS-based application in your organization. There may be times when you need your users to be able to build frames dynamically, for example, to build data entry screens. It is possible to use the work area class from version 6 to build a run-time frame “editor” by which you can allow users to add/remove/move widgets and set their properties. A frame can then be written which will dynamically instantiate widgets the user has built at design-time. The parent frame can even add and change methods and properties of those widgets as needed. Presented in this paper is a data entry application that allows users to build screens and uses a SAS/AF Frame to draw the screens at run-time. This technique can provide a wealth of rich functionality to your SAS/AF applications, especially if you need your users to design part of your application.

INTRODUCTION

The ability to dynamically instantiate widgets on your SAS/AF frames comes from two SCL operators: _new_ and _neo_. According to the SAS OnlineDoc version 8, _new_ "creates an object and runs an associated class constructor". The class constructor is also referred to in AF-speak as the _init method. This method is called automatically, just like the _term method when an object is deleted. Again, according to OnlineDoc, "the _NEO_ operator provides a faster and more direct way to create an object. It combines the actions of loading a class with LOADCLASS and initializing the object with the _new method, which invokes the object's _init method".

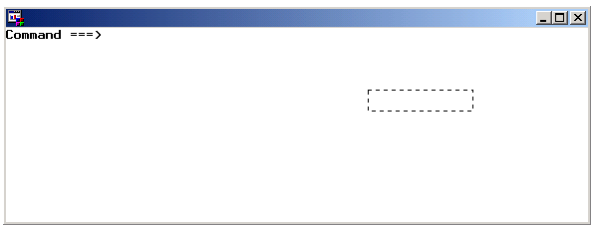
YOUR FIRST EXPERIMENT

To get an idea of what creating widgets at run time looks like, create a test frame named sasuser.widgets.test1.frame. Do not create any widgets; simply go directly to the source code for the frame. Enter the following code:

```
Import sashelp.classes;
Dcl textentry_c text1 = _new_ textentry_c();
init:
return;
```

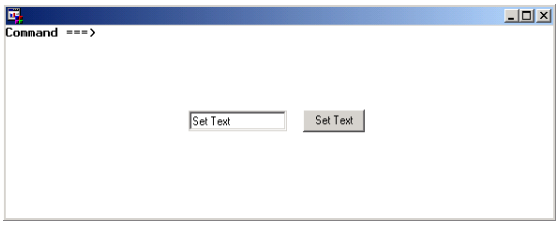
The code above creates a new object named text1 when the frame is executed. You may have seen this syntax when creating non-visual objects such as the logical, library, or catalog classes in the catalog sashelp.fsp. In version 6, SCL used the instance() and loadclass() functions. The _new_ operator provides a more intuitive syntax to do the same thing. In this example, however, we are creating a new instance of the text entry class, the same class used to create text boxes on frames.

Next, run the frame. Immediately you will see the outline of a text entry box that follows your mouse pointer around the frame. Clicking anywhere on the frame displays your text entry in the location where you clicked. You can now reference the text entry box in the same way you always have.



We can add a button to the frame at design time that sets the text property of our text entry box. Once the widget is created, clicking on the button will display the text.

```
button1:
  text1.text='Set Text';
return;
```

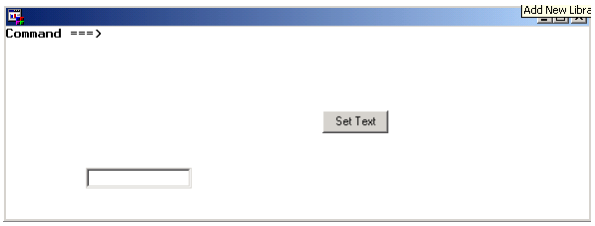


SETTING THE LOCATION OF THE WIDGET

The previous example would prove frustrating to end-users. Each time they ran the frame they would be forced to decide where to place the text entry box. As a developer, you would probably not want to give that kind of control to your users either. The solution is to pass the _neo_ operator a list of region attributes so that the text box will know how to "draw" itself. The frame SCL should now be:

```
Import sashelp.classes;
Dcl textentry_c text1;
init:
  startcol=10;  startrow=10;
  AttrList={};
  RegionList={};
  rc = setniteml(AttrList, RegionList,
'_region_');
  rc = setnitemn(AttrList, -1, 'num');
  rc = setnitemn(RegionList, startcol,
'ulx');
  rc = setnitemn(RegionList, startrow,
'uly');
  text1=_neo_ textentry_c(AttrList);
return;
button1:
  text1.text='Set Text';
return;
```

We have added code to create an attribute list with two parameters, ULX and ULY. These correspond to the column/row position that the text entry control should be created at.



Once your widget has been created and is placed on the screen you can set other desired properties. Any editable property (see the OnlineDoc or the properties window of the frame builder) can be accessed. If you wanted to change the color of the text entry box you would write:

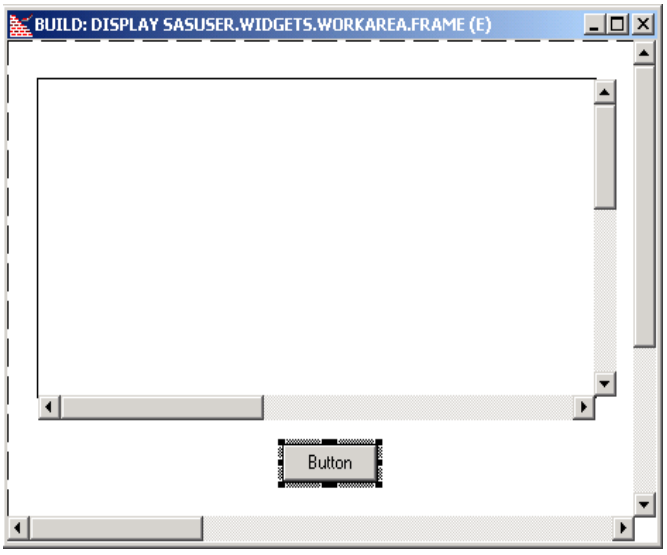
```
text1.backgroundcolor='red';
```

Of course, make sure you place this code after the _neo_ statement.

SAVING WIDGET PROPERTIES

Typically, developers would use this kind of functionality to allow users to build or customize part of an application themselves. For example, you could allow your users to drag and drop and set properties for a welcome frame or main menu. This way, they could customize images, widget sizes, etc. But how can we save the information each user creates about where they would like their widgets placed and what properties they would like to set? The answer is the work area control from version 6. It allows users to create and manipulate widgets at run time and to save the entire area out as an SCL list.

Create a new frame called sasuser.widgets.workarea.frame. Place a simple workarea in the frame and add a push button anywhere. Your frame should now look something like this:



Enter the following SCL code for the frame:

```
pushbutton1:
  widgets=makelist();
  call notify ('obj1','_get_widgets_',widgets);
  rc=putlist(widgets,'',0);
  rc=dellist(widgets,'Y');
return;
```

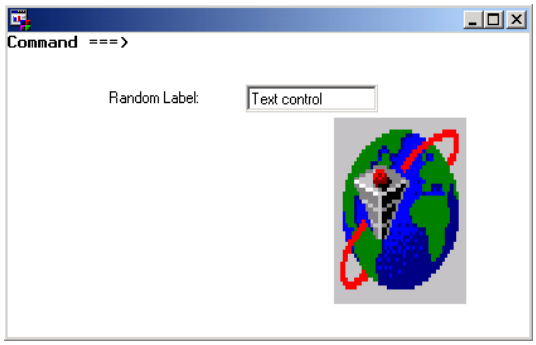
After running the frame, you will notice that you can right-click anywhere inside the workarea control and choose the option "add item". Doing so shows you a list of all built-in SAS widgets (or controls). Choosing the text entry control will display an outlined control much like in our first example. It is up to you to decide where to place it. Once you have created a widget, right-clicking on the widget and choosing "properties" will allow you to set design-time properties of the widget. Once you have created some widgets and set their properties click on the push button. In your log you should see a rather large list containing one sublist per widget. The list contains all information about each widget, even the ULX and ULY values we looked at earlier. You can use the _region_ sublist in conjunction with the _neo_ operator to create exact copies of the designed widgets on a frame.

USING THE WORKAREA LISTS

After saving the list you viewed in our last example, you need to use it in a practical way to generate widgets. Simply using the workarea again is not practical because users can make changes and you cannot easily manipulate widgets inside programmatically. The answer is to build a new frame that reads the widget list and creates a new widget for each in the list. Here is some code to get you started:

In this case we need to use the "old-fashioned" instance() and loadclass() functions because the type of widget is not known at compile time. Notice that each time we used _neo_ we had to hard-code the widget type (textentry_c above).

Here is a screen shot of our simple example and the SCL code used to create it:



And the SCL source code:

```
Dcl object widget;
Dcl char classname;
init:
  widgets={};
  rc=fillist('catalog',
   'sasuser.widgets.workarea.slist',
    widgets);
  do i = 1 to listlen(widgets)-2;
    * LAST 2 ARE NOT WIDGETS *;
    sublist=getiteml(widgets,i);
     region=getniteml(sublist,'_REGION_');
     ** REMOVE BORDER ATTRIBUTE NONE **;
     rc=delnitem(region,'border_style');
     classname=getnitemc(sublist,
       '_CLASSNAME_');
    classname='sashelp.classes.'||
      scan(classname,1,'.');
     widget=instance(loadclass(classname),
       sublist);
  end;
return;
```

This code exactly reproduces what the user designed in the workarea.

NEXT STEPS

In order to use your instantiated widgets effectively, you will need to store the object identifiers in SCL variables or in an SCL list. If you have an undetermined number of widgets you should use an SCL list with the inserto(), getitemo() functions. This way, you can programmatically set properties of the widgets. Additionally, you can use the _addEvent, _addAttribute, etc. functions to trap events, override methods, or do anything else that you would normally do at design time. This way you can execute code when a user uses these new widgets.

In the presentation of this paper a data entry screen builder will be demonstrated that uses these techniques to allow the design of potentially complex screens.

CONCLUSION

In this paper you have been exposed to a technique to dynamically instantiate widgets on AF frames. This is a powerful way to let your users customize or design part of your application and can be used for such things as building data entry screens.

ACKNOWLEDGMENTS

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

David L. Ward
InterNext, Inc.
254 Resnik Ct.
Somerset, NJ 08873
Work Phone: (732) 745-9823
Email (preferred): dward@sashelp.com

Proc Format, a Speedy Alternative to Sort/Merge

Jenine Eason, AUTOTRADER.COM, Atlanta, Georgia

ABSTRACT

Many users of SAS System software, especially those working with large data sets, are often confronted with several challenges. How can one reduce the data set size and reduce the amount of time required retrieving specific data? In this paper, we attempt to do both using a matching method utilizing Proc Format to replace the CPU heavy Sort/Merge. It is ideal for situations when a key from one file is needed to extract data from another file. It is more apparently useful when at least one of the files is quite large. This method has been proven time and again to decrease CPU by 70 – 80%. This paper is intended for the intermediate to advanced SAS user. It is effective on all platforms and uses Base SAS.

INTRODUCTION

Many users of SAS System software, especially those working with large data sets, are often confronted with several challenges. How can one reduce the data set size and reduce the amount of time required retrieving specific data? In this paper, we attempt to do both using a matching method utilizing Proc Format to replace the CPU heavy Sort/Merge. It is ideal for situations when a key from one file is needed to extract data from another file. It is more useful when at least one of the files is quite large. This method has been proven time and again to decrease CPU by 70% - 80%.

First, let's look at the traditional matching routine Sort/Merge.

TRADITIONAL SORT/MERGE

```
Proc Sort data=key(keep=key) nodupkey;
  By key;
Run;

Proc Sort data=bigfile;
  By key;
Run;

Data all; merge   bigfile(in=a)
                  key(in=b);
  By key;
  If a and b;
Run;
```

Sort/Merge is used here when key values from one file are needed to extract records from another file with the same key. For a clean merge, both data sets have to be sorted. Note that both files have to be read twice. If either or both the files are large, CPU time can be considerable.

There are other basic concerns when running any Sort/Merge. Are there any duplicate records in either data set being used in the merge? Is the "king" merge logic being handled properly, so that required data from matching data sets will not be accidentally overlaid? These are no longer concerns with the Proc Format method.

SPEEDY METHOD USING PROC FORMAT

```
Data key; set keyfile(keep=key);
   rename key = start ;
   Fmtname = '$key';
   Label = '*';
Run;

Proc Sort data=key nodupkey;
   By start;
Run;

Proc Format cntlin=key;
Run;

Data all; Set bigfile;
   If put(key,$key.)='*' ;
Run;
```

First, a SAS data set needs to be created from the input file with the required format variables LABEL, START, and FMTNAME.

- START is set to the variable assigned as the key.
- FMTNAME becomes the format name to be referenced later. (cannot end in a numeric)
- LABEL becomes the symbol that the desired key values are associated with.

It is very important that the symbol assigned to LABEL will never have an occurrence in the key character string of the master file. Otherwise, an unwanted hit will occur. The asterisk (*) symbol works in most situations.

This pre-format data set needs to be sorted and any duplicates eliminated. Format will not allow duplicate values and will present a "this range is repeated" error. This also holds true for version 8 SAS.

To actually create a working format, execute PROC FORMAT using the CNTLIN= option. It converts the data stored in the pre-format SAS data set to a SAS format.

Example of a format created using this method:

```
data keyfile;
  infile cards;
  input key $8.;
  cards;
Ohio
Georgia
Idaho
Virginia
;
run;

~~~~~~  code creating format ~~~~~~

proc format library=work fmtlib;
 select $key;
run;
```

```
+-----------------------------------------------------+
|FORMAT NAME:$KEY  LENGTH: 1  NUMBER OF VALUES:  4    |
|MIN LENGTH: 1  MAX LENGTH:40  DEFAULT LENGTH: 1      |
|--------------_--------------------------------------|
|START    |END     |LABEL (VER. 8.230APR2001:14:04:25)|
|-----------------------------------------------------|
|Georgia  |Georgia |*                                 |
|Idaho    |Idaho   |*                                 |
|Ohio     |Ohio    |*                                 |
|Virginia |Virginia|*                                 |
+-----------------------------------------------------+
```

This format can be used anywhere in a program for selecting matches to the key. In essence, the assignment statement gives the value of LABEL (in this case *) to a matching key. This in turn can be used for additional coding. In the above script, it is used to select records matching the key.

Using the Proc Format method, only one file is processed twice and only one variable is needed from it. Sorting of the master file is not required.

ALTERNATIVE METHODS COMPARING EXACT RESULTS

For the non-believers, the results below show several different methods for comparison. The same input files were used in all examples. All tests were run using the same Unix Sun platform on Version 6 SAS. The key file had 730 observations. The larger file has over 1.5 million records. For these examples, the following CPU times were recorded.

Comparing the Proc Format method to the traditional Sort/Merge; the additional CPU time required to create the pre-format data set and create the format is more than made up by avoiding the pre-sort of the large file. The selection data step instead of the merge is 14.4% more efficient and the results are exactly the same.

	CPU	% time reduction
Proc Format Method	10.267sec.	-
Sort/Merge	42.823sec.	76%
Indexing	8.000sec.	73%

ADDITIONAL USES

Merging using more than 1 Variable

Frequently, merging requires more than one variable. Two solutions would include concatenation or the use of more than one format. The examples below show the combination of key1 and key2 as the selection criteria.

2 Formats Method

```
data keya; set keyfile(keep=key1);
  start = key1;
  fmtname = '$keya';
  label = '*';
run;

proc sort data=keya nodupkey;
  by start;
run;

proc format cntlin=keya;
run;

data keyb; set keyfile(keep=key2);
  start = key2;
  fmtname = '$keyb';
  label = '*';
run;

proc sort data=keyb nodupkey;
  by start;
run;

proc format cntlin=keyb;
run;

data all; set bigfile;
  if put(key1,$keya.) = '*'  and
     put(key2,$keyb.) = '*';
run;
```

Concatenation of 2 Variables Method

```
data key; set keyfile;
  start = trim(key1)||trim(key2);
  fmtname = '$key';
  label = '*';
run;

proc sort data=key nodupkey;
 by start;
run;

proc format cntlin=key;
run;

data all; set bigfile;
  if put(trim(key1)||trim(key2),$key.) = '*';
run;
```

Using a Numeric Variable as a Key

```
data key; set keyfile;
  start = key;
  fmtname = 'key';
  label = '*';
run;

proc sort data=key nodupkey;
 by start;
run;

proc format cntlin=key;
run;

data all; set bigfile;
  if put(key,key.) = '*';
run;
```

Many other solutions could be used to perform special processes only for those records matching the key or eliminating records that match the key.

- Make the LABEL more meaningful.

  (In first data step)

  label = ‘Match’;

- Creating more than one format using unique keys.

  (In first data step)

  ```
  if key = ‘one’ then label = ‘one’;
  if key = ‘two’ then label = ‘two’;
  ```

  (In last data step)

  ```
  data testone testtwo; set bigfile;
    if put(key,$key.)='testone' then output
      test one; else
    if put(key,$key.)='testtwo' then output
      testtwo; else
    delete;
  runl
  ```

LOGS FOR 3 VERSIONS FOR TIME SAVINGS COMPARISON

Proc Format Method LOG (Total CPU time 10.267)

```
data key; set key(keep=key);
    start = key;
    fmtname = '$key';
    label = '*';
    keep start fmtname label;
  run;

NOTE: The data set WORK.KEY has 730 observations and 3
variables.
NOTE: DATA statement used:
      real time          0.210 seconds
      cpu time           0.058 seconds

  proc sort data=key nodupkey;
    by fmtname start;
  run;

NOTE: 0 observations with duplicate key values were deleted.
NOTE: The data set WORK.KEY has 730 observations and 3
variables.
NOTE: PROCEDURE SORT used:
      real time          0.200 seconds
      cpu time           0.043 seconds


  proc format cntlin=key;

NOTE: Format $KEY has been output.
  run;

NOTE: PROCEDURE FORMAT used:
      real time          0.080 seconds
      cpu time           0.055 seconds


  data all; set bigfile;
    if put(key,$key.)='*';
  run;

NOTE: The data set WORK.ALL has 13705 observations and 9
variables.
NOTE: DATA statement used:
      real time          13.000 seconds
      cpu time           10.053 seconds
```

Sort/Merge method LOG (Total CPU time 42.823)

```
   proc sort data=key(keep=key) nodupkey;
     by key;
   run;

NOTE: 0 observations with duplicate key values were deleted.
NOTE: The data set WORK.KEY has 730 observations and 1
variables.
NOTE: PROCEDURE SORT used:
      real time          0.500 seconds
      cpu time           0.062 seconds

   proc sort data=bigfile;
     by key;
   run;

NOTE: The data set WORK.BIGFILE has 1548721 observations
and 9 variables.
```

```
NOTE: PROCEDURE SORT used:
      real time          40.480 seconds
      cpu time           31.013 seconds

  data all; merge bigfile(in=a)
                  key(in=b);
    by key;
    if a and b;
  run;

NOTE: The data set WORK.ALL has 13705 observations and 9
variables.
NOTE: DATA statement used:
      real time          15.890 seconds
      cpu time           11.748 seconds
```

Indexing Method LOG (Total CPU time 38.000)

```
proc datasets lib=home;
                              -----Directory-----

                             Libref:           HOME
                             Engine:           V612
                             Physical Name:    /home/jeason
                             File Name:        /home/jeason
                             Inode Number:     101001
                             Access Permission: rwxr-xr-x
                             Owner Name:       jeason
                             File Size (bytes): 8192

                              #  Name      Memtype  Indexes
                              -----------------------------
                              3  KEY       DATA
       modify key;
       index create key;
NOTE: Single index KEY defined.
  run;

NOTE: PROCEDURE DATASETS used:
      real time          3.550 seconds
      cpu time           0.185 seconds

data test1all; set bigfile;
                set home.key key=key;
    if _error_ = 1 then do;
       _error_ = 0;
    end;
    else output;
  run;

NOTE: The data set WORK.TEST1ALL has 13705 observations
and 9 variables.
NOTE: Compressing data set WORK.TEST1ALL decreased size
by 0.00 percent.
      Compressed is 2 pages; un-compressed would require 2
pages.
NOTE: DATA statement used:
      real time          39.330 seconds
      cpu time           37.757 seconds
```

REFERENCES

SAS Procedures Guide ver. 6. Page 282

Rick Aster and Rhena Seidman, Professional SAS Programming Secrets, Matching pp. 251-258

AUTHOR CONTACT INFORMATION

Jenine Eason
Autotrader.com
5775 Peachtree Dunwoody Road
Atlanta, Georgia 30342

Phone: (404) 843-7199
Email: Jenine.eason@autotrader.com
Home: Easonconsulting@aol.com

Using the SAS Annotate Facility for Creating Custom Graphs

Patrick M. McGown, FSD Data Services, Inc., Winston-Salem, NC

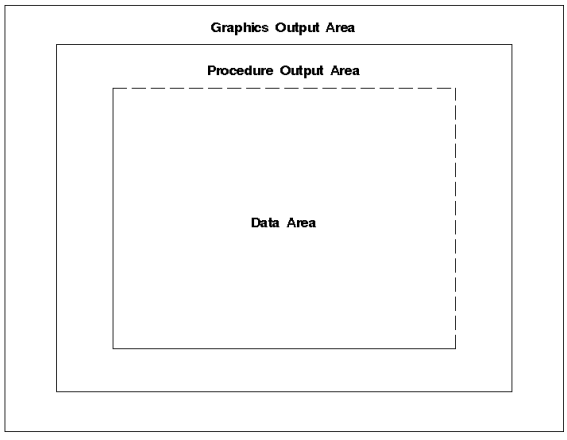
Introduction

With survey reporting, it is often necessary to produce graphical reports with pages of horizontal or vertical bar charts. Originally, these report applications used Proc Gchart to produce the graphs. However, when the customer changed the number of groups for a report, the application had to be modified to accommodate the new specifications. It became obvious that time was being wasted having to modify the report programs for different specifications. For that reason, it was decided to program the report applications using SAS Annotate. Combined with a macro loop, the system provided flexibility in placing different numbers of items and groups on any given page and being able to control all aspects of the formatting and presentation of the data.

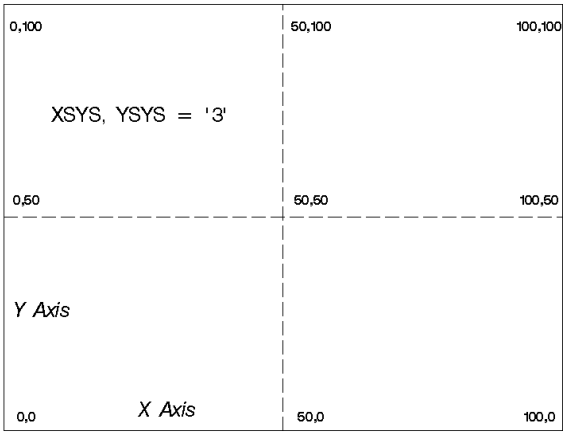
This paper will provide a brief introduction to the SAS Annotate Facility and some of its key features. The rest of the paper will work through building a sample graph from generating the summary data set to the final result.

Defining the Work Space

The features provided in the SAS Annotate Facility can be used either to modify graphics produced using the SAS Graph procedures or by itself to create custom graphics. This paper focuses on using Annotate as a separate tool. SAS defines three different regions on the page when producing graphics: the data area, the procedure output area and the graphics output area (pg. 476.) For the purpose of creating custom graphics, the graphics output area provides access to the entire page.



In addition, there is an absolute and relative coordinate system for each of the three areas. Within each area by system combinations, the coordinates can be percentage based or cell based. This paper utilizes the absolute percentage coordinate system in the graphics output area. In SAS code, this is specified by setting the coordinate system variables, xsys and ysys, to '3' (pp. 476-477.)



The numbers in the figure are percentages of the graphic output area of a landscape 8.5 x 11 page.

Starting the SAS Annotate Macros

Using the SAS Annotate Facility as a separate tool utilizes annotate macros. In order to use the macros, they must first be compiled. This is done by executing the following statement:

```
%annomac;
```

Once the macros have been compiled, the annotate data set can be created from the summary data set created earlier. A couple of statements need to be included in the data step, namely, the length and the retain statements. The text, function, color and style variables are used by the annotate macros and the retain statement sets the coordinate system to use for the macros.

```
data slide;
 set sketch;
length text $200 function color style $8;
retain xsys ysys '3';
```

The data step above doesn't contain any instructions for the Annotate Facility and would not produce any graphics. The next section will begin to add the macros that produce the graphics.

Using the Annotate Macros

This paper will discuss three of the annotate macros, %label, %line and %bar. There are several more that can be used but are not presented in this paper. The syntax for the three macros is:

```
%label(x1,y1,text,color,ang,rot,ht,font,pos);
%line(x1,y1,x2,y2,color,linetype,linewidth);
%bar(x1,y1,x2,y2,color,bartype,pattern);
```

The x and y values used in the macros will be absolute percentages of the graphics output area. These can be

expressed explicitly or stored in variables. The details of the syntax for the three macros follows.

```
%label(x1,y1,text,color,ang,rot,ht,font,pos);
```

The text field contains the text to be printed on the page with the formatting specified by the rest of the fields. 'Ang' specifies the angle of the text string from horizontal and the 'rot' field slants the string from vertical. 'Ht' defines the size of the text and font specifies the font. The 'pos' or position is the placement of the text relative to the x,y coordinates specified. There are 15 possible placements for the text (pg. 522.)

```
%line(x1,y1,x2,y2,color,linetype,linewidth);
```

The line macro is fairly straightforward with the x1,y1 values providing the starting point of the line and the x2,y2 values defining the end point. There are 46 different line types including solid, dashed, dotted and combinations of dashes and dots (pg. 429.)

```
%bar(x1,y1,x2,y2,color,bartype,pattern);
```

As with the line macro, the x1,y1 values give the starting point of the bar and the x2,y2 values provide the end point. In the case of the bar, the x1,y1 would be the lower left hand corner of the bar and the x2,y2 would be the upper right. There are four different bar type options, 0-3, with 0 providing a solid bar, 1 giving a vertically adjusted bar, 2 outputs a horizontally adjusted bar and 3 creates a bar without a border. The pattern variable can be solid, empty or crosshatched. The crosshatching is defined by R|X|L and the density of the hatching goes from 1-5 with 5 being most dense.

The %label, %line and %bar macros can be used to create custom bar charts and displayed using Proc Gslide.

Creating the Summary Data

In report applications, the summary data sets are generated either using SAS procedures or in data steps. The following code creates the summary data set used to create the sample graphs. Perfav and perunf represent percent favorable and percent unfavorable respectively with perneu being percent neutral. N represents the sample size for each group.

```
data sketch;
length group $15;
input n perfav perunf group $;
perneu=100-(sum(of perfav,perunf));
cards;
112 45 22  Professional
105 23 40  Administration
74  39 50  Manufacturing
68  47 15  Executive
215 36 40  Warehouse
98  25 38  Other
;
run;
```

One Time Details

In generating graphs using the SAS Annotate Facility, there are items that need to be created once and items that will need to be created for every observation in the summary data set. Features such as titles and page numbers need to be drawn only once while group labels and bars need to drawn for every observation in the summary data set. To create the one time details, the program executes those statements for only the first observation in the summary data set. See the example program below.

```
data slide;
 set sketch;
length text $200 function color style $8;
retain xsys ysys '3';
if _n_=1 then do;
        *Include one time statements here;
end;
        *Rest of annotate statements here;
run;
```

For example, if a title is needed for the graph, the following code can be used:

```
data slide;
 set sketch;
length text $200 function color style $8;
retain xsys ysys '3';
if _n_=1 then do;
 %label(50,95,'Title 1',black,0,0,3.5,swissb,5);
end;
run;
```

This code produces a data set that includes the commands from the macro statement to plot the title. To display the title, use Proc Gslide (pp.1261-1268) with the following syntax:

```
proc gslide annotate=slide; run;
```

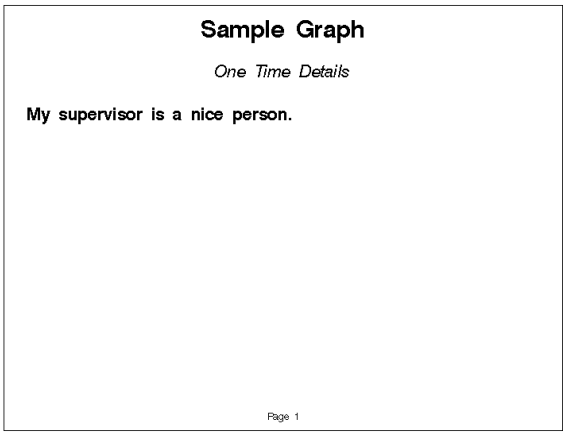
This statement performs all of the commands listed in the annotate data set and produces a slide based on those commands.

Title 1

If this statement was executed for all observations in the summary data set, the 'Title 1' data would be in six observations in the annotate data set and would be drawn onto the graph six times. This can take up space in the data set and time in resolving the graph.

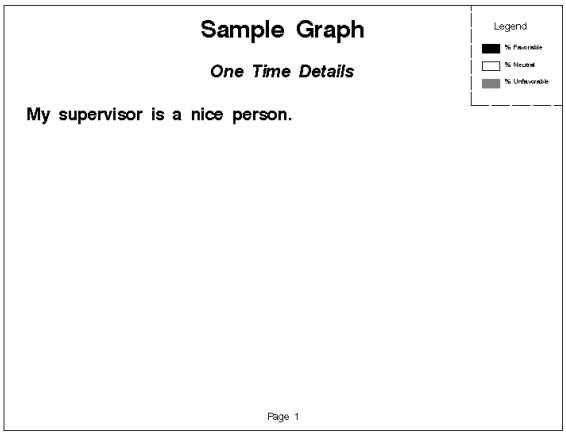
The following example provides a more complete program displaying features that only need to be processed once such as item text and page number.

```
data slide;
 set sketch;
length text $200 function color style $8;
retain xsys ysys '3';
if _n_=1 then do;
 %label(50,95,'Sample Graph',black,0,0,3.5,swissb,5);
 %label(50,85,'One Time Details',black,0,0,2.5,swissi,5);
 %label(5,75,'My supervisor is a nice
            person.',black,0,0,2.5,swissb,6);
 %label(50,5,'Page 1',black,0,0,1.5,swiss,5);
 %line(1,99,99,99,black,1,5);
 %line(1,1,1,99,black,1,5);
 %line(1,1,99,1,black,1,5);
 %line(99,1,99,99,black,1,5);
end;
run;
```



In addition to titles, page numbers and frame lines, items like legends can also be included in this block of code. Inserting the following code into the program in the do-loop above results in the legend in the upper right hand corner of the page.

```
*Legend Definition;

%label(90,95,'Legend',black,0,0,1.5,swiss,5);
%bar(85,90,88,88,black,0,solid);
%bar(85,86,88,84,black,0,empty);
%bar(85,82,88,80,gray,0,solid);
%label(89,90,'% Favorable',black,0,0,1,swiss,6);
%label(89,86,'% Neutral',black,0,0,1,swiss,6);
%label(89,82,'% Unfavorable',black,0,0,1,swiss,6);
%line(83,76,99,76,black,5,3);
%line(83,76,83,99,black,5,3);
```



Depending on the type of presentation desired, more or fewer details can be used.

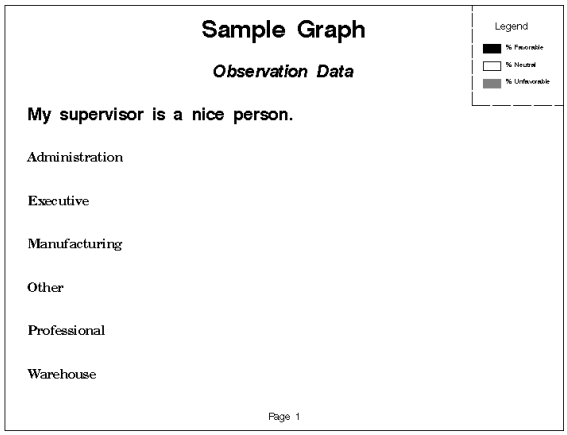
Multi-Observation Statements

The macro statements used to create the bars need to be executed for each observation in the summary data set. There are several components that make up the bars: the group label, the three bar components and the numeric labels for each part of the bar. Since these need to be run for each observation, the statements will follow the do-loop containing the one time details. One important part of this process is retaining the value of y as the statements are processed. To do this, the value of y is retained using a retain statement and a starting point is set using the y=75 statement prior to leaving the do-loop. The following code places the group labels down the left hand side of the page.

```
data slide;
 set sketch;
length text $200 function style color $8;
retain xsys ysys '3';
if _n_=1 then do;
 %label(50,95,'Sample Graph',black,0,0,3.5,swissb,5);
 %label(50,85,'Observation Data',black,0,0,2.5,swissbi,5);
 %label(5,75,'My supervisor is a nice
            person.',black,0,0,2.5,swissb,6);
 %label(50,5,'Page 1',black,0,0,1.5,swiss,5);
 %line(1,99,99,99,black,1,5);
 %line(1,1,1,99,black,1,5);
 %line(1,1,99,1,black,1,5);
 %line(99,1,99,99,black,1,5);

*Legend Definition;
 %label(90,95,'Legend',black,0,0,1.5,swiss,5);
 %bar(85,90,88,88,black,0,solid);
 %bar(85,86,88,84,black,0,empty);
 %bar(85,82,88,80,gray,0,solid);
 %label(89,90,'% Favorable',black,0,0,1,swiss,6);
 %label(89,86,'% Neutral',black,0,0,1,swiss,6);
 %label(89,82,'% Unfavorable',black,0,0,1,swiss,6);
 %line(83,76,99,76,black,5,3);
 %line(83,76,83,99,black,5,3);
 y=75;
end;
retain y;
y=y-10;
%label(5,y,group,black,0,0,2,centb,6);
run;
```

The code added to the previous program (underlined) sets the starting y value at 75 before ending the do-loop.

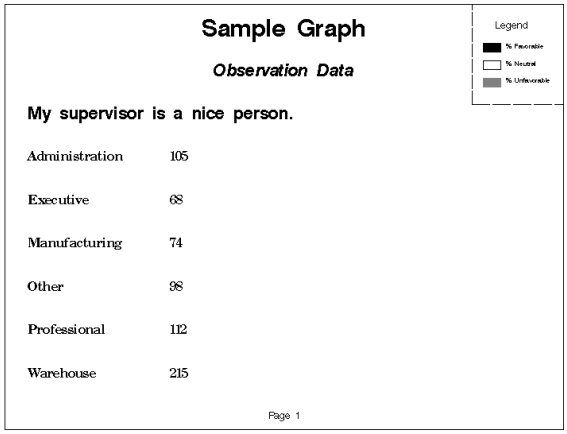
When SAS processes the next observation, the value of y goes to 65 and the label is printed on the page. The next observation will have a y value of 55 and continues until all of the observations in the summary data set are processed. The added code results in this graph:



By using the vertical coordinate value stored in the y variable, it takes only one label command to put the label for each of the six groups on the page. Likewise, by adding the following code to the above program after the last label macro statement, it will add the sample size for each group to the graph.

```
%label(30,y,left(put(n,3.0)), black,0,0,2,centb,6);
```

The text field is 200 characters long and without the 'left' function, the numbers would be plotted off the page to the right.
This results in the following graph:



Creating the Bars

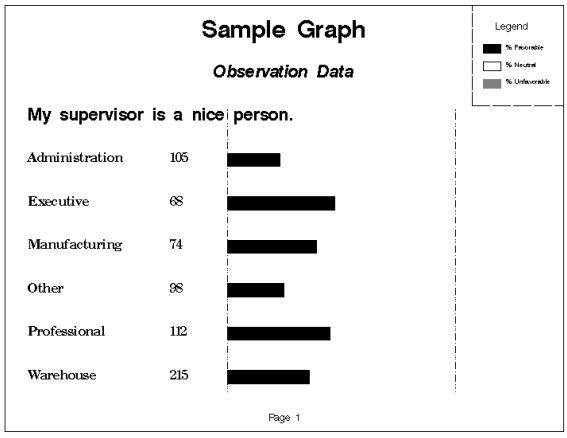
The first step in creating the bars is defining a plotting area for the bars. For this graph, the starting point will be x=40 and the finishing point will be x=80. This provides the system with 40% of the page width for plotting the bars. Determining the length of the bar segments is a matter of calculating the percentage of the plotting area for each segment. The statement for plotting the percent favorable bar is:

```
%bar(40,y,40+(40*(perfav/100)),
        y-3,black,0,solid);
y=y+3;
```

The first x value, 40, is the starting value of the plotting area. The second x value is the proportion of the plotting area for the percent favorable value. The equation for this value is:

```
40+(40*(perfav/100))
```

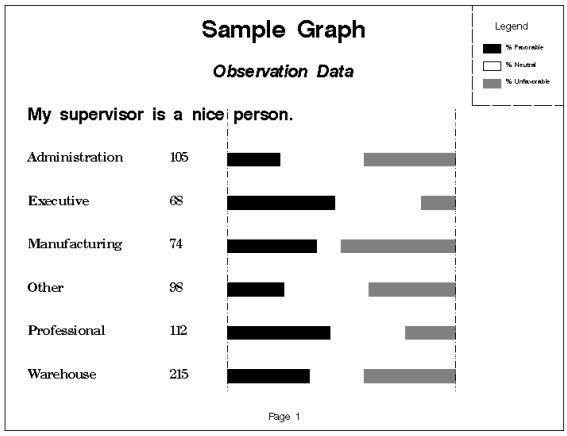
This equation places the perfav value on the scale of the plotting area and then adds that value to the starting value of the plotting area. The y=y+3; is added to maintain the spacing between the groups on the page. If this value was not added back in, the lower groups would plot off the page and the spacing between them would increase by 3%. The above statements added to the program result in:



The vertical dotted lines represent x=40 and x=80 and are put there for reference only. The bar for percent unfavorable is plotted in the same fashion but instead of starting at x=40 and adding some value, the bar will start at x=80 and subtract the scaled percentage of the plotting area. The statement for the percent unfavorable bar is:

```
%bar(80,y,80-(40*(perunf/100)),y+3, gray,0,solid);
*y=y+3;
```

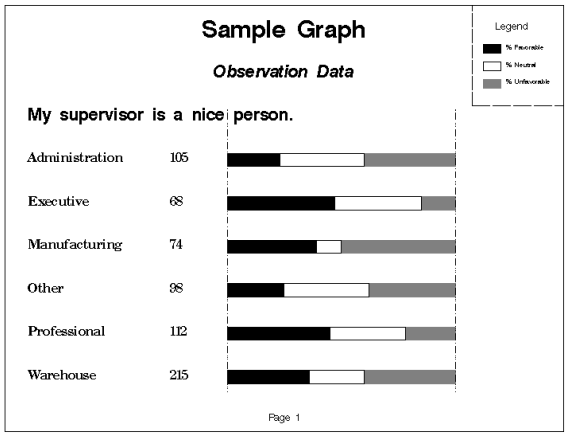
The y=y+3; is commented out since the 3 % is added back during this %bar statement. Adding this statement produces the following:



The final step in completing the bars is plotting the percent neutral bar in the middle. This statement uses both equations in determining the right and left hand side of the bar. The code for the percent neutral bar is:

```
%bar(40+(40*(perfav/100)),y,
        80-(40*(perunf/100)), y-3,black,0,empty);
y=y+3;
```

The first x value is the same as the x value calculated for the right hand side of the percent favorable (black) bar. The second value is the same as the x value calculated for the left hand side of the percent unfavorable (gray) bar. The 'empty' option will give a white box with black line borders. Once again, the y=y+3; must be used to maintain the spacing between the groups. The graph appears like this:

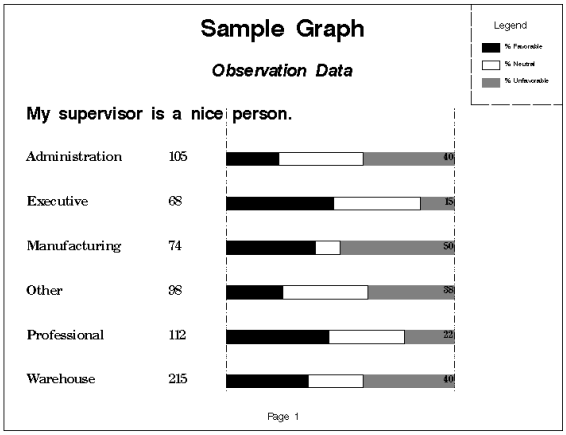


Adding the Labels for the Bars

The last step for completing the graph is inserting the percentage values into the bars. This is done in the same manner as calculating the start and end points for the bars. The easiest to plot is the percent unfavorable (gray) value. This value will be plotted to the right of the left hand edge of the gray bar. The statement is:

```
%label(80,y-.25,left(put(perunf,3.0)),
        black,0,0,1.5,centb,4);
```

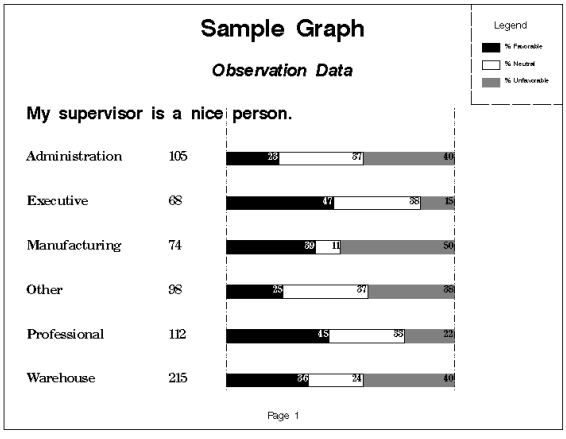
The .25 is subtracted from y to bring the number down a little from the top of the bar. Once again, without the 'left' statement, the text would print off the page since the text field is 200 characters long. The '4' position option places the label to the right of the x,y coordinate. The results is:



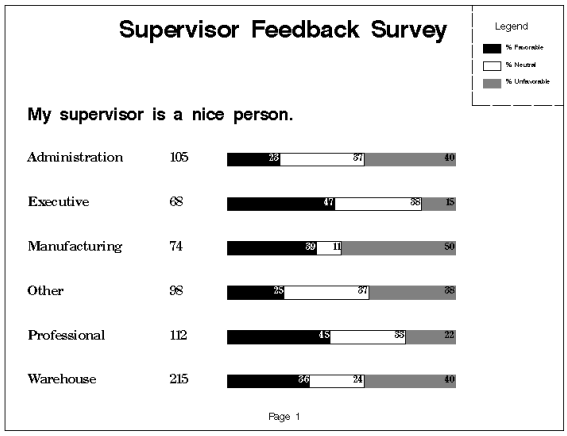
Adding the labels for percent favorable and percent unfavorable is done in the following code:

```
%label(40+(40*(perfav/100)),y,left
        (put(perfav,3.0)),white,0,0,1.5,centb,4);
%label(80-(40*(perunf/100)),y,left
        (put(perneu,3.0)),black,0,0,1.5,centb,4);
```

The x value for the percent favorable label uses the value for the right hand side of the bar and then the '4' position option used before plots the text to the left. The percent neutral statement uses the same syntax but is plotted at the left hand side of the percent unfavorable (gray) bar. The two statements added produces:



After removing the reference lines from the program and a title change, the final graph appears as:



The code to produce this final graph using the data set created earlier is:

```
data slide;
 set sketch;
length text $200 function style color $8;
retain xsys ysys '3';
if _n_=1 then do;
 %label(50,95,'Supervisor Feedback
          Survey',black,0,0,3.5,swissb,5);
 %label(5,75,'My supervisor is a nice
          person.',black,0,0,2.5,swissb,6);
 %label(50,5,'Page 1',black,0,0,1.5,swiss,5);
 %line(1,99,99,99,black,1,5);
 %line(1,1,1,99,black,1,5);
 %line(1,1,99,1,black,1,5);
 %line(99,1,99,99,black,1,5);

*Legend Definition;

 %label(90,95,'Legend',black,0,0,1.5,swiss,5);
 %bar(85,90,88,88,black,0,solid);
 %bar(85,86,88,84,black,0,empty);
 %bar(85,82,88,80,gray,0,solid);
 %label(89,90,'% Favorable',black,0,0,1,swiss,6);
 %label(89,86,'% Neutral',black,0,0,1,swiss,6);
 %label(89,82,'% Unfavorable',black,0,0,1,swiss,6);
 %line(83,76,99,76,black,5,3);
 %line(83,76,83,99,black,5,3);
 y=75;

end;
retain y;
y=y-10;
%label(5,y,group,black,0,0,2,centb,6);
%label(30,y,left(put(n,3.0)),black,0,0,2,centb,6);
%bar(40,y,40+(40*(perfav/100)),y-3,black,0,solid);
%bar(80,y,80-(40*(perunf/100)),y+3,gray,0,solid);
%bar(40+(40*(perfav/100)),y,80-(40*(perunf/100)),y-
          3,black,0,empty);
y=y+3;
%label(80,y-.25,left(put(perunf,3.0)),black,0,0,1.5,centb,4);
%label(40+(40*(perfav/100)),y,left(put(perfav,3.0)),white,0,0,1.5,c
          entb,4);
%label(80-
          (40*(perunf/100)),y,left(put(perneu,3.0)),black,0,0,1.5,c
          entb,4);
run;

proc gslide anno=slide;
run;
```

Adding Flexibility to the System

In this example, the six bars for the one item fit on one page, but in actual applications, most of the reports either put multiple items on one page or there are so many groups for each item, the items have to be spit between multiple pages. Using a macro variable to designate the total number of groups and a macro loop, it is possible to dynamically calculate the number of items and/or groups to put on each page.

Determining the Number of Groups

The first step in this process is to find out how many groups there are for each item. By running a frequency of the group variable and outputting the results to a datastep, the value of _N_ for the last observation will indicate how many groups there are and put this value to a macro variable.

```
proc freq data=sketch noprint;
tables group/out=counts;
run;

data _null_;
 set counts end=final;
  if final=1 then do;
   call symput('ngroups',_n_);
 end;
run;
```

The macro variable NGROUPS now has a value of 6, representing the total number of groups for each item.

Adding an Index Variable

In order to utilize the NGROUPS variable, a unique index value for each group is needed so that the macro loop can select the groups for each page. A quick way to accomplish this is to sort the data by the group variable and then create a variable called order to store the unique identifier. This will also result in the data being ordered alphabetically by group.

```
proc sort data=sketch;
 by group;
run;

data sketch;
 set sketch;
 by group;
 retain order;
 if _n_=1 then order=0;
 if first.group then order=order+1;
run;
```

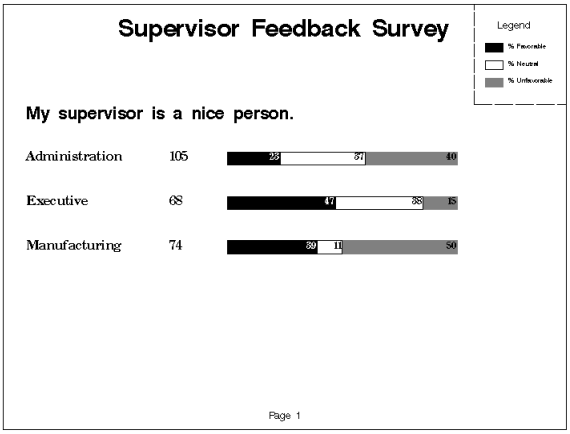
Creating the Macro Loop

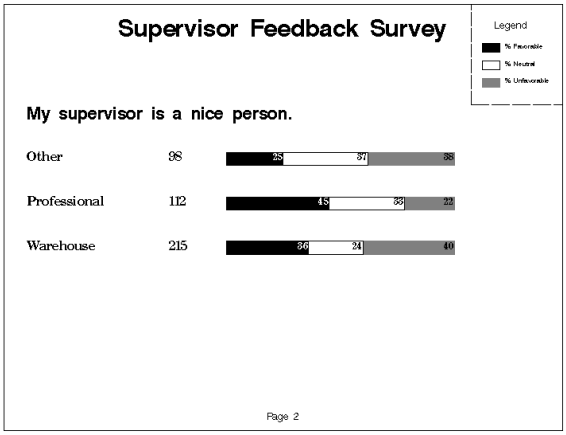
By utilizing a macro and a macro do loop along with the macro variables, MIN, MAX, NUMBER, NGROUPS, and PAGE, the system can dynamically put different bars on separate pages. NGROUPS is the number of groups determined by the frequency procedure in an earlier step and is equal to 6. The MIN variable stores the first observation in the range of observations to keep for each page. For the first page, MIN=1 and in the second page, MIN=4. MAX stores the last observation in the range of observations to keep for each page. For the first page,

MAX=3 and MAX=6 for the second page. The MIN and MAX values provide the range of valid values of the order variable to select for each page. The first page presents the groups where order equals 1, 2, or 3. The second page presents the groups where order equals 4, 5, or 6. NUMBER represents the number of bars per page minus 1. In this example, it is set to 2, resulting in three bars per page. PAGE stores a count of the pages and is used to put the page number at the bottom of the page using the Label macro in the program below.

```
%macro pages;
%let min=1;
%let max=;
%let number=2;
%let page=0;
%do %until(&max>=&ngroups);
data _null_;
call symput('max',&min+&number);
run;
data temp;
set sketch(where=(&min<=order<=&max));
call symput('page',&page+1);
run;
data slide;
set temp;
length text $200 function style color $8;
retain xsys ysys '3';
if _n_=1 then do;
 %label(50,95,'Supervisor Feedback
        Survey',black,0,0,3.5,swissb,5);
 %label(5,75,'My supervisor is a nice
        person.',black,0,0,2.5,swissb,6);
 %label(50,5,'Page '||left(trim("&page")),black,0,0,1.5,swiss,5);
 %line(1,99,99,99,black,1,5);
 %line(1,1,1,99,black,1,5);
 %line(1,1,99,1,black,1,5);
 %line(99,1,99,99,black,1,5);

*Legend Definition;

 %label(90,95,'Legend',black,0,0,1.5,swiss,5);
 %bar(85,90,88,88,black,0,solid);
 %bar(85,86,88,84,black,0,empty);
 %bar(85,82,88,80,gray,0,solid);
 %label(89,90,'% Favorable',black,0,0,1,swiss,6);
 %label(89,86,'% Neutral',black,0,0,1,swiss,6);
 %label(89,82,'% Unfavorable',black,0,0,1,swiss,6);
 %line(83,76,99,76,black,5,3);
 %line(83,76,83,99,black,5,3);
 y=75;
end;
retain y;
y=y-10;
%label(5,y,group,black,0,0,2,centb,6);
%label(30,y,left(put(n,3.0)),black,0,0,2,centb,6);
%bar(40,y,40+(40*(perfav/100)),y-3,black,0,solid);
%bar(80,y,80-(40*(perunf/100)),y+3,gray,0,solid);
%bar(40+(40*(perfav/100)),y,80-(40*(perunf/100)),y-
        3,black,0,empty);
y=y+3;
%label(80,y-.25,left(put(perunf,3.0)),black,0,0,1.5,centb,4);
%label(40+(40*(perfav/100)),y,left(put(perfav,3.0)),white,0,0,1.5,c
        entb,4);
%label(80-
        (40*(perunf/100)),y,left(put(perneu,3.0)),black,0,0,1.5,c
        entb,4);
run;

proc gslide anno=slide;
run;
quit;

data _null_;
call symput('min',&max+1);
run;
%end;
%mend;
%pages;
```

The above program results in the following two graphs.





As specified using the macro variable, the system plots three groups per page and the page number automatically increments by one. If the value of NUMBER were set to 1, the system would generate three pages with two groups per page.

New Feature in V8: Including an Image

In version 8, the Annotate Facility has added an image function, allowing for the inclusion of an image on SAS Graphs. The following code places an image called 'fsdlogo.gif' on the center of the page. The two x and y values define the area for the image to be placed. The image file is fit into the space defined by the x,y pairs.

```
x=40; y=65; function='move'; output;
x=60; y=35; imgpath="fsdlogo.GIF";
style = 'fit';
function='image'; output;
```

This slide was produced using the following program:

```
data slide;
 set sketch;
length text $200 function style color $8;
retain xsys ysys '3';
if _n_=1 then do;
  %label(50,95,'Sample of Image File',black,0,0,2,swiss,5);
  x=40; y=65; function='move'; output;
  x=60; y=35;
imgpath="d:\projects\ssu2001_anno\fsdlogo.GIF";
  style = 'fit';
  function='image'; output;
  %line(1,1,99,1,black,1,5);
  %line(1,1,1,99,black,1,5);
  %line(1,99,99,99,black,1,5);
  %line(99,1,99,99,black,1,5);

end;
run;
```

Limitations

While this technique is very useful in creating graphs that are completely customized, programming annotate graphs can be a time consuming process. The placement of objects and text may take many iterations of trying different values. This technique is also susceptible to exceptions that don't work with the above code like zero percentage values and text strings that overlap bars. These exceptions can be solved through more detailed programming.

Conclusions

This paper examined a technique using the SAS Annotate macros for creating custom graphs. While the programming can be time consuming, once the code is finalized, you can have a powerful tool for creating graphs that are completely customizable and can end the frustration of trying to label, place titles and create legends in the SAS Graph Procedure output.

References

SAS Institute Inc. (1990), *SAS/GRAPH® Software: Reference, Version 6, First Edition, Volumes 1 & 2*, Cary, N.C.: SAS Institute

Contacting the Author

Please direct any questions or feedback to the author at:

FSD Data Services, Inc.
1001 S. Marshall Street
Suite 125, Box 25
Winston-Salem, NC 27101

E-mail: patrickm@fsddatasvc.com

Paper 607

An Assembler Written in SAS®

Ed Heaton, Westat, Rockville, MD

ABSTRACT

Is SAS a programming language, i.e. can it handle problems typically in the domain of a real language like C? To prove the point, when I was asked to write an assembler for a hypothetical machine (Leland Beck's **SIC/XE**) as a class project for a Systems Programming class at Johns Hopkins University, I chose to write the program in SAS.

This paper will present the programming strategy, techniques, and problems encountered in carrying out the task. Structured programming methods will be emphasized. Macro is used. Numerous functions and formats will be encountered along the road to success. In short, a lot of basic SAS is used and can be learned in this presentation.

THE HYPOTHETICAL MACHINE

This assembler was written to produce object code for the hypothetical machine described in Leland L. Beck's book, "*An Introduction to Systems Programming*." That text describes two machines: a basic machine called **SIC** (Simplified Instructional Computer) and an extended-feature machine called **SIC/XE**. This program will assemble code for either machine.

THE SIC

The features of the **SIC** machine include:

1. Memory – 8-bit bytes in 3-byte words for a total of 2^{15} bytes.
2. Registers – 24-bits each consisting of:
   a. **A** (0) – accumulator (used for arithmetic operations);
   b. **X** (1) – index register (used for addressing);
   c. **L** (2) – linkage register (the Jump to Subroutine (**JSUB**) instruction stores the return address in this register);
   d. **PC** (8) – program counter (contains the address of the next instruction to be fetched for execution); and
   e. **SW** (9) – status word (contains a variety of information, including a Condition Code).
3. Data formats:
   a. Character – strings of 8-byte ASCII codes and
   b. Integer – 24-bit, 2's complement.
4. Instruction formats (24-bit) consisting of:
   a. first 8 bits contain the operation code (**OpCode**);
   b. 9th bit is a flag (**X**) to indicate indexed-addressing; and
   c. last 15 bits for the address.
5. Addressing modes (re. the **X** bit):
   a. direct – the address is the target address and
   b. indexed – the contents of the **X** register is added to the address to get the target address.
6. An instruction set consisting of 24 distinct instructions.

THE SIC/XE

The **XE** version of the hypothetical machine includes the following features:

1. Memory increased to 2^{20} bytes.
2. Registers – four additional registers:
   a. **B** (3) – base register (used for addressing);
   b. **S** (4) – general working register (no special use);
   c. **T** (5) – another general working register; and
   d. **F** (6) – floating-point accumulator (48 bits).
3. Data formats – allows 48-bit floating-point numbers:
   a. First bit indicates the sign;
   b. Next 11 bits hold the exponent; and
   c. Last 36 bits hold the mantissa.
4. Instruction formats:
   a. Format 1 (1 byte) is simply the **OpCode**;
   b. Format 2 (2 bytes) consists of
      i. 8 bits for the **OpCode**,
      ii. 4 bits for the address of the first register, and
      iii. 4 bits for the address of the second register;
   c. Format 3 (3 bytes) consists of
      i. 6 bits for the **OpCode**,
      ii. a bit-flag (**n**) requests indirect addressing,
      iii. a bit-flag (**i**) specifies immediate operands,
      iv. a bit-flag (**x**) to indicate indexed addressing,
      v. a bit-flag (**b**) for base-relative addressing,
      vi. a bit-flag (**p**) for program-counter-relative addressing,
      vii. a bit-flag (**e**) which must be 0 for this format, and
      viii. 12 bits for the displacement.
   d. Format 4 (4 bytes) consists of
      i. 6 bits for the **OpCode**,
      ii. the same six flags as in the 3-byte instruction, and
      iii. 20 bits for the address.
5. Addressing modes – the **XE** machine allows two additional addressing modes:
   a. Base-relative – the displacement ($0 \le$ displacement ≤ 4095) from the Format-3 instruction is added to the value in the **B** register to get the address in memory and
   b. Program-counter – the displacement ($-2048 \le$ displacement ≤ 2047) from the Format-3 instruction is added to the value in the **PC** register to get the address in memory.
6. Instruction set – the **XE** machine has an additional 35 operation codes.

ASSEMBLY CODE

The format for the assembly code is as follows.

- 1 A decimal point indicates a comment line.
- 1-6 Statement label.
- 8-14 Mnemonic operation code.
- 16-21 Operand.
- 23-60 Comment.
- 16-60 Byte Literal. Hexadecimal literals are of the form
  X'000030'
  and character literals should be of the form
  C'The prime numbers less than 1024.'
  Comments can follow the Byte Literal.

THE ASSEMBLER

Most assemblers make two passes of the source program. This assembler is no exception. The first pass collects label definitions and assigns addresses. The second pass performs most of the work. The top-level macro of the assembler (macro **MAIN** – called at the end of the code) consists mainly of two macro calls.

```
%Macro MAIN ;
   %PassOne()
   %PassTwo()
%Mend MAIN ;
```

We need to generate error messages; so start a macro to do just that. We will add WHEN statements as we go along.

```
%Macro GenerateErrorMessage ( errorNumber ) ;
   error = &errorNumber ;
   Put
      @01 "ERROR("
      @07 error
      @11 "): Line="
      @19 lineNumber
```

```
      @ ;
      Select ( error ) ;
         …
         Otherwise ;
      End ;
   %Mend GenerateErrorMessage ;
```

PASS 1

Ok, now to work. Start with pass one. Here we will define our symbols. The tasks are:

- Assign addresses to all statements in the program;
- Save the values (addresses) assigned to all labels for use in Pass 2; and
- Perform some processing of *assembler directives*. (This includes processing that affects address assignments, such as determining the length of data areas defined by **BYTE**, **RESW**, etc.).

We need to read the source code and create two tables. One is the *symbol table* and the other is simply an intermediate working file. The following DATA step will create the two tables and an ASCII text file of error messages. The **SymTab** table includes the name and the location counter for each statement label in the source program. The **IntermediateFile** contains each soruce statement with its assigned address, error indicators, etc. The error file gets written to by the **%GenerateErrorMessage** macro.

We need to create an array to hold the statement labels so that we can check to assure that they are unique. There can only be one label per statement, so set the length of the array is equal to the number of statements. Add the task of finding the number of statements to the macro before the DATA step.

```
   %Macro PassOne () ;
      %GetNumberOfProgramStatements()
      Data
         %DefineSymTab()
         %DefineIntermediateFile()
      ;
         File Errors ;
         %DefinePassOneColumns()
         InFile Program
            lRecL=60
            pad
            end=_lastRecord
         ;
         Input @ ;
         lineNumber = _n_ ;
         code = _inFile_ ;
         If ( subStr( code , 1 , 1 ) eq "." )
            then go to FINISH ;
            Else do ;
               %ProcessAStatement()
            End ;
   FINISH: Output library.IntermediateFile ;
      Run ;
   %Mend PassOne ;
```

The input to the DATA step is the assembly program. The record length is 60 characters; I like to use LRECL and PAD to ward off input errors. The **END** statement is not required in the assembly language, so we used END= to mark the last record.

Notice that we did not really input anything. We just read the record into the input buffer. The contents of the input buffer is simply a character string; we can process this character string like we could any character variable.

Lines beginning with "." contain comments only.

That's the end of the DATA step and the first pass.

PASS 2

In the second pass we will assemble the instructions and generate the object program. The tasks are:

- Assemble instructions (translating operation codes and looking up addresses).
- Generate data values defined by **BYTE**, **WORD**, etc.
- Perform additional processing of assembler directives.
- Write the object program and the assembly listing.

```
   %Macro PassTwo () ;
      %GetLabelAddresses()
      %SetBaseAddresses()
      %If &XEFlag %then %do ;
         %CalculateDisplacement()
      %End ;
      %CreateInstructionCodes()
      %CreateObjectCodes()
      %WriteProgramListing()
      %WriteObjectCodeFile()
   %Mend PassTwo ;
```

That's it. Done.

DETAILS

Oh, yeah. We need to define these macros that we used.

SUPPORTING MACROS

GET NUMBER OF PROGRAM STATEMENTS

We chose to store the labels in an array and in the **SymTab** table. The array is used to assure that the label is unique. To set the upper bound for the array, scan the assembler code, reading only the first byte of each record into the buffer. Then save the number of records in a macro variable. Here is a macro to get this upper bound. Comments start with a decimal point (**.**); let's not include them in the count.

```
   %Macro GetNumberOfProgramStatements () ;
      %Global statements ;
      Data _null_ ;
         InFile program lRecL=1 end=eof ;
         Input ;
         If ( _inFile_ ne "." ) then
            statements ++1
         ;
         If eof then call symPut(
            "statements"
          , left( put( statements , 8. ) )
         ) ;
      Run ;
   %Mend GetNumberOfProgramStatements ;
```

DEFINE SYMTAB

The symbol table includes the name and value (address) for each label in the source program.

```
   %Macro DefineSymTab () ;
      library.symTab (
         label= "Symbol Table"
         index= ( label )
         keep= label locCtr
         where= ( label not in ( " " , "." ) )
      )
   %Mend DefineSymTab ;
```

DEFINE INTERMEDIATE FILE

Pass 1 usually writes an intermediate file that contains each source statement together with its assigned address, error indicators, etc.

```
   %Macro DefineIntermediateFile () ;
      library.IntermediateFile (
         label= "Intermediate File"
         keep=
            lineNumber loc code locCtr label
            mnemonic operand literal byteString
            nFlag iFlag xFlag bFlag pFlag eFlag
            eFlagShift format2Flag error
      )
   %Mend DefineIntermediateFile ;
```

DEFINE PASS-ONE COLUMNS

Let's use an **ATTRIB** statement to define the columns of both tables output by the **%PassOne** macro.

```
   %Macro DefinePassOneColumns () ;
      Attrib
         lineNumber
            label="Line Number"
            format=z4.
         loc
```

```
            label="Address of Instruction"
            format=hex6.
         Code
            label="Code"
            length=$60
            format=$60.
         locCtr
            label="Location Counter"
            format=hex6.
         label
            label="Label"
            length=$6
            format=$6.
         mnemonic
            label="Mnemonic Operation Code"
            length=$6
            format=$6.
         operand
            label="Operand"
            length=$8
            format=$8.
         literal
            label="Literal Value"
            format=hex6.
         byteString
            label="Literal Value Byte String"
            length=$84
         nFlag
            label="Indirect Flag"
         iFlag
            label="Immediate Flag"
         xFlag
            label="Index Flag"
         bFlag
            label="Base-Relative Flag"
         pFlag
            label="Program-Counter Flag"
         eFlag
            label="Extended-Addressing Flag"
         eFlagShift
            label="Extended-Address Bit-Shifter"
            format=hex4.
         format2Flag
            label="Format 2 Flag"
         error
            label="Error Code"
            format=z4.
      ;
      Length _label1-_label&statements $6 ;
      Array symTab [*] $
         _label1-_label&statements
      ;
      Retain _firstRecord 1 ;
      Retain locCtr _label1-_label&statements ;
   %Mend DefinePassOneColumns ;
```

We retained **_firstRecord**; it will keep us informed on how to process the statements. We also retained the location counter (**locCtr**) and buckets for the statement labels (**_label1-_label&statements**).

PROCESS A STATEMENT

Read the characters in columns 1-6 of the assembler code; this is reserved for labels. Then compare the length of the label with all leading and trailing spaces removed with the length of the label with all spaces removed to determine if there are imbedded spaces.

```
   %Macro ProcessAStatement () ;
      Input label $ 1-6 @ ;
      label = upCase( label ) ;
      If (
         length( label ) ne
         length( compress( label ) )
      ) then do ;
         %GenerateErrorMessage( 1 )
      End ;
      %CheckForEFlag()
      %ReadMnemonic()
      Select ;
         When ( _firstRecord ) do ;
            %ProcessFirstStatement()
         End ;
         When ( mnemonic eq "END" ) do ;
            %ProcessEndStatement()
         End ;
         Otherwise do ;
            %ProcessMiddleStatement()
         End ;
      End ;
   %Mend ProcessAStatement ;
```

We checked to see if the **E** flag needed to be set because extended format instructions are four bytes long. We also read the value in the field for mnemonic operation codes; if it contains the **END** directive, we need to proceed differently.

The **%GenerateErrorMessage** macro will write error messages; let's put this message in the macro before we forget.

```
When ( 1 ) put @24
   "Label contains imbedded spaces."
;
```

GET LABEL ADDRESSES

Add the location counter values from the *Symbol Table* to the records that have a symbol in the *Operand Field*. First merge the *location counter* values into the table created by **%PassOne**.

Then we need to generate errors for those assembler statements where the *operation code* requires a valid label (**SIC** machine only).

We will use the **??** modifier to supress SAS error handling as we attempt to read the *operand* into a numeric field. If the conversion works, we will write that number to the *target address*.

We need a list of the *mnemonic operation codes* to be sure that the statement is valid. We will use a macro variable to hold this list. This macro variable will look like the following.

```
"ADD" "ADDF" "ADDR" "AND" ... "TIXR" "WD"
```

The *OpCodes* are stored in a SAS data set called **OpTab**. Let's look at the code to create this macro variable.

```
%Macro GetOpCodes () ;
   %Global opCodes ;
   Proc sql noPrint ;
      Select
         quote( trim(mnemonic) )
            into :opCodes separated by " "
         from library.OpTab
      ;
   Quit ;
%Mend GetOpCodes ;
```

We will call this macro from the **%PassOne** macro.

```
%Macro PassOne () ;
   %GetNumberOfProgramStatements()
   %GetOpCodes()
   Data
      …
   Run ;
%Mend PassOne ;
```

Now, we can code our **%GetLabelAddresses** macro.

```
%Macro GetLabelAddresses () ;
   Proc sql ;
      Create table PassTwo as
         select
            IntermediateFile.*
          , symTab.locCtr as targetAddress
         from library.IntermediateFile
            left join library.symTab
         on (
            IntermediateFile.operand
          = symTab.label
         )
         order by IntermediateFile.lineNumber
      ;
   Quit ;
   Data PassTwo ;
      File Errors mod ;
      Set PassTwo ;
      If (
         not &XEFlag
       & ( mnemonic in ( &opCodes ) )
```

```
         & ( mnemonic not in (
              &standAloneOpCodes
           ) )
         & missing( targetAddress )
        ) then do ;
           %GenerateErrorMessage( 2 )
        End ;
        If (
           missing( targetAddress )
         & not missing( input(operand,?? 9.) )
        ) then
           targetAddress = input(operand,?? 9.)
        ;
     Run ;
  %Mend GetLabelAddresses ;
```

Add the error message to the **%GenerateErrorMessage** macro.

```
  When ( 2 ) put @24
     "Your operand is not a defined label."
  ;
```

SET BASE ADDRESSES

Search for the **BASE** assembler directive. If found, store the address for computing *base-relative* addressing.

```
  %Macro SetBaseAddresses () ;
     Data PassTwo ;
        Set PassTwo ;
        Attrib
           baseAddr
              label= "Base Address"
              length= 4
              format= hex6.
        ;
        Retain baseAddr ;
        If ( mnemonic eq "BASE" ) then do ;
           baseAddr = targetAddress ;
           targetAddress = . ;
        End ;
     Run ;
  %Mend SetBaseAddresses ;
```

CALCULATE DISPLACEMENT

If we are assembling code for an **XE** machine, then we need to calculate the displacement for *relative* addressing.

If the *operand* field contains a number, we do not want to calculate a displacement. To test for the number, attempt to convert the value to a number with the INPUT function. Use the ?? format modifier to suppress error messages. If the field does not contain a number, the the INPUT function will return a missing value which can be tested with the MISSING function.

We will first test to see if we can use *program-counter-relative* addressing. If we cannot, we will test to see if we can use *base-relative* addressing. If we can't use either, then the **E** flag had better be set.

```
  %Macro CalculateDisplacement () ;
     Data PassTwo ;
        Set PassTwo ;
        File Errors mod ;
        If missing(
           input( operand , ?? 9. )
        ) then select ;
           When (
              -2048 le
              ( targetAddress - locCtr ) le
              2047
           ) do ;
              pFlag = 1 ;
              targetAddress =
                 targetAddress - locCtr
              ;
              If ( targetAddress lt 0 ) then
                 targetAddress =
                    input( subStr( put(
                       targetAddress , hex6.
                    ) , 4 )
                  , hex6.
                 )
              ;
           End ;
           When ( 0 le (
              targetAddress - baseAddr
           ) le 4095 ) do ;
              bFlag = 1 ;
              targetAddress =
                 targetAddress - baseAddr
               ;
           End ;
           Otherwise if (
              missing( eFlag )
            & ( upCase( mnemonic ) ne "END" )
            & not missing( targetAddress )
           ) then do ;
              %GenerateErrorMessage( 3 )
           End ;
        End ;
     Run ;
  %Mend CalculateDisplacement ;
```

Add the error message to the **%GenerateErrorMessage** macro.

```
  When ( 3 ) put @24
     "You must use extended format for direct"
     " addressing."
  ;
```

CREATE INSTRUCTION CODES

This macro will combine the *OpCode*, the address, and, if necessary, an index flag to create an *Instruction Code*.

- The **n** flag bit is used to indicate *indirect-addressing* mode. It is the 18th bit from the right. E.g.:
  0000 0010 0000 0000 0000 0000 -> X'020000'
- The **i** flag bit is used to indicate *immediate-addressing* mode. It is the 17th bit from the right. E.g.:
  0000 0001 0000 0000 0000 0000 -> X'010000'
- The **x** flag bit is used to indicate *indexed-addressing* mode. It is the 16th bit from the right. E.g.:
  0000 0000 1000 0000 0000 0000 -> X'008000'
- The **b** flag bit is used to indicate *base-relative-addressing* mode. It is the 15th bit from the right. E.g.:
  0000 0000 0100 0000 0000 0000 -> X'004000'
- The **p** flag bit is used to indicate *program-counter-addressing* mode. It is the 14th bit from the right. E.g.:
  0000 0000 0010 0000 0000 0000 -> X'002000'
- The **e** flag bit is used to indicate *extended-addressing* mode. It is the 21st bit from the right of this 4-byte instruction. E.g.:
  0000 0000 0001 0000 0000 0000 0000 0000 -> X'00100000'

```
  %Macro CreateInstructionCodes ;
     Proc sql ;
        Create table library.PassTwo as
           select
              PassTwo.*
            , OpTab.opCode
            , sum(
                 OpTab.opCode
                  * input( "010000" , hex6. )
                  * eFlagShift
               , PassTwo.nFlag
                  * input( "020000" , hex6. )
                  * eFlagShift
                  * &XEFlag
               , PassTwo.iFlag
                  * input( "010000" , hex6. )
                  * eFlagShift
                  * &XEFlag
               , PassTwo.xFlag
                  * input( "008000" , hex6. )
                  * eFlagShift
               , PassTwo.bFlag
                  * input( "004000" , hex6. )
                  * eFlagShift
                  * &XEFlag
               , PassTwo.pFlag
                  * input( "002000" , hex6. )
                  * eFlagShift
                  * &XEFlag
```

```
                , PassTwo.eFlag
                   * input( "00100000" , hex8. )
                   * &XEFlag
                , PassTwo.targetAddress
               ) as instructionCode format=hex6.
            from PassTwo left join library.OpTab
            on (
              PassTwo.mnemonic eq OpTab.mnemonic
            )
            order by PassTwo.lineNumber
         ;
      Quit ;
   %Mend CreateInstructionCodes ;
```

We have not created the variable **eFlagShift**. When we do, it will be used to left-shift the bits of the *Instruction Code* so that we have a *Format-4* instruction. **EFlagShift** will have values of **X'0001'** or **X'0100'**. If the instruction code is multiplied by the latter, it becomes a 4-byte instruction. Multiplying by the former does nothing.

The flags **nFlag**, **iFlag**, **xFlag**, **bFlag**, **pFlag**, and **eFlag** have values of zero or one. If zero, then zero is added to the sum.

CREATE OBJECT CODES

Let's create character-based, user-readable representations of the object code for our instructions.

```
   %Macro CreateObjectCodes () ;
      Data library.PassTwo ;
         Length objectCode $100 ;
         Set library.PassTwo ;
         If ( mnemonic not in (
            "START","BASE","EQU","ORG","END"
         ) ) then do ;
            If ( eFlag )
               then objectCode = put(
                  instructionCode , hex8.
               ) ;
               Else objectCode = put(
                  instructionCode , hex6.
               ) ;
         End ;
         If format2Flag then objectCode =
            subStr( objectCode , 1 , 4 )
         ;
         If not missing( literal ) then
            objectCode = put( literal , hex6. )
         ;
         If not missing( byteString ) then
            objectCode = byteString
         ;
      Run ;
   %Mend CreateObjectCodes ;
```

WRITE PROGRAM LISTING

Write a file that contains the program listing and the errors. Notice that I used the MOD option so that I could append the error file to the assembler listing. For that listing I simply read a record using the INPUT statement and wrote that record using the _INFILE_ automatic variable.

```
   %WriteProgramListing () ;
      Title2 "Assembler Listing" ;
         Proc printTo print=listing new ;
            Proc print data=library.PassTwo ;
               Id lineNumber ;
               Var loc code objectCode ;
            Run ;
         Proc printTo print=print ;
         Data _null_ ;
            InFile Errors ;
            File listing mod ;
            Input ;
            If ( _n_ eq 1 ) then put //
               "***** Error Messages *****"
            / ;
            Put _inFile_ ;
         Run ;
      Title2 ;
    %Mend WriteProgramListing ;
```

WRITE OBJECT CODE FILE

Create the *Object Code File*. Well, actually create a file of ASCII records of the hexidecimal representation of the object code using a header record, text records, and an end record. We will use two passes of the **library.PassTwo** data set to accomplish this: pass one will write the *Header* and *Text* records; pass two will write the *Modify* and *End* records.

To make the *Object Code File* easier for humans to read, we will insert a caret (**^**) between each instruction. We will use the COMPRESS function to find the length of the object code with the carets removed. This is the true length of the object code, which is limited to 30 bytes (60 half-bytes).

A character string input by a **BYTE** directive can contain more than 30 bytes, so we need to be able to partition that string across two text records.

First we will cycle through the object code for the program instructions, accumulate lines of object code, and write either a header record or text records to the *Object Code File*.

```
   %Macro WriteObjectCodeFile () ;
      Data _null_ ;
         Format
            startingAddress         hex6.
            lengthOfObjectCodeField hex2.
         ;
         Length objectCodeField $90 ;
         Retain startingAddress objectCodeField;
         Retain splitFlag 0 ;
         Set library.PassTwo ( where=(
            subStr( code , 1 , 1 ) ne "."
         ) ) ;
         File objCode ;
         Select ( mnemonic ) ;
            When ( "START" ) do ;
               %WriteHeaderRecord()
            End ;
            When ( "END" ) do ;
               %WriteTextRecord()
            End ;
            Otherwise do ;
               If (
                  mnemonic in ("RESW","RESB")
               ) then do ;
                  %WriteTextRecord()
                  %ProcessTextRecord()
               End ;
               If missing( objectCodeField )
                  then startingAddress = loc
               ;
               If ( (
                  length( compress(
                     objectCodeField , "^"
                  ) ) + length( objectCode )
               ) le 60 )
                  then do ;
                     %AccumulateObjectCode()
                  End ;
                  Else do ;
                     If ( length( compress(
                        objectCodeField , "^"
                     ) ) gt 60 )
                        then do ;
                          %SplitLongByteString()
                          %ProcessTextRecord()
                        End ;
                        Else do ;
                          %WriteTextRecord()
                          %ProcessTextRecord()
                        End ;
                  End ;
               End ;
         End ;
      Run ;
```

Next, cycle through the object code again, searching for *extended-format* flags. When found, write **Modify** records. Write an **End** record when done.

```
      Data _null_ ;
         Length objectCodeField $90 ;
```

```
      Format startingAddress hex6. ;
      Set library.PassTwo ( where=
         ( subStr( code , 1 , 1 ) ne "." )
      ) ;
      File objCode mod ;
      Select ;
         When (
            ( eFlag eq 1 )
          & missing( input(operand, ?? 9.) )
         ) do ;
            %WriteModifyRecord()
         End ;
         When ( mnemonic eq "END" ) do ;
            %WriteEndRecord()
         End ;
         Otherwise ;
      End ;
   Run ;
%Mend WriteObjectCodeFile ;
```

CHECK FOR E FLAG

If the 4-byte extended format (Format 4) is used, the format must be specified with the prefix **+** added to the operation code in the source statement. It is the programmer's responsibility to specify this form of addressing when it is required.

```
%Macro CheckForEFlag () ;
   Input @7 _eFlagField $char1. @ ;
   Select (_eFlagField) ;
      When (" ") eFlagShift = 1 ;
      When ("+") do ;
        eFlag = 1 ;
        eFlagShift = input("0100",hex4.) ;
      End ;
      Otherwise do ;
        %GenerateErrorMessage( 4 )
      End ;
   End ;
%Mend CheckForEFlag ;
```

Of course we don't want to forget to describe error 2 in our **%GenerateErrorMessage** macro.

```
When ( 4 ) put @24
   "Column 7 must be"
   %If (&XEFlag eq 1) %then " a + or" ;
   " blank."
;
```

READ MNEMONIC

The *mnemonic operation code* is restricted to columns 8 through 14. This macro will read the mnemonic code and test to assure that it contains no blanks.

```
%Macro ReadMnemonic () ;
   Input mnemonic $ 8-14 @ ;
   mnemonic = upCase( mnemonic ) ;
   If (
      length(mnemonic) ne
      length( compress(mnemonic) )
   ) then do ;
      %GenerateErrorMessage( 5 )
   End ;
%Mend ReadMnemonic ;
```

Now let's go back to our **%GenerateErrorMessage** macro and describe error 3.

```
When ( 5 ) put @24
   "The Mnemonic contains imbedded spaces."
;
```

PROCESS FIRST STATEMENT

The processing of the first record is unique enough that we have a macro just to read it.

```
%Macro ProcessFirstStatement () ;
   If (mnemonic eq "START")
      then do ;
         %ProcessNOriFlag()
         %ReadOperand()
         locCtr = input( operand , hex6. ) ;
         Call symPut(
            "StrtAddr"
          , trim( left( put(locCtr,hex6.) ) )
         ) ;
         End ;
         Else locCtr = 0 ;
      loc = locCtr ;
      _firstRecord = 0 ;
%Mend ProcessFirstStatement ;
```

PROCESS END STATEMENT

The processing of the last record is unique enough that we have a macro just to read it.

```
%Macro ProcessEndStatement () ;
   loc = locCtr ;
   Call symPut(
      "EndAddr"
    , trim( left( put(locCtr,hex6.) ) )
   ) ;
   % ReadOperand()
%Mend ProcessEndStatement ;
```

PROCESS MIDDLE STATEMENT

First we want to store the address of the instruction. If we have a value in the space reserved for statement labels we need to make sure it is a valid label. Finally, we need to determine if the *OpCode* is a *Format-2* instruction and proceed accordingly.

```
%Macro ProcessMiddleStatement () ;
   loc = locCtr ;
   If ( upCase( label ) ne " " ) then do ;
      %VerifyLabel
   End ;
   If ( mnemonic in ( &opCodes ) )
      then do ;
         %ProcessOpCodeInstruction()
      End ;
      Else do ;
         %ProcessAssemblerDirective()
      End ;
%Mend ProcessMiddleStatement ;
```

WRITE HEADER RECORD

This macro will write a header record. The format of the header record is as follows; the numbers are the columns.

- 1 **H**
- 2 field separator (**^**)
- 3-8 program name
- 9 field separator (**^**)
- 10-15 starting address of object program (hexadecimal)
- 16 field separator (**^**)
- 17-22 length of object probram in bytes (hexadecimal)

```
%Macro WriteHeaderRecord () ;
   _ProgramLength = put(
      input( "&EndAddr" , hex6. ) -
      input( "&StrtAddr" , hex6. )
   , hex6.
   ) ;
   Put
      @01 "H^"
      @03 label
      @09 "^&StrtAddr"
      @16 "^"
      @17 _ProgramLength
   ;
%Mend WriteHeaderRecord ;
```

WRITE TEXT RECORD

This macro will write a text record. The format of the text record is as follows; the numbers are the columns.

- 1 **T**
- 2 field separator (**^**)
- 3-8 starting address for object code in this record (hexadecimal)
- 9 field separator (**^**)
- 10-11 length of object code in this record in bytes (hexadecimal)
- 12 field separator (**^**)
- 13-?? object code, represented in hexadecimal (2 columns

per byte of object code -- This section of the record will have field separators between each section of object code. The object code is limited to 60 half-bytes, but the field separators will lengthen this.)

```
%Macro WriteTextRecord () ;
   lengthOfObjectCodeField = length(
      compress( objectCodeField , "^" )
   ) / 2 ;
   If (
      lengthOfObjectCodeField gt 1
   ) then put
      @01 "T^"
      @03 startingAddress
      @09 "^"
      @10 lengthOfObjectCodeField
      @12 objectCodeField
   ;
%Mend WriteTextRecord ;
```

PROCESS TEXT RECORD

Start a text record by initializing the object code field to a caret followed by the first object code. If the split flag is set, then there is already some object code, so simply append to it.

```
%Macro ProcessTextRecord () ;
   If splitFlag
      then do ;
         objectCodeField =
            trim( objectCodeField )
            || "^" || objectCode
         ;
         splitFlag = 0 ;
      End ;
      Else do ;
        If not missing( objectCode )
           then do ;
              objectCodeField =
                 "^" || objectCode
              ;
              startingAddress = loc ;
           End ;
           Else objectCodeField = " " ;
      End ;
%Mend ProcessTextRecord ;
```

ACCUMULATE OBJECT CODE

Add the new object code to the text record, and increment the **lengthOfObjectCode** variable.

```
%Macro AccumulateObjectCode () ;
   If ( not missing( objectCode ) ) then do ;
      objectCodeField =
         trim(objectCodeField)
         || "^" || objectCode
      ;
      lengthOfObjectCodeField =
         lengthOfObjectCodeField
       + length( objectCode ) / 2
      ;
   End ;
%Mend AccumulateObjectCode ;
```

SPLIT LONG BYTE STRING

First set the split-code flag. Then write the extra object code to the **RemainingObjectCode** variable. Now write the text record with the first 60 half-bytes of object code. Finally, initialize the object code field of the new text record with the bytes of object code that we chopped off from the too-long object code.

```
%Macro SplitLongByteString () ;
   splitFlag = 1 ;
   remainingObjectCode = subStr( compress(
      objectCodeField , "^"
   ) , 61 ) ;
   objectCodeField = subStr(
      objectCodeField , 1 , 61
   ) ;
   %WriteTextRecord
   startingAddress = startingAddress + 30 ;
   objectCodeField =
      "^" || remainingObjectCode
   ;
%Mend SplitLongByteString ;
```

%WRITEMODIFYRECORD

This macro will write a **MODIFY** record. The format of the modify record is as follows.

- 1 **M**
- 2 field separator (**^**)
- 3-8 starting location of the address field to be modified, relative to the beginning of the program (hexadecimal)
- 9 field separator (**^**)
- 10-11 length of the address field to be modified, in half-bytes (hexadecimal)

```
%Macro WriteModifyRecord () ;
   startingAddress = loc + 1 ;
   Put
      @01 "M^"
      @03 startingAddress
      @09 "^"
      @10 "05"
   ;
%Mend WriteModifyRecord ;
```

%WRITEENDRECORD

This macro will write an **END** record.

- 1 **E**
- 2 field separator (**^**)
- 3-8 address of first executable instruction in object program (hexadecimal)

```
%Macro WriteEndRecord () ;
   Put @01 "E^" @03 instructionCode ;
%Mend WriteEndRecord ;
```

PROCESS N OR I FLAG

In our assembler language, *indirect* addressing is indicated by adding the prefix @ to the operand. The **n** bit is set to indicate that the contents stored at this location represent the address of the operand, not the operand itself. A # prefix indicates *immediate* addressing where the target address (not the contents stored at that address) becomes the operand; so we will set the **i** bit.

```
%Macro ProcessNOrIFlag () ;
   Input @15 _NIFlagField $char1. @ ;
   If ( (nFlag + iFlag) ne 0 )
      then select ( _NIFlagField )
   ;
      When ( " " ) do ;
         nFlag = 1 ;
         iFlag = 1 ;
      End ;
      When ( "@" ) do ;
         nFlag = 1 ;
         iFlag = 0 ;
      End ;
      When ( "#" ) do ;
         nFlag = 0 ;
         iFlag = 1 ;
      End ;
      Otherwise do ;
         %GenerateErrorMessage( 6 )
      End ;
   End ;
%Mend ProcessNOrIFlag ;
```

Finnally describe error 6 in the **%GenerateErrorMessage** macro.

```
When ( 6 ) put @24
   "Invalid character in OpCode previx."
 / @30 "Specify @ for Indirect addressing"
 / @30 "or # for Immediate addressing "
;
```

READ OPERAND

This macro will read the operand and set the **x** flag. Things can get a bit tricky here; the operand can be:

- a statement label followed by a comma and an **X** (for *indexed* addressing),
- an integer if the *mnemonic* field contains the **RESB**, **RESW**, or **WORD** directives, or
- a character or hex string if the *mnemonic* field contains the **BYTE** directive.

Not all of these are limited to eight characters, so let's just read the entire remainder of the record and then throw away all that follows the first blank space.

```
%Macro ReadOperand () ;
   Input _opPlus $ 16-60 @ ;
   If not missing( _opPlus ) then
      _opPlus = subStr(
         _opPlus
       , 1
       , index( _opPlus , " " ) - 1
      )
   ;
   If (
      length( _opPlus ) gt 8
   ) then do ;
      %GenerateErrorMessage( 7 )
   End ;
   operand = upCase( _opPlus ) ;
   If index( operand , ",X" )
      then do ;
         operand = subStr(
            operand
          , 1
          , index( operand , "," ) - 1
         ) ;
         xFlag = 1 ;
      End ;
      Else xFlag = 0 ;
   If (
      subStr( left(operand) , 1 , 1 ) eq "="
   ) then do ;
      %GenerateErrorMessage( 8 )
   End ;
   If (
      indexC(
         operand
       , "'"
       , '"'
       , "!@#$%^&*()_=|\:;<>?./~`"
      )
    | ( indexc(operand,"+-") gt 1 )
   ) then do ;
      %GenerateErrorMessage( 9 )
   End ;
   If (
      length( operand ) ne
      length( compress( operand ) )
   ) then do ;
      %GenerateErrorMessage( 10 )
   End ;
%Mend ReadOperand ;
```

So let's add the error descriptions to the **%GenerateErrorMessage** macro.

```
When ( 7 ) put @24
   "Comments must be blank-separated from"
   " the operand."
;
When ( 8 ) put @24
   "This assembler does not support"
   " literals."
;
 When ( 9 ) put @24
   "You have an illegal character in your"
   " operand."
;
When ( 10 ) put @24
   "The operand contains imbedded spaces."
;
```

VERIFY LABEL

The label field should contain only labels. So use the array of previously stored labels to determine that we have no duplication. If the label already exists in **symTab**, there is an error. If we have searched all the previously stored labels and have not found the current label, then we will add the current label to the stored labels and cease looking.

```
%Macro VerifyLabel () ;
   Do _i=1 to dim( symTab ) ;
      If ( label eq symTab[_i] )
         then do ;
            %GenerateErrorMessage( 11 )
         End ;
         Else if missing( symTab[_i] )
         then do;
            symTab[_i] = label ;
            Go to WRITETOSYMTAB ;
         End ;
   End ;
WRITETOSYMTAB: Output library.symTab ;
%Mend VerifyLabel ;
```

Now let's write the error message.

```
When ( 11 ) put @24
   "The label has already been used."
;
```

PROCESS OPCODE INSTRUCTION

Some of the operation codes are 2-byte instructions. We need to know which ones. So let's go back to the beginning of our assembler create a list of the *mnemonics* for these *OpCodes*.

```
%Let Format2OpCodes =
   "CLEAR"
 , "COMPR"
 , "DIVR"
 , "MULR"
 , "RMO"
 , "SHIFTL"
 , "SHIFTR"
 , "SUBR"
 , "SVCR"
 , "TIXR"
;
```

Compare the *mnemonic* of the *OpCode* against this list and, if this is a two-byte instruction, set the appropriate flags and increment the *location counter*.

```
%Macro ProcessOpCodeInstruction () ;
   If ( mnemonic in ( &Format2OpCodes ) )
      then do ;
         %ProcessFormat2Instruction()
      End ;
      Else do ;
         format2Flag = 0 ;
         locCtr = sum( locCtr , 3 , eFlag ) ;
         %ProcessNOrIFlag()
         %ReadOperand()
         If (
            ( mnemonic not in (
               &standAloneOpCodes
            ) )
          & missing( operand )
         ) then do ;
            %GenerateErrorMessage( 12 )
         End ;
         If (
            not &XEFlag
          & not missing(
               input( operand , ?? 9. )
            )
         ) then do ;
            %GenerateErrorMessage( 13 )
         End ;
      End ;
%Mend ProcessOpCodeInstruction ;
```

Only the **XE** machine allows numbers in the operand field. So, if we assembled on the basic **SIC** machine, we tested the operand by attempting to write the characters to a number. Use the ?? format modifier to suppress messages to the log.

We used a macro variable called **&standAloneOpCodes** that we need to define. Let's define it at the beginning of our assembler where we defined **&Format2OpCodes**.

```
%Let standAloneOpCodes =
   "FIX"
 , "FLOAT"
 , "HIO"
 , "NORM"
 , "RSUB"
 , "SIO"
 , "TIO"
;
```

Again, we need to add the error descriptions to the **%GenerateErrorMessage** macro.

```
When ( 12 ) put @24 "Operand required." ;
When ( 13 ) put @24
   "Your basic SIC computer cannot use"
   " numbers for operands."
;
```

PROCESS ASSEMBLER DIRECTIVE

Assembler directives tell the assembler how to reserve memory and how to preload that memory. They optionally control addressing modes.

- Let's start with the **WORD** directive. This directive tells the assembler to generate a one-word integer constant.
- Next we have the **RESW** directive. It tells the assembler to reserve the indicated number of words for a data area.
- Now lets process the **RESB** directive to tell the assembler to reserve the indicated number of bytes for the data area.
- Next comes the **BYTE** directive. It is used to generate a character or hexadecimal constant, occupying as many bytes as necessary.
- The **BASE** directive is only for the **SIC/XE** machine which allows *base-relative* addressing. The programmer must tell the assembler what the *base* register will contain during execution of the program so that the assembler can compute displacements.

```
%Macro ProcessAssemblerDirective () ;
   format2Flag = 0 ;
   Select ( mnemonic ) ;
      When ( "WORD" ) do ;
         locCtr ++3 ;
         %ReadOperand()
        If missing(
            input( operand , ?? 9.)
         ) then do ;
            %GenerateErrorMessage( 14 )
         End ;
         Else literal = operand ;
      End ;
      When ( "RESW" ) do;
         %FindLengthOfRESW()
         locCtr = locCtr + _length ;
      End ;
      When ( "RESB" ) do ;
         %FindLengthOfRESB()
         locCtr = locCtr + _length ;
      End ;
      When ( "BYTE" ) do ;
         %ProcessBYTE()
         locCtr = locCtr + _length ;
      End ;
      When ( "BASE" ) do ;
         %ReadOperand()
      End ;
      Otherwise do ;
         %GenerateErrorMessage( 15 )
      End ;
   End ;
```

Again, we need to add the error descriptions to the **%GenerateErrorMessage** macro.

```
When ( 14 ) put @24
   "Undefined mnemonic."
;
When ( 15 ) put @24
   "WORD must be a number."
;
```

PROCESS FORMAT-2 INSTRUCTION

If this is a two-byte instruction, set the appropriate flags and increment the *location counter*. Two-byte instructions allow register addresses as the *operands*; read these addresses.

```
%Macro ProcessFormat2Instruction () ;
   Format2Flag = 1 ;
   NFlag = 0 ;
   IFlag = 0 ;
   LocCtr = LocCtr + 2 ;
   %ReadRegisters()
%Mend ProcessFormat2Instruction ;
```

FIND LENGTH OF RESW

Read the number for the **WORD** directive directly from the assembler statement. Use the ?? input modifier, it will suppress error handling so that anything read that is not the character representation of a number will cause a missing value but no error message will be written to the SAS log.

```
%Macro FindLengthOfRESW () ;
   Input @16 _words ?? @ ;
   If missing( _words )
      then do ;
         %GenerateErrorMessage( 16 )
      End ;
      Else _length = 3 * _words ;
%Mend FindLengthOfRESW ;
```

And add the error message to the **%GenerateErrorMessage** macro.

```
When ( 16 ) put @24
   "Invalid number of reserved words."
;
```

FIND LENGTH OF RESB

Read the number of requested reserved words directly from the assembler statement. Again, use the ?? input modifier.

```
%Macro FindLengthOfRESB () ;
   Input @16 _length ?? @ ;
   If missing( _length ) then do ;
      %GenerateErrorMessage( 17 )
   End ;
%Mend FindLengthOfRESB ;
```

Add the error message to the **%GenerateErrorMessage** macro.

```
When ( 17 ) put @24
   "Invalid number of reserved bytes."
;
```

PROCESS BYTE

To process the **BYTE** directive, we first need to determine if we are reading a character string or a hexadecimal string. Since the operands must start in the 16th column, and since the assembler code is limited to 60 byte statements, we are limited to 43-byte strings.

If the string in the assembler statement is a hexadecimal string, then the length of the memory needed to store the string is half the length of the byte string (It takes two characters to represent a byte.).

If the string in the assembler statement is a character string, then we must convert the character string to a hexadecimal string.

```
%Macro ProcessBYTE () ;
   Length _byteString $42 ;
   Input _hexOrChar $ 16-16 @ ;
   Select ( upCase( _hexOrChar ) ) ;
      When ( "X" ) do ;
         %GetByteString()
         _length = length(_byteString) / 2 ;
         byteString = _byteString ;
      End ;
      When ( "C" ) do ;
         %GetByteString()
         _length = length( _byteString ) ;
         byteString = put(
            trim( _byteString ) , $hex.
         ) ;
      End ;
      Otherwise do ;
```

```
            %GenerateErrorMessage( 18 )
         End ;
      End ;
   %Mend ProcessBYTE ;
```

Add the error message to the **%GenerateErrorMessage** macro.

```
   When ( 18 ) put @24
      "Byte string must be hexadecimal (X) or"
      " character (C)"
   ;
```

READ REGISTERS

This macro will read the registers for the two-byte instructions. It will read the first character of **operand** as the first register and, if the second character of **operand** is a comma, it will then read the third character as the second register.

```
%Macro ReadRegisters () ;
   Input operand $ 16-18 @ ;
   operand = upCase( operand ) ;
   Select ( subStr( operand , 1 , 1 ) ) ;
      %AssignRegisterAddress( r1 )
   End ;
   If (subStr(operand,3,3) eq ",") then do ;
      Select ( subStr(operand,3,3) ) ;
         %AssignRegisterAddress( r2 )
      End ;
   End ;
   operand = left( put( sum(
      ( r1 * input( "1000" , hex4. ) )
    , ( r2 * input( "0100" , hex4. ) )
   ) , 6. ) ) ;
%Mend ReadRegisters ;
```

GET BYTE STRING

Read the remainder of the assembler statement starting with the first character after the **X** or **C** character which indicates the type of byte string. Check to ascertain that the string is delimited by single quote marks. Then strip the quote marks and store the string in **_byteString**.

```
%Macro GetByteString () ;
   Input _byteString $ 17-60 @ ;
   If ( subStr(_byteString,1,1) ne "'" )
      then do ;
         %GenerateErrorMessage( 19 )
      End ;
      Else do ;
         _byteString = subStr(
            _byteString , 2
         ) ;
         If not indexC( _byteString , "'" )
            then do ;
               %GenerateErrorMessage( 19 )
            End ;
            Else _byteString = subStr(
               _byteString
             , 1
             , index( _byteString , "'" ) - 1
            ) ;
         End ;
%Mend GetByteString ;
```

Add the error message to the **%GenerateErrorMessage** macro.

```
   When ( 19 ) put @24
      "Byte string must be enclosed in single"
      " quotes."
   ;
```

ASSIGN REGISTER ADDRESS

This macro will convert the *mnemonic* register name to its address.

```
%Macro AssignRegisterAddress ( register ) ;
   When ( "A" ) &register = 0 ;
   When ( "X" ) &register = 1 ;
   When ( "L" ) &register = 2 ;
   When ( "B" ) &register = 3 ;
   When ( "S" ) &register = 4 ;
   When ( "T" ) &register = 5 ;
   When ( "F" ) &register = 6 ;
   Otherwise do ;
      %GenerateErrorMessage( 20 )
   End ;
%Mend AssignRegisterAddress ;
```

Don't forget the error message.

```
When ( 20 ) put @24
   "Register must be one of"
   " A, X, L, B, S, T, or F."
;
```

CONCLUSION

This assembler operates from a **%MAIN** macro that is called on the last line of the code. Some preliminary initialization code precedes the **%MAIN** macro. This includes definition of global macro variables, creation of a window to get user-supplied information, and creation of references to working libraries and files.

The **%MAIN** macro called the **%PassOne** and **%PassTwo** macros. **%PassOne** generated a list of the mnemonic *operation codes* and stored them in an array. It created another array to store the *statement labels*. **%PassOne** then generated a SAS data set (**IntermediateFile**) containing the *line number*, the location in memory of the code that it will generate for each statement, the *statement label*, the mnemonic *operation code*, the *operand* for the operation codes, *literal values*, and hexadecimal representations of *BYTE* directives. It also produced flags for the *n*, *i*, *x*, *b*, *p*, and *e* bits, and for 2-byte instructions. Finally, **%PassOne** created a symbol table (**SymTab**).

%PassTwo merged the addresses from **SymTab** onto **IntermediateFile** from **%PassOne** using the **%GetLabelAddresses** macro. It assigned base addressees to each record and calculated displacements. It then created hexadecimal instructions and the object code for each assembly statement. Finally, **%PassTwo** wrote the assembler listing, including any error messages, and the object code file.

While this was an exercise for a college class, I hope that you found something to take home with you. We used several data manipulation techniques, and the overall scheme is an example of top-down programming in SAS.

REFERENCES

Beck, Leland L., An Introduction to Systems Programming, 3d ed. (Reading, Massachusetts: Addison Wesley Longman, Inc., 1997).

ACKNOWLEDGMENTS

I want to thank Ian Whitlock of Westat for his continual support and encouragement in my career growth. He was an inspiration before I met him, and has proven to be a wonderful mentor and friend since.

I also want to thank Dianne Rhodes of Westat, who directed my focus toward more career-enhancing facilities such as SAS Users Groups and the SAS-L list-server when we both worked elsewhere.

Finally, I want to thank all the wonderful and insightful contributors to SAS-L for their selfless contributions. They have proven to be my most valuable teaching aid.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Ed Heaton
Westat: An Employee-Owned Research Corporation
1650 Research Boulevard
Rockville, MD 20850
Work Phone: (301) 610-4818
Fax: (301) 294-3992
Email: EdwardHeaton@Westat.com

A Couple of Tasty SAS® Programming Tunes

Paul M. Dorfman

CitiCorp AT&T Universal Card
Jacksonville, Fl

ABSTRACT

Of all wildly different questions asked by daily participants in SAS-L jam sessions, programming problems in the strict sense of the word represent a relatively small part. An even smaller fraction belongs to interesting, difficult, or unconventional problems. While SAS programming puzzles of this nature stubbornly resist brute force, they quite often yield to a good measure of general-programming, not necessarily SAS-specific, thinking, applied before the first line of code has even been typed. When as a result, an efficient and complete SAS solution falls in place as if by magic, it feels not unlike finding a beautiful tune. One who discovers the music is rewarded by a SAS programming melody so tasty that it just must be heard outside of SAS-L. This presentation is an attempt to share a couple of such programming tunes with the SSU audience.

INTRODUCTION

In this paper, I am going to display a small miscellany of serendipitous programming problems asked on SAS-L, for which I was fortunate to offer a solution. All these problems were different in nature but shared one distinctive feature: None could be solved head-on using one of standard SAS programming techniques; or at least a standard solution, if found, would be quite inefficient due to either amount or shape of the data.
In the 'normal' SAS programming practice, the necessity to code *ab ovo*, at a very low algorithmic level, is rarely required or, should such necessity arise, an attempt is usually made to circumvent the labor by

1. Using a less efficient but standard method.
2. Buying a piece of software designed to tackle this sort of problems.
3. Asking a question on SAS-L.

The latter method works particularly well, because many SAS-L responders:

1. Are experienced, and not necessarily SAS-only, programmers.
2. Are willing to take their time to help.
3. Like programming challenges and view a difficult programming task as an interesting mental exercise.
4. Test their solutions and offer their extensions if necessary.
5. Are free.

Besides, some, very rare respondents, such as Ian Whitlock, Peter Crawford, Pete Lund, Lauren Haworth, are known to always get is right and trusted with their solutions as if they had an iron-clad guarantee coming with them.

I have always adhered to the thought that since SAS is a general programming language in its own right -- and would be even if its only component were the DATA step -- there is virtually no general programming problem that could be solved in a different software but not in SAS. By the 'general programming problem', I mean not some fancy GUI development or systems programming, but rather the implementations of schemes described in general texts on algorithms, such as Knuth[1,2], Sedgewick[3], or Standish[4].

You will see that the problems examined below pretty much fall under the cover of this loose definition. They are presented almost 'as is', with the questions as formulated by the original poster and my answers edited only for the sake of making them more suited for a paper than a personal response.

1. A KNAPSACK OF ADJUSTMENTS

This question was asked twice, almost in the same terms, by two different posters a year or so apart. Here I adhere to the thread initiated by Greg Moron gmoron@netscape.net.

Q.:

I have 50 so-called usage charges:

```
 48 30 45 08 19 22 37 46 13 14
 49 36 05 25 44 01 11 03 32 32
 09 05 35 32 03 41 03 20 28 39
```

```
 40 46 47 33 03 03 08 13 50 15
 06 42 20 09 32 24 16 41 21 01
```

I also have a large (over 3 million observations) SAS data set TARGET with 1 variable called TARGET listing 'target adjustments'. For example:

OBS	TARGET
1	53
2	278
3	115
4	75
5	264
6	278
7	156
8	153
9	15
10	20

For each target adjustment value in this file, I have to create a 'real' adjustment according to the following rules. If the target adjustment matches one of the usage charges, the charge is chosen as the real adjustment. E.g., it would be the case for TARGET=15 and 20. If a match is not found, I must try to combine 2 charges so that their sum would equal the target adjustment. For instance, for TARGET=53 the charges 50+3 would produce the required adjustment. If I can't find 2 terms like that I must try 3 terms, and so on, up to 6 terms. Generally, if a target figure can be obtained in a number of ways, a combination with fewest terms and largest possible terms must be selected. If the TARGET value can't be matched by any 1-, 2-, ..., 6-term combination, TARGET+1 value should be considered as a target adjustment, and so on. At end, I have to have a file ADJ written with TARGET and 6 variables ADJ1--ADJ6. For instance, for the first 10 observations from the file TARGET I can find ‘by hand’ that:

TARGET	ADJ1	ADJ2	ADJ3	ADJ4	ADJ5	ADJ6
53	50	3	.	.	.	.
278	50	49	48	47	45	39
115	50	49	16	.	.	.
75	50	25	.	.	.	.
264	50	49	48	47	46	24
278	50	49	48	47	45	39
156	50	49	48	48	.	.
153	50	49	48	48	.	.
15	15	.	.	.	.	.
20	20	.	.	.	.	.

As a matter of fact, I am not even sure I have selected the optimum required combinations for the large targets but at least, the terms total to the target adjustment. I am not so much confused about creating the combinations, rather as to how to search them efficiently in this particular case. Now -- to the response.

A.:

The task reminds the "knapsack" problem usually attacked using greedy algorithms. However, in this particular case, the number of combinations at hand does not prohibit an exact solution. The total number of 1-, 2-, ... 6-term combinations out of 50 is 18,260,635, believe it or count it. Of course, storing all of them in a kind of memory table to facilitate a comparison-type (i.e. sequential, binary, formatted, etc.) search for the needed sum each time a record comes from TARGET would be a painful experience. Moerover, it would be unclear how to make good use of equal usage charges such as 3 and 32. However, once we agree to forget about comparing TARGET to the usage charges and move from comparison-based searching to searching based on direct addressing into an array, the pieces of the puzzle fall in place as if by magic.

Let us first consider what kind of sums we are dealing with.
Rearranging the usage charges into descending order, we have:

```
50 49 48 47 46 46 45 44 42 41
41 40 39 37 36 35 33 32 32 32
32 30 28 25 24 22 21 20 20 19
16 15 14 13 13 11 09 09 08 08
06 05 05 03 03 03 03 03 01 01
```

Hence, the maximum sum the charges can adjust is (50 + 49 + + 48 + 47 + 46 + 46) = 286, the minimum sum obviously being 1. Therefore, instead of creating all the combinations to search, we can only compute all possible sums from 1 to 286 that 1-, 2-,...6-term combinations are able to produce. Then we will key-index an 286-bucket array by the sums corresponding to different combinations. Each sum as a search key will be used as an index into the array itself (hence key-indexing).

To account for all six possible terms, we will need a 2-dimensional array S(1:6,1:286). By creating the needed combinations from an input containing the usage charges in descending order, we can satisfy the precedence requirement naturally by not allowing the buckets already key-indexed (we will know it by the contents of the first row cell in the array S) to be overwritten by a combination having a lower priority.

```
Data Uc;
   Input Uc @@;
Cards;
48 30 45 08 19 22 37 46 13 14
49 36 05 25 44 01 11 03 32 32
```

```
09  5 35 32 03 41 03 20 28 39
40 46 47 33 03 03 08 13 50 15
06 42 20 09 32 24 16 41 21 01
;
Run;

Proc Sort Data=Uc; By Descending Uc; Run;

Data _Null_;
   Do X=1 To N;
      Set Uc Nobs=N;
      If X Eq N Then Min  = Uc;
      If X Le 6 Then Max ++ Uc;
   End;
   Call Symput ('Min',Compress(Put(Min,Best.)));
   Call Symput ('Max',Compress(Put(Max,Best.)));
   Call Symput ('Ucn',Compress(Put(N  ,Best.)));
Run;

Data Adj (Keep=Target Adj1-Adj6);
   Array A   (       1:&Ucn) _Temporary_;
   Array S   (1:6,&Min:&Max) _Temporary_;
   Array Adj (1:6          )            ;

  Do X=1 To &Ucn;
      Set Uc;
      A(X) = Uc;
   End;

   X0 = 0;
   Do X1=X0+1 To &Ucn;
      Sum = A(X1);
      If S(1,Sum) Then Continue;
      S(1,Sum) = A(X1);
   End;

   Do X1=X0+1 To &Ucn-1;
   Do X2=X1+1 To &Ucn-0;
      Sum = A(X1)+A(X2);
      If S(1,Sum) Then Continue;
      S(1,Sum) = A(X1);
      S(2,Sum) = A(X2);
   End; End;

   Do X1=X0+1 To &Ucn-2;
   Do X2=X1+1 To &Ucn-1;
   Do X3=X2+1 To &Ucn-0;
      Sum = A(X1)+A(X2)+A(X3);
      If S(1,Sum) Then Continue;
      S(1,Sum) = A(X1);
      S(2,Sum) = A(X2);
      S(3,Sum) = A(X3);
   End; End; End;

   <. . Fill the gap by induction . . >

   Do X1=X0+1 To &Ucn-5;
   Do X2=X1+1 To &Ucn-4;
   Do X3=X2+1 To &Ucn-3;
   Do X4=X3+1 To &Ucn-2;
   Do X5=X4+1 To &Ucn-1;
   Do X6=X5+1 To &Ucn-0;
      Sum = A(X1)+A(X2)+A(X3)+A(X4)+A(X5)+A(X6);
      If S(1,Sum) Then Continue;
      S(1,Sum) = A(X1);
      S(2,Sum) = A(X2);
      S(3,Sum) = A(X3);
      S(4,Sum) = A(X4);
      S(5,Sum) = A(X5);
      S(6,Sum) = A(X6);
   End; End; End; End; End; End;

   Do Until (Eof);
      Set Target End=Eof;
      If Not (&Min Le Target Le &Max)
        Then Continue;
      Do While (Not S(1,Target));
         Target ++ 1;
      End;
      Do X=1 To 6;
         Adj(X) = S(X,Target);
      End;
      Output;
   End;
Run;
```

The Data _Null_ step was used to find the minimum and maximum ranges pertaining to the usage charges and create macro variables for sizing up arrays in the subsequent step.
In the code above, the combination loops are kept unrolled for the sake of initial clarity. However, now it is now clear by induction how to write a macro capable of assembling this code, the last step can be made much more concise. For example, the needed macro could be coded this way:

```
%Macro Cmb(N);
   %Local Z;
   %Do Z=1 %To &N;
      Do X&Z=X%Eval(&Z-1)+1 To &Ucn-(&N-&Z);
   %End;
   Sum = A(X1)
   %Do Z=2 %To &N; +A(X&Z)             %End;;
   If S(1,Sum) Then Continue;
   %Do Z=1 %To &N; S(&Z,Sum) = A(X&Z); %End;
   %Do Z=1 %To &N; End;                %End;
%Mend Cmb;
```

Now the macro can be used to shorten the program dramatically:

```
Data Adj;
   Array A   (       1:&Ucn) _Temporary_;
   Array S   (1:6,&Min:&Max) _Temporary_;
   Array Adj (1:6          )            ;
   Do X=1 To &Ucn;
      Set Uc;
      A (X) = Uc;
   End;
```

```
   X0 = 0;

   %Cmb (1)  %Cmb (2)  %Cmb (3)
   %Cmb (4)  %Cmb (5)  %Cmb (6)

   Do Until (Eof);
      Set Target End=Eof;
       If Not (&Min Le Target Le &Max)   Then
       Continue;
      Do While (Not S(1,Target));
         Target ++ 1;
      End;
      Do X=1 To 6;
         Adj(X) = S(X,Target);
      End;
      Output;
   End;
Run;
```

Computing the combinatorial sums and preparing the key-indexed table is the longest an most slowly executed part of the program. When the last combination macro has executed, the table is ready for searching. The TARGET file is then read in an explicit DO loop, and each target value coming from its observations is searched via a single array reference.

How long does it take the computer to solve this, at the first glance, CPU and I/O intensive problem? With 1,000,000 observations in TARGET, going over all 18,260,635 combinations, using them to key-index the sums, and computing the adjustments takes about a minute, in all, on a 233MHz P-II machine running 6.12 under Windows NT.

2. KEY-LINKING

This type of problem arises frequently in practical applications, especially in business situations where a customer is identified by a key (credit card number, telephone number) that is likely to change due a number of circumstances. A customer, for instance, could lose a credit card – in which case a new number is issued, while the old number remains in the database as a secondary key. Numerous changes of this kind produce a chain of keys that, in the absence of a unified shadow key, sooner or later needs to be traced. At the end of 1998, a problem of that sort was raised on SAS-L for the first time by Ludwig Boltzmann, and a number of people, notably Ian Whitlock, Karsten Self, and I offered solutions that basically differed in terms of methods different responders had employed to search the columns with ‘new’ and ‘old’ keys.

Here I am presenting the most recent version of this problem formulated and posted by Max Zwingli maxzwingli@mail.nu and my response, in which I took an opportunity to do an exercise in structured SAS programming comparing LINK and macro approaches, and, frankly, to throw in a couple of arguments for hashing as a searching technique.

Q.:

I've got a SAS file with 2 variables: OLD and NEW. They represent old and new phone numbers once the customer's phone has changed. When this happens the old-new obs is simply inserted somewhere in the file, no ordering or indexing of any kind, and the numbers have nothing to do with the time the record was inserted. Here's a snapshot of how the data might look like:

```
  OLD         NEW
8888888888 9999999999
2222222222 3333333333
5555555555 6666666666
7777777777 8888888888
0000011111 0000022222
3333333333 4444444444
9999999999 0000000000
1111111111 2222222222
```

A human eye will easily see that first-old to last-new chains are:

1. 5555555555-6666666666
2. 7777777777-8888888888-9999999999-0000000000
3. 0000011111-0000022222
4. 1111111111-2222222222-3333333333-4444444444

The file having upwards of 800000 observations, I need to devise a programmatic means of doing this. Actually with the sample like above the output should look like this:

```
   OLD           NEW        NLINKS
5555555555    6666666666       2
7777777777    0000000000       4
0000011111    0000022222       2
1111111111    4444444444       4
```

Where NLINKS represents the number of links in each "chain". So far my efforts have resulted in about fifteen steps of sorts and merges taking quite some time to run. Could anyone suggest a more elegant and/or efficient approach with fewer passes through the input?

A.:

The principal algorithm for solving the problem is quite simple:

1. Allocate 3 hash tables:
   - Containing OLD phones
   - Containing NEW phones
   - Containing NEW phones *parallel* to the first table.

2. Read the file record by record and load the hash tables.
3. Read a record from the file again. Search for OLD phone in the NEW hash table. If the search is unsuccessful, the OLD phone is the beginning of a chain. Otherwise go to step 3.
4. Take the corresponding NEW phone and search it in the OLD table. If a match is found, take the corresponding NEW phone from the hash table 3 and search the OLD table again. Otherwise it is the end of the chain, so stop and output the endpoints, then go to step 3.

Now we see that the problem boils down to repeated hash searches of the same kind, so it makes sense to concoct a unified procedure and apply it (with a degree of flexibility) throughout the program. First, let us try to make use of the Macro Language to create an %HSEARCH() routine that can chameleon itself depending upon the table and key it is searching. I assume that the input data set is called A.

```
%Let H = 1000003; * Prime Number => Nobs*2;

%Macro Hsearch (Table=, Key=);
   %If       %Upcase(&Table) = %Upcase(Old)
             %Then %Let Table = 1;
   %Else %If %Upcase(&Table) = %Upcase(New)
            %Then %Let Table = 2;
   Found = 0;
   Do J=1+Mod(&Key,&H) By 1
      Until (H(&Table,J)=. Or Found);
      If J = &H Then J = 1;
      If H(&Table,J) = &Key Then Found = 1;
   End;
%Mend Hsearch;

*H-dimensions: 1=Old, 2=New, 3=(Old||New) ;

Data Oldnew (Keep=Old New Nlinks);
   Array H (3,&H) _Temporary_;
   Do Until (E1);
      Set A End=E1;
     %Hsearch (Table=Old, Key=Old);
      H(1,J) = Old;
      H(3,J) = New;
     %Hsearch (Table=New, Key=New);
      H(2,J) = New;
   End;
   Do Until (E2);
      Set A End=E2;
     %Hsearch (Table=New, Key=Old);
      If Found Then Continue;
      Do Nlinks=2 By 1 Until (Not Found);
         %Hsearch (Table=Old, Key=New);
         If Found Then New = H(3,J);
      End;
      Output;
   End;
Run;
```

A good number of people dislike the Macro Language whenever a more conventional means of structured programming can be employed. I do not necessarily share that viewpoint given certain SAS limitations, but let us give the LINK subroutine a fair shot as well:

```
*H-dimensions: 1=Old, 2=New, 3=(Old||New) ;

Data Oldnew (Keep=Old New Nlinks);
   Array H (3,&H) _Temporary_;
   Do Until (E1);
      Set A End=E1;
      Table = 1; Key = Old; Link Hsearch;
      H(1,J) = Old;
      H(3,J) = New;
      Table = 2; Key = New; Link Hsearch;
      H(2,J) = New;
   End;
   Do Until (E2);
      Set A End=E2;
      Table = 2; Key = Old; Link Hsearch;
      If Found Then Continue;
      Table = 1;
      Do Nlinks=2 By 1 Until (Not Found);
         Key = New; Link Hsearch;
         If Found Then New = H(3,J);
      End;
      Output;
   End;
   Stop;
   Hsearch:
   Found = 0;
   Do J=1+Mod(Key,&H) By 1
      Until (H(Table,J) = . Or Found);
      If J = &H Then J = 1;
      If H(Table,J) = Key Then Found = 1;
   End;
Run;
```

In both cases, the problem is solved in a single step, albeit with two passes through the file. Above, hash table size is assumed about twice the number of the keys to search, in order to obtain the fastest searching speed with the simplest collision resolution policy -- the linear probing.

3. TOPOLOGICAL SORT

This very intriguing problem was posted by Alex Martchenko amartchenko@netscape.net ,who called it 'Job Scheduling'. This problem, too, lends itself to a kind of structured DATA step programming, but here it is more of a necessity, otherwise the algorithm is difficult to follow given the code.

Q.:

I have a bunch of 'jobs', and all I know
about them is which must be execute before which. Suppose the info is recorded in a SAS data set 'pairs'. Every observation in PAIRS tells that some J_BEF must be performed before J_AFT.

```
DATA PAIRS;
INPUT J_GRP J_BEF: $8. J_AFT: $8.;
LINES;
1 III BBB
1 CCC GGG
1 GGG EEE
1 EEE HHH
1 HHH FFF
1 DDD FFF
1 AAA CCC
1 GGG DDD
1 III EEE
1 BBB HHH
3 555 888
3 555 777
3 666 999
3 444 666
3 000 111
3 222 444
3 111 222
3 111 333
3 000 999
3 333 666
3 444 777
3 222 555
;
RUN;
```

I need to identify distinct jobs within each group (1 and 3 above) and
output them to a variable JOB_SEQ so that no job listed in J_BEF for that
group follows J_AFT. I don't care about the exact output sequence as long as this rule is not violated. For example the output done by hand may look like shown below (in the 3-group I on purpose selected the jobs pairs to make the output look sorted like 000, 111, 222, 333...):

Obs	J_GRP	JOB_SEQ
1	1	III
2	1	AAA
3	1	BBB
4	1	CCC
5	1	GGG
6	1	EEE
7	1	DDD
8	1	HHH
9	1	FFF
10	3	000
11	3	111
12	3	222
13	3	333
14	3	444
15	3	555
16	3	666
17	3	777
18	3	888
19	3	999

In the example I used 3 byte character values for simplicity but in actuality J_BEF and J_AFT have full 8 bytes. Also even though the file is pretty big (> 21 m obs) each group is limited to no more than 500 records. Questions: 1) Is there a SAS procedure that can accomplish the task; 2) if not has someone an idea how to program it in base SAS. Frankly, I have none. First I decided it'd be easy to do with a couple of sorts and datasteps but the more I think of it, the more hazy it seems.

A.:

What we have got here is the simplest scheduling problem with a topological sort to be done within each by-group. Assuming that you have no cyclic references, it is fairly easy. At least the basic idea of the algorithm lies on the surface: Let us first find all the jobs having *no* predecessors. Apparently, we can place them in the output right away. Then if we remove them from the input, thus erasing their 'before-after' relationships with the remaining items, some other jobs will emerge with no predecessors. Repeating the process until the input has been exhausted, we will finally have output a linear list of items where no successor is listed before any of its predecessors.

Telling the computer to do the same is, however, a more intricate matter. First, it is convenient (and apparently more efficient) to (1) determine the number of predecessors of each job beforehand. Having obtained such a frequency, we can then safely (2) output the items with zero frequencies, since they have no predecessors. Then (3) for each item like that, we also want to look at each of its successors and decrease its predecessor count by a unity. As a result of this operation, some other items will end up having zero counts. Now we can go back to (3), repeating the process until all counts have been zeroed out. To facilitate it, we need to associate a list of successors with each job item. Since there are only 500 items maximum per group, it can be done using a 2-dimensional 500x500 array instead of (more complex) collection of coalesced link lists. Of course, most of the memory space occupied by the array nodes will be wasted, but since it is only 2 Mb, one can afford to sacrifice it for simplicity.

The scheme described above might sound simple, but it is inefficient to scan the entire (modified) input for zero counts each time the process iterates. This can be avoided by maintaining an output queue. After the *initial* scan for zeroes the queue can be initialized by the items having zero predecessor counts. Then we eject the job sitting in the front of the queue, place it in the output, and decrease the predecessor counts of all its successors. If any of them have gone down to zero, the corresponding item is inserted in the rear of the queue. The new front queue item is ejected again, and the loop repeats until the queue is empty and there are no successors to deal with, at which point the goal is apparently reached.

Let us consider the first group:

```
1 III BBB
1 CCC GGG
1 GGG EEE
1 EEE HHH
1 HHH FFF
1 DDD FFF
1 AAA CCC
1 GGG DDD
1 III EEE
```

and do the algorithm by hand - it will make the way of writing a program strikingly clear. First, let us enumerate all the distinct jobs in the group:

III -> 1, BBB -> 2, CCC -> 3, GGG -> 4, EEE->5, HHH -> 6, FFF -> 7, DDD -> 8, AAA -> 9

Second, let us read the input and populate the following table using the enumerating numbers instead of the actual job names:

	1	2	3	4	5	6	7	8	9	10	11	12	...	500
Pre_Cnt	0	1	1	1	2	1	2	1	0	.	.	.		
Suc_Cnt	2	0	1	2	1	1	0	1	1	.	.	.		
Suc1	2	.	4	5	6	7	.	7	3	.	.	.		
Suc2	5	.	.	8	.	.	.	.	.	.	.	.		
Suc3	.	.	.	.	.	.	.	.	.	.	.	.		
....	.	.	.	.	.	.	.	.	.	.	.	.		
Suc500	.	.	.	.	.	.	.	.	.	.	.	.		

According to the plan above, since the items 1 and 9 have zero predecessor counts, we grab them and insert in the rear of the queue one by one, so the initialized queue (rear to front as left to right) now looks like Q[9,1].

Item 1 is in the front of the queue, so we take it and move to the output: OUT[1]. Item 1 has two successors: 2 and 5. Decreasing the predecessor count if item 2 yields 0, so 2 goes in the rear of the queue, and thus now the queue is Q[2,9]. Decreasing PRE_CNT of item 5 yields 1, so we leave it alone -- for now. Since 9 has moved to the front of the queue, we take it and move to the output: OUT[1,9]. But item 9 has item 3 as its successor, so we should decrement PRE_CNT of item 3 by 1. It yields 0, so 3 goes in the queue: Q[3,2]. At this point, the table has acquired the following form (only non-missing rows and columns shown):

	1	2	3	4	5	6	7	8	9
Pre_Cnt	0	0	0	1	1	1	2	1	0
Suc_Cnt	2	0	1	2	1	1	0	1	1
Suc1	2	.	4	5	6	7	.	7	3
Suc2	5	.	.	8	.	.	.	.	.

Next item in front of the queue is 2, and so it goes to the output: OUT[1,9,2]. Item 2 has no successors, so we turn out attention back to the queue, which at the moment is Q[3]. Item 3 goes to the output: OUT[1,9,2,3], and it has item 4 as its sole successor. Subtracting 1 from PRE_CNT(4) knocks it down to zero, so 4 is inserted in the queue: Q[4]. There are no more successors to process for item 4, so we go back to the queue and eject item 4 into the output: OUT[1,9,2,3,4]. Its successors are 5 and 8. Decreasing PRE_CNT of both makes them ripe for the queue, and that is where they go: Q[8,5]. Front item 5 goes to the output: OUT[1,9,2,3,4,5]. Its successor 6 has PRE_CNT(6)=1, so decreasing it by 1 results in 0, and 6 goes in the queue: Q[6,8]. Item 8 is in front, so it is output: OUT[1,9,2,3,4,5,8]. It then causes PRE_CNT(7) of its only successor 7 to go down by 1, and the queue, now Q[6], spits out its only entry: OUT[1,9,2,3,4,5,8,6]. The only successor of 6, item 7, has thereby its PRE_CNT gone down to 0, and so it goes in the rear of the queue: Q[7]. Item 7 goes to the output list: OUT[1,9,2,3,4,5,8,6,7]. This terminates the process, since there are no more successors to get involved with, and the queue is empty: Q[]. All we have to do now is replace the numbers with their respective job names:

JOB_SEQ [III AAA BBB CCC GGG EEE DDD HHH FFF].

Obviously, the scheme is very easy to program if the jobs are contiguously enumerated, for because the table is indexed by item numbers, any item in question is in effect located immediately by key-indexed search. Otherwise we would have to conduct a search of a different type - and nothing comes close to key-indexing in speed and simplicity. Therefore, it makes sense to spend some time enumerating unique items within each by-group. It can be done on the fly, and the most natural medium to do it is a hash table. We simply allocate a hash table JN_HSH of a prime size about 2*500 and try inserting the next job item. If it is the first time in, the item gets the next

serial number recorded in a parallel table J_To_N, and the inverse relationship is recorded in a table N_To_J (apparently having to house maximum 500 nodes). If the item is duplicate, we simply use the number already sitting in this hash location of J_To_N.

Organizing a queue is even easier. (Note that here, SAS's ready-to-go method of organizing a queue, LAG, cannot go anywhere: We need a queue that would facilitate ejection of the item from the front asynchronous with insertion in the rear - while LAG does both simultaneously.) An array of size 500 will do the job if we use 2 pointers, REAR and FRONT, initially positioned at 0 and 1. To insert, we increment REAR by a unity and stick an item in QUEUE(REAR). To eject, we output QUEUE(FRONT) and increment FRONT, effectively moving the next array node to the front of the queue. FRONT = REAR means the queue having become empty.

Neither the QUEUE, nor PRE_CNT array has to be cleaned up before the next by-group: The ex-contents of the queue matter not when the FRONT/REAR pointers are initialized; and at the end of the process, all predecessor counts are zeroed out. However, both the hash table and the successor table have to be cleaned up properly.

The DATA step program below solves the scheduling problem within given limitations (no more than 500 distinct jobs per group). Writing the code, I have significantly deviated from my own non-structured, parsimonious practice in hope that it could make the intent of the instructions more transparent. You be the judge if I have failed or not. If you prefer a more concise style, you can, for instance, pack arrays Pre_Cnt, Suc_Cnt, and Queue in 0, -1, and -2 rows of Suc_Lst, shorten notation, and so on.

```
Data Pairs;
   Input J_Grp J_Bef: $8. J_Aft: $8.;
Cards;
1 III BBB
1 CCC GGG
1 GGG EEE
1 EEE HHH
1 HHH FFF
1 DDD FFF
1 AAA CCC
1 GGG DDD
1 III EEE
1 BBB HHH
3 555 888
3 555 777
3 666 999
3 444 666
3 000 111
3 222 444
3 111 222
3 111 333
3 000 999
3 333 666
3 444 777
3 222 555
2 LLL MMM
2 LLL YYY
2 TTT XXX
2 VVV TTT
2 SSS VVV
2 UUU SSS
2 UUU WWW
2 XXX QQQ
2 WWW TTT
2 VVV YYY
2 SSS LLL
;
Run;

%Let M =  500; *Max Distinct Jobs Per Group;
%Let H = 1009; *Hash Size For Enumeration  ;

Data Schedule (Keep=J_Grp Job_Seq);
   Array Suc_Lst (1:&M, 1:&M)    _Temporary_;
   Array Pre_Cnt (1:&M      )    _Temporary_;
   Array Suc_Cnt (1:&M      )    _Temporary_;
   Array Queue   (1:&M      )    _Temporary_;
   Array N_To_J  (1:&M      ) $8 _Temporary_;
   Array J_To_N  (1:&H      )    _Temporary_;
   Array JN_Hsh  (1:&H      ) $8 _Temporary_;

   Link Init;

   Do Until (Last.J_Grp);
      Set Pairs;
      By J_Grp Notsorted;

      J = J_Bef; Link Enum; Pre_X = J_To_N (X);
      J = J_Aft; Link Enum; Suc_X = J_To_N (X);

      Pre_Cnt (Suc_X) ++ 1;
      Suc_Cnt (Pre_X) ++ 1;
      Suc_Lst (Pre_X, Suc_Cnt(Pre_X)) = Suc_X;
   End;

   Do Q = 1 To N;
      Link Q_Insert;
   End;

   Do Front = 1 By +1 Until ( Front = Rear );
      Link Q_Eject;
      Do X=1 By +1 While ( Suc_Lst(Q_Front,X) );
         Q = Suc_Lst (Q_Front,X);
         Pre_Cnt (Q) +- 1;
         Link Q_Insert;
      End;
   End;

   Link Clean_Up;
Return;

Init:
```

```
   Do X = 1 To &H;
      JN_Hsh (X) = ' ';
      J_To_N (X) =  0 ;
   End;
   N    = 0;
   Rear = 0;
Return;

Enum:
   Do X = Mod(Input(J,Pib6.),&H) + 1 By +1
                  Until ( JN_Hsh(X) = J );
      If H > &H Then H = 1;
      If Not J_To_N (X) Then Do;
         N ++ 1        ;
         J_To_N (X) = N;
         N_To_J (N) = J;
         JN_Hsh (X) = J;
      End;
   End;
Return;

Q_Insert:
    If Not Pre_Cnt (Q) Then Do;
       Rear ++ 1;
       Queue (Rear) = Q;
    End;
Return;

Q_Eject:
      Q_Front = Queue (Front);
      Job_Seq = N_To_J (Q_Front);
      Output;
Return;

Clean_Up:
   Do X = 1 To N;
      Suc_Cnt (X) = 0;
      Do Q = 1 To N;
         Suc_Lst (X, Q) = 0;
      End;
   End;
Return;
Run;
```

The topological sort program can be modified in order to detect vicious cycles (that is, intransitive relations of the type B->W W->Z Z->B). On a more philosophical note, the problem of topological sort is similar to that of delineating a directed acyclic graph.

CONCLUSION

SAS problems and, by consequence, SAS programs are not created equal. Some, even a majority, eagerly yield to a relatively few ready-to-go SAS coding techniques. Others, probably a minority, are more stubborn and need actual programming to tackle them. On the other hand, their solutions are quite useful to those asking for help and satisfactory to those finding solutions. Playing good music is a great gift. But finding a new, beautiful melody is unlike anything else.

REFERENCES

1. D. E.Knuth, *The Art of Computer Programming*, 2.
2. D. E.Knuth, *The Art of Computer Programming*, 3.
3. R. Sedgewick, Algorithms in C, Parts 1-4.
4. T. A. Standish. Data Structures, Algorithms and Software Principles in C.

ACKNOWLEDMENTS

My hat goes off to Ian Whitlock for providing his friendly support for this unconventional kind of SAS programming on SAS-L. I would also like to thank Ian for the invitation to present these SAS programming tunes in SSU Serendipity Section.

And this paper would have never been written if it had not been for my wife Doris, her infinite patience and non-intrusive care.

AUTHOR CONTACT INFORMATION

Paul M. Dorfman,
SAS Programmer

10023 Belle Rive Blvd. 817,
Jacksonville, FL 32256

(904) 564-1931 (h)
(904) 905-5428 (o)

sashole@bellsouth.net
paul_dorfman@hotmail.com

Problem Solving Techniques with SQL

Kirk Paul Lafler, Software Intelligence Corporation

ABSTRACT

Solving technical problems involves the use of good techniques. Data processing problems frequently involve a great deal of data manipulation and computing resources. When confronted with problems of this nature, you could approach them using conventional DATA, and/or PROC step methods, or you could use the strengths of the SQL procedure to manipulate and process this data. Attendees will learn how SQL can be used to solve traditional, as well as not so traditional, problems including sorting and grouping data, using summary functions, performing two-, three-, multi-table, and outer joins, and constructing subqueries to pass results from one query to another for further processing.

INTRODUCTION

The SQL procedure is a wonderful tool for querying and subsetting data; creating tables; ordering, grouping, and regrouping data; joining two or more tables (up to 32); constructing views (virtual tables); and creating subqueries. Occasionally, a problem comes along where the SQL procedure is either better suited or easier to code than conventional DATA and/or PROC step methods.

Although each problem and proposed solution should be examined on their own, an area where the SQL procedure excels is in the joining of two or more tables. In particular, joining two or more tables using the outer join syntax (left, right, and full). One significant difference between outer joins and conventional inner joins is that outer joins can only join two tables at a time, while inner joins can process up to sixteen tables.

The advantages of using an outer join becomes most apparent when trying to determine not only the rows of data that match the WHERE clause, but rows that don't match. Certainly, these techniques can be accomplished using other methods, but the conventions used in the SQL procedure are especially interesting.

This paper presents how views and outer joins can be used to accomplish ordinary data processing tasks.

VIEW ADVANTAGES - WHY USE THEM

There are many reasons for constructing and using views. A few of the more common are presented here.

Minimize, or perhaps eliminate, the need to know the table or tables underlying structure. Often a great degree of knowledge is required to correctly identify and construct the particular table interactions that are necessary to satisfy a requirement. When this prerequisite knowledge is not present, a view becomes a very attractive alternative. Once a view is constructed, users can simply execute the view. This results in the baseline or underlying tables being processed. Consequently, data integrity and control is maintained since a common set of instructions (view) is used.

Reduce the amount of typing for longer requests. Often, a query will involve many lines of instruction combined with logical and comparison operators. When this occurs, there is any number of places where a typographical error or inadvertent use of a comparison operator may present an incorrect picture of your data. The construction of a view is advantageous in these circumstances, since a simple call to a view virtually eliminates the problems resulting from a lot of typing.

Knowledge of SQL language syntax is non-existent or lacking among users. When users are unfamiliar with the SQL language or the construction techniques of views, they only need to execute the desired view (by specifying its name) using simple calls (or select choices from a menu). This simplifies the process and enables users to perform simple to complex operations with custom-built views.

Provide security to sensitive parts of a table. Security measures can be realized by designing and constructing views designating what pieces of a table's information is available for viewing. Since data should always be protected from unauthorized use, views can provide some level of protection (one should also consider and use security measures at the operating system level).

Change and customization independence. Occasionally, table and/or process changes may be necessary. When this happens, it is advantageous to make it a painless process for users. When properly designed and constructed, a view modifies the underlying data without the slightest hint or impact to users, with the one exception that results and/or output may appear differently. Consequently, views can be made to maintain a greater level of change independence.

TYPES OF VIEWS

Views can be typed or categorized according to their purpose and construction method. Joe Celko, author of SQL for Smarties and a number of other SQL-related books, describes views this way, *"Views can be classified by the type of SELECT statement they use and the purpose they are meant to serve."* To classify views in a SAS System environment, one looks at how the SELECT statement is constructed. The following classifications are useful when describing a view's capabilities.

Single-Table Views
Views constructed from a single table are often used to control or limit what is accessible from that table. These views generally limit what columns, rows, and/ or both are viewed.

Ordered Views
Views constructed with an ORDER BY clause arrange one or more rows of data in some desired way.

Grouped Views
Views constructed with a GROUP BY clause divide a table into sets for conducting data analysis. Grouped views are more often than not used in conjunction with aggregate functions (see aggregate views below).

Distinct Views
Views constructed with the DISTINCT keyword tell the SAS System how to handle duplicate rows in a table.

Aggregate Views
Views constructed using aggregate and statistical functions tell the SAS System what rows in a table you want summary values for.

Joined-Table Views
Views constructed from a join on two or more tables use a connecting column to match or compare values. Consequently, data can be retrieved and manipulated to assist in data analysis.

Nested Views
Views can be constructed from other views, although extreme care should be taken to build views from base tables.

CREATING VIEWS

When creating a view, its name must be unique and follow SAS naming conventions. Also, a view cannot reference itself since it does not already exist.

The following example illustrates the process of creating an SQL view.

```
PROC SQL;
  CREATE VIEW  PERM.COLLGRAD  AS
    SELECT LASTNAME, SSN, DOB
      FROM PERM.EMPLOYEE
        WHERE EDUC > 16
        ORDER BY LASTNAME;
QUIT;
```

In this example the CREATE VIEW statement tells the SAS System that a view is to be created using the instructions and conditions specified in the SELECT statement. The resulting view for all intensive purposes looks and behaves like a real table. Although in this case, something similar to a temporary internal table is created.

OUTER JOIN SYNTAX

Most often, we think of joins as being able to relate rows in one table with rows in another. But occasionally, you may want to include rows from one or both tables that have no related rows. This concept is referred to as row preservation and is a significant feature offered by the outer join construct.

There are operational and syntax differences between inner (natural) and outer joins. First, the maximum number of tables that can be specified in an outer join is two (the maximum number of tables that can be specified in an inner join is 32). Like an inner join, an outer join relates rows in both tables. But this is where the similarities end because the result table also includes rows with no related rows from one or both of the tables. This special handling of “matched” and “unmatched” rows of data is what differentiates an outer join from an inner join.

An outer join can accomplish a variety of tasks that would require a great deal of effort using other methods. This is not to say that a process similar to an outer join could not be programmed – it would just require more work. Let’s take a look at a few tasks that are possible with outer joins:

- List all customer accounts with purchases during the month, including customer accounts with no purchase activity.
- Compute the number of orders placed by each customer, including customers who have not placed an order.
- Identify automobile owners who had their 35,000-mile checkup at the scheduled interval, and those who did not.

Another obvious difference between an outer and inner join is the way the syntax is constructed. Outer joins use keywords such as LEFT JOIN, RIGHT JOIN, and FULL JOIN, and has the WHERE clause replaced with an ON clause. These distinctions help identify outer joins from inner joins.

Finally, specifying a left or right outer join is a matter of choice. Simply put, the only difference between a left and right join is the order of the tables they use to relate rows of data. As such, you can use the two types of outer joins interchangeably and is one based on convenience. The syntax requirements for ***left*** outer joins follows:

Left Outer Join

```
proc sql;
  title1   'Left Outer Join Syntax';
  select  *
     from libref.left
          left join
          libref.right
       on left.key  =  right.key;
quit;
```

Figure 1.

The syntax requirements for ***right*** outer joins follows:

Right Outer Join

```
proc sql;
  title1  'Right Outer Join Syntax';
  select  *
     from libref.left
          right join
          libref.right
       on left.key  =  right.key;
quit;
```

Figure 2.

The syntax requirements for a ***full*** outer join follows:

Full Outer Join

```
proc sql;
  title1  'Full Outer Join Syntax';
  select  *
     from libref.left
          full join
          libref.right
       on left.key  =  right.key;
quit;
```

Figure 3.

USING A LEFT OUTER JOIN

The following data sets are used as input to illustrate left outer join results.

DATASET - AUTOS

Variable	Type	Length
CODE	Char	4
TYPE	Char	15
YEAR	Num	4

Figure 4.

DATASET - INVEST

Variable	Type	Length
CODE	Char	4
GRADE	Char	1
COLOR	Char	8
OWNER	Char	8

Figure 5.

Figure 6 illustrates the result of using a left outer join to identify and match investment autos having an 'A' grade from the AUTOS and INVEST data sets. The resulting output displays all rows for which the SQL expression matched in both tables (is true) and all rows from the left table (AUTOS) that did not match any row in the right (INVEST) table.

Left Outer Join

```
proc sql;
  title1   'Left Outer Join';
  select  *
     from work.autos
          left join
          work.invest
       on autos.code  =  invest.code   and
          invest.grade = 'A';
quit;
```

Figure 6.

Figure 7 illustrates the result of using a right outer join to identify and match investment autos having an 'A' grade from the AUTOS and INVEST data sets. The resulting output displays all rows for which the SQL expression is true and all rows from the right table (INVEST) that do not match any row in the left (AUTOS) table.

Right Outer Join

```
proc sql;
  title1   'Right Outer Join';
  select  *
     from work.autos
          right join
          work.invest
       on autos.code  =  invest.code   and
          invest.grade = 'A';
quit;
```

Figure 7.

Figure 8 illustrates the result of using a full outer join to identify and match investment autos having an 'A' grade from the AUTOS and INVEST data sets. The resulting output displays all rows for which the SQL expression is true and all rows from both tables (AUTOS and INVEST) that do not match any row in the other table.

Full Outer Join

```
proc sql;
  title1   'Full Outer Join';
  select  *
     from work.autos
          full join
          work.invest
        on autos.code  =  invest.code   and
          invest.grade = 'A';
quit;
```

Figure 8.

CONCLUSION

The SQL procedure has an assortment of tools and techniques for solving common data processing problems. Although other methods may exist for resolving certain tasks, at times one method stands out as either being the best or possibly the easiest to code. Performing outer joins in the SQL procedure can certainly be classified as a technique worth further research.

Trademark Citations

ABOUT THE AUTHOR

Kirk is a SAS Quality Partner® and SAS Certified Professional® with 25 years of experience working with the SAS System. He has authored over one hundred articles on computing and technology that have appeared in professional journals, SUGI, regional SAS User Groups, and local SAS User Groups. His popular SAS Tips column appears regularly in the SANDS and SESUG Newsletters. His expertise includes application design and development, training, and programming using base-SAS, SQL, ODS, SAS/FSP, SAS/AF, SCL, FRAME, and SAS/EIS software.

Comments and suggestions can be sent to:

Kirk Paul Lafler
Software Intelligence Corporation
P.O. Box 1390
Spring Valley, California 91979-1390
E-mail: KirkLafler@cs.com
Website: www.software-intelligence.com

Functional Functions

Gary M McQuown, Data and Analytic Solutions Inc., Fairfax, VA
Dorothy E. Brown, Independent Consultant, Matthews, NC

Abstract:

Functions are an important aspect of data step programming that are often overlooked and under utilized. Not only can functions be used to resolve a data step dilemma; they can be mixed and matched to create efficient and precise code. The arrival of V8 includes a number of new functions, making it even more difficult to stay up to date. The following prose and examples cover many of the newly introduced functions as well as some unusual methods of using some old favorites.

Introduction:

SAS functions are among the most basic and commonly used data step tools. A SAS function performs a computation or system manipulation on arguments and returns a value. Most functions use arguments supplied by the user, but a few obtain their arguments from the operating environment. In base SAS software, you can use SAS functions in DATA step programming statements, in a WHERE expression, in macro language statements, in PROC REPORT, and in Structured Query Language (SQL). Some statistical procedures also use SAS functions. With V8, we have over fifty official new functions and some minor enhancements to a few of our old favorites. The following is a collection of documentation and code from various sources (mostly from SAS or SAS-L) intended to explain and promote interest in these functions.

Many of the functions listed are not actually "new". Most have been around on an undocumented experimental basis since late in the V6 series or emulate SCL function behavior. Regardless of their lineage or timing, they are worth learning more about.

Enhancements to PUT SCAN and QUOTE

Two of the most commonly used functions are PUT and SCAN. PUT returns a value using a specified format. It is often used to convert numeric value as a character value. With V8, we have the option of specifying the alignment of the character value returned in addition to its format. This saves us the chore of aligning the value in an additional step. In many ways, that is what functions are all about: making code more efficient, concise and convenient.

- **PUT**(*source*, *format*.)

Returns a value using a specified format

The new alignment specifications for PUT are: -L for left alignment, -C for center alignment and –R for right alignment.

```
Example:
text = "Where does it go";
put text $50. -L ;
put text $50. -C ;
put text $50. -R ;
results =
Where does it go
        Where does it go
                Where does it go
```

- **SCAN**(*argument,n*<, *delimiters*>)

Selects a given word from a character expression

The SCAN function is used to select a given work from a character expression. It has been modified to accept a negative value directing it to read character segments starting from the end of the character string rather than from the beginning.

```
Example:
destination = “New Orleans LA”;
state = scan(destination, -1);
results =  LA .
```

- **QUOTE**(*argument*)

Adds double quotation marks to a character value

The third function to receive modifications is the QUOTE function. The QUOTE function places double quotes around a character value and now retains all trailing blanks. Previous versions of this function removed trailing blanks.

```
Example:
x='George"s';
y=quote(x);
put y;
result = "George's"
```

The following SQL code uses quote to create a list of values.

```
PROC SQL noprint;
select quote( trim( string) ) into :list separated by ', ' from
data_set;
run;
```

New Mathematical and Probability Functions

The introduction of the new mathematical functions makes it easier to compute factorials, permutation, and combinations.

- **COMB**(*n, r*)

Computes the number of combinations of **n** elements taken **r** at a time and returns a value

- **CONSTANT**(*constant*<, *parameter*>)

Computes some machine and mathematical constants and returns a value

CONSTANT allows you to pass certain mathematical values, some of which may be platform or environment specific.

```
Example:
pi = constant ('PI');
```

The following is a list of the constants that can be returned.

Constant 'Argument'

The natural base 'E'
Euler constant 'EULER'
Pi 'PI'
Exact integer 'EXACTINT' <,nbytes>
The largest double-precision number 'BIG'
The log with respect to base of BIG 'LOGBIG' <,base>
The square root of BIG 'SQRTBIG'
The smallest double-precision number 'SMALL'
The log with respect to base of SMALL 'LOGSMALL' <,base>
The square root of SMALL 'SQRTSMALL'
Machine precision constant 'MACEPS'
The log with respect to base of MACEPS 'LOGMACEPS' <,base>
The square root of MACEPS 'SQRTMACEPS'

- **DEVIANCE**(*distribution*, *variable*, *shape-parameter(s)* < ,*[EPSIV]*>)

Computes the deviance and returns a value

- **FACT**(*n*)

Computes a factorial and returns a value

- **PERM**(*n*<,*r*>)

Computes the number of permutations of **n** items taken **r** at a time and returns a value

- **PROBBNRM**(*x, y, r*)

Computes a probability from the bivariate normal distribution and returns a value

- **PROBMC**(*distribution, q, prob, df, nparms*<, *parameters*>)

Computes a probability or a quintile from various distributions for multiple comparisons of means, and returns a value

Character-String Matching Functions

The following RX functions and CALL routines provide character-string matching functionality. That is, they enable you to search for (and, optionally, to replace) patterns or characters in a string.

- **CALL RXCHANGE** (*rx, times, old-string*<*, new-string*>);

Changes one or more substrings that match a pattern

- **CALL RXFREE** (*rx*);

Frees memory allocated by other regular expression (RX) functions and CALL routines

- **CALL RXSUBSTR** (*rx, string, position, length, score*);

Finds the position, length, and score of a substring that matches a pattern

- *position*=**RXMATCH** (*rx, string*)

Finds the beginning of a substring that matches a pattern and returns a value

- *rx*=**RXPARSE**(*pattern-expression*)

Parses a pattern and returns a value

The ability to parse strings can be useful in many different ways. While some use these tools to explore, clean and modify their data, others use the same tools to automate their processes. The following example shows how a SAS log or the output from PROC CONTENTS can be processed to determine directory paths.

```
Parsing a SAS Entry Name from a Line of Text
By Jack Hamilton on SAS-L

data _null_;

length result $35.;
drop rx string;
retain rx;

if _n_ = 1 then
   rx = rxparse(`` <:> <$n "." $n "." $n ".program"> <:> to =2');
infile cards end=end;
input string $char80.;
call rxchange(rx, 2, string, result);
put result=;
if end then
   call rxfree(rx);
run;

data:
0jd#abc.def.xyz.program6834efghijklmn.op.qr.program2633
123defghijklmn.op.qr.program2633
hijklmn.op.qr.programsandmore
results:
abc.def.xyz.program
hijklmn.op.qr.program
hijklmn.op.qr.program
```

Variable Information Functions

The largest category of new functions supplies variable information. The information returned ranges from whether or not the variable is in an array, is character or numeric, to the name of the format or informat associated with the variable and its label. This category is actually two complementary sets of functions that perform the same task but with different arguments. The first set begins with the letter V and requires a variable name or array reference as its argument. The second also begins with a V, but ends with an X and requires a character string as the argument. For each of the V-X functions, SAS evaluates the argument to determine the variable name.

- **VARRAY** (*name*)

Returns a value that indicates whether the specified name is an array

- **VARRAYX** (*expression*)

Returns a value that indicates whether the value of the specified argument is an array

- **VINARRAY** (*var*)

Returns a value that indicates whether the specified variable is a member of an array

- **VINARRAYX** (*expression*)

Returns a value that indicates whether the value of the specified argument is a member of an array

VARRAY, VARRAYX, VINARRAY, VINARRAYX, VTYPE, and VTYPEX make determinations and return a specific value. VARRAY and VARRAYX determine if the specified name or expression is the name of an array. VINARRAY and VINARRAYX are used to determine if a specified name or expression is a member of an array. Both VARRAY and VARRAYX return a 1 if the argument is the name of an array and a 0 if it is not, but VARRAYX requires an expression rather than a name. The same is true for VINARRAY and VINARAYX, which determine if the name or expression is a member of an array.

Example:

```
an_array=varray(name);
an_array=varrayx(expression(x)) ;
in_array=vinarray(name);
in_array=vinarrayx(expression(x));
```

- **VTYPE** (*var*)

Returns the type (character or numeric) of the specified variable

- **VTYPEX** (*expression*)

Returns the type (character or numeric) for the value of the specified argument

VTYPE and VTYPEX are a little different in that they return the letter N if the variable is numeric and the letter C if it is a character variable.

Example:

```
v_type=vtype(name);
v_type=vtypex(expression(x));

if vtype(&varname )='N' then do;
      /* code for numeric processing */
  end;
  else do;
     /* code for character processing */
end;
```

The VLABEL, VLENGTH and VNAME function pairs are especially helpful tools when processing arrays. VNAME and VNAMEX return the name of the requested variable and VLABEL and VLABELX return any label associated with it. VLENGTH and VLENGTHX return the length at processing time.

- **VLABEL** (*var*)
Returns the label that is associated with the specified variable

- **VLABELX** (*expression*)
Returns the variable label for the value of a specified argument

- **VLENGTH** (*var*)
Returns the compile-time (allocated) size of the specified variable

- **VLENGTHX** (*expression*)
Returns the compile-time (allocated) size for the value of the specified argument

- **VNAME** (*var*)
Returns the name of the specified variable

- **VNAMEX** (*expression*)
Validates the value of the specified argument as a variable name

```
data a;
length x1-x3 $8;
label x1 = "first"
      x2 = "second"
      x3 = "third" ;
array x(3) x1-x3;
 x1 = 'abc';
 x2 = 'cde';
 x3 = '';
v_name=vname(x(1));
v_length=vlength(x(1));
x_length=length(x1) ;
v_label=vlabel(x(3));
put v_name= v_label= v_length= x_length=;
run;

results:
v_name=x1 v_label=third v_length=8 x_length=3
```

The remaining sixteen V functions return information about the format or informats associated with a variable. Because a separate function exists for formats and another for informats as well as the name or expression argument discussed earlier, we now have two sets of pared functions. Their tasks are to return the format or informat, the format or informat name, the format or informat length and the format or informat decimal value for the given variable. Those that are associated with formats begin with VFORMAT and while those associated with informats begin with VINFORMAT. As with the other V Functions, those ending with an X must receive an expression while those that do not end in an X must receive a name or array reference.

- **VFORMAT** (*var*)
Returns the format that is associated with the specified variable

- **VFORMATD** (*var*)
Returns the format decimal value that is associated with the specified variable

- **VFORMATDX** (*expression*)
Returns the format decimal value that is associated with the value of the specified argument

- **VFORMATN** (*var*)
Returns the format name that is associated with the specified variable

- **VFORMATNX** (*expression*)
Returns the format name that is associated with the value of the specified argument

- **VFORMATW** (*var*)
Returns the format width that is associated with the specified variable

- **VFORMATWX** (*expression*)
Returns the format width that is associated with the value of the specified argument

- **VFORMATX** (*expression*)
Returns the format that is associated with the value of the specified argument

- **VINFORMAT** (*var*)
Returns the informat that is associated with the specified variable

- **VINFORMATD** (*var*)
Returns the informat decimal value that is associated with the specified variable

- **VINFORMATDX** (*expression*)
Returns the informat decimal value that is associated with the value of the specified argument

- **VINFORMATN** (*var*)
Returns the informat name that is associated with the specified variable

- **VINFORMATW** (*var*)
Returns the informat width that is associated with the specified variable

- **VINFORMATNX** (*expression*)
Returns the informat name that is associated with the value of the specified argument

- **VINFORMATWX** (*expression*)
Returns the informat width that is associated with the value of the specified argument

- **VINFORMATX** (*expression*)
Returns the informat that is associated with the value of the specified argument

The following example illustrates how V functions VYPE and VFORMAT can be used to write a formatted value to another variable, while retaining the original format.

```
if vtype(name)='N' then do;
   new_var=putn(name,vformat(name));
end;
else do;
   new_var= putc(name,vformat(name));
end;
```

New Date and Time Functions

A common topic among SAS programmers is the different ways to determine and or define duration: roughly the amount of time passing between two points in time. In the past, most solutions involved the use of INTCK or INTNX, which have their strong and weak points. The new functions DATDIF and YRDIF should make the task of determining time duration easier.

- **DATDIF**(*sdate*,*edate*,*basis*)
Returns the number of days between two dates

- **JULDATE7**(*date*)
Returns a seven-digit Julian date from a SAS date value

- **YRDIF**(*sdate*,*edate*,*basis*)

Returns the difference in years between two dates

Both DATDIF and YRDIF use the arguments for start date, end date and basis. Start and End dates are very straightforward, but defining "basis" is more complicated. As per the on-line docs:

Basis identifies a character constant or variable that describes how SAS calculates the date difference. The following character strings are valid:

'30/360'
specifies a 30-day month and a 360-day year in calculating the number of years. Each month is considered to have 30 days, and each year 360 days, regardless of the actual number of days in each month or year. Alias: '360'
Tip: If either date falls at the end of a month, it is treated as if it were the last day of a 30-day month.

'ACT/ACT'
uses the actual number of days between dates in calculating the number of years. SAS calculates this value as the number of days that fall in 365-day years divided by 365 plus the number of days that fall in 366-day years divided by 366. Alias: 'Actual'

'ACT/360'
uses the actual number of days between dates in calculating the number of years. SAS calculates this value as the number of days divided by 360, regardless of the actual number of days in each year.

'ACT/365'
uses the actual number of days between dates in calculating the number of years. SAS calculates this value as the number of days divided by 365, regardless of the actual number of days in each year.

```
Example:
data _null;
   startdate = '11jul71'd;
   enddate = '11jul01'd;
   actday = datdif(startdate, enddate, 'act/act');
   days360 = datdif(startdate, enddate, '30/360');
   months = yrdif(startdate, enddate, 'act/act')*12;
   yr30 = yrdif(startdate, enddate, '30/360');
   yract = yrdif(startdate, enddate, 'act/act');
   yra_360 = yrdif(startdate, enddate, 'act/360');
   yra_365 = yrdif(startdate, enddate, 'act/365');
put  actday = days360 = months = yr30 = yract =
      yra_360 = yra_365 = ;
run;

results:
   actday=10958
   days360=10800
   months=360
   yr30=30
   yract=30
   yra_360=30.438888889
   yra_365=30.021917808
```

Missing and Error Functions

- **MISSING**(*numeric-expression* | *character-expression*)

Returns a numeric result that indicates whether the argument contains a missing value

Like several of the previously mentioned functions, the MISSING function returns an affirmative indicator of 1 if a variable contains a missing value and negative indicator of 0 if the value is non-missing. It works for both a character and numeric expressions.

- *character-variable*=**IORCMSG**()

Returns a formatted error message for _IORC_

IORCMSG returns the formatted error message associated with the most recently posted IROC code. A _IORC_ message is the value of an automatic variable created when the Modify statement or the Set statement with the KEY= option is used. This return code indicates whether the retrieval for matching observation was successful. A returned value of 0 indicates a successful execution; a -1 indicates an end-of-file error; and any other value indicates a non-match occurrence.

In the following program, observations are either rewritten or added to the updated master file that contains bank accounts and current bank balance. The program queries the _IORC_ variable and returns a formatted error message if the _IORC_ value is unexpected.

```
Example:
libname bank 'SAS-data-library';

data bank.master;
  set bank.trans;
  modify bank.master key=Accountnum;
  if (_IORC_ EQ %sysrc(_SOK)) then
    do;
      balance=balance+deposit;
      replace;
    end;
else
  if (_IORC_ = %sysrc(_DSENOM)) then
    do;
      balance=deposit;
      output;
      _error_=0;
    end;
else
  do;
    errmsg=IORCMSG();
    put 'Unknown error condition:'
    errmsg;
  end;
run;
```

Web-Based Functions

- **HTMLDECODE**(*argument*)

Decodes a string containing HTML numeric character references or HTML character entity references and returns the decoded string

- **HTMLENCODE**(*argument*)

Encodes characters using HTML character entity references and returns the encoded string

- **URLDECODE**(*argument*)

Returns a string that was decoded using the URL escape syntax

- **URLENCODE**(*argument*)

Returns a string that was encoded using the URL escape syntax

```
Example:
data _null_;
  text="This string contains characters !@#$%^& that must be
encoded";
  html=
    '<a href="/cgi-bin/broker.exe?_service=
      default&_program=test.echo.sas&text=
      '!!urlencode(text) !!'">Show encoded text</a>'
  ;
  put html;
  run;
```

which produces the following valid HTML hyperlink:

```
<a href="/cgi-
bin/broker.exe?_service=default&_program=test.echo.sas&text
=This%20string%20contains%20characters%20%21@%23%2
4%25%5E%26%20that%20must%20be%20encoded">Show
encoded text</a>
```

Urldecode() works the other way to decode these cryptic strings e.g.

```
data _null_;
text="%21Hello+World%21";
text=urldecode(text);
put text=;
run;
```

which produces:

```
TEXT=!Hello World!
```

Financial Functions

With SAS being used by virtually all of the major financial institutions, some financial functions were certainly in order.

- **CONVX**(*y,f,c(1), ... ,c(k)*)

Returns the convexity for an enumerated cashflow

- **CONVXP**(*A,c,n,K,k0,y*)

Returns the convexity for a periodic cashflow stream, such as a bond

- **DUR**(*y,f,c(1), ... ,c(k)*)

Returns the modified duration for an enumerated cashflow

- **DURP**(*A,c,n,K,k0,y*)

Returns the modified duration for a periodic cashflow stream, such as a bond

- **PVP**(*A,c,n,K,k0,y*)

Returns the present value for a periodic cashflow stream, such as a bond

- **YIELDP**(*A,c,n,K,k0,p*)

Returns the yield-to-maturity for a periodic cashflow stream, such as a bond

Interesting Uses of Functions

The use of functions is often limited only by the imagination, creativity and need of the programmer. The following code shows how various functions can be combined to solve dilemmas and make life easier.

- TIP 00270 from WWW.SCONSIG.COM ****/

A COMPRESS function for Macro Variables
By Peter Crawford

```
%macro Remove__ ( STRING, REMVECHR );
 %sysfunc( compress( &string, &REMVECHR));
%mend Remove__ ;

/*** Sample Call to Invoke ***/
%let  string__ = %remove__( "PLA" Derivative, "");

%put &string__ ;
PLA Derivative
```

You will need to be careful in compressing quotes (single or double) - make sure you surround your preference (quotes to be compressed) with a pair of opposite quotes (ie, "" or """).

- TIP 00136 from WWW.SCONSIG.COM

To Generate Nine Variables from a Nine Length Character String
By Paul Dorfman

```
data manyvar2(drop=addr len);
  array v(10) $1;
  addr = addr(v(1));
  len  = dim(v);
  do until (eof);
    set in end=eof;
    call poke (string, addr, len);
    output;
  end;
run;
```

About the Authors

Gary McQuown is a SAS Quality Partner with Data and Analytic Solutions, Inc. of Fairfax VA. He has previously presented at NESUG and SESUG.

Dorothy Brown is a SAS Consultant currently on contract at Sprint Communications World Headquarters in Kansas. This is her first presentation.

Author Contact

Gary McQuown
Data and Analytic Solutions, Inc.
10502 Assembly Drive, Fairfax, VA 2200
mcquown@DASconsultants.com
www.DASconsultants.com

Dorothy Brown
819-201 Cameron Village Drive, Matthews, NC 28105

Bibliography:

William F. Heffner, "DATA Step in Version 7: What's New?" SUGI 23 Proceedings

Denise J Moorman and Deanna Warner, "Updating Data Using the Modify Statement and the KEY=Option" SAS Observations

Mike Rhoads, "Hidden Nuggets in Version 8: New Informats, Formats and Functions" SUGI 23 Proceedings

SAS Institute Inc., Changes and Enhancements to Base SAS Software Release V8.1, Cary, NC: SAS Institute,

Trademark Information

Creating Regional Maps with Drill-Down Capabilities

Deb Cassidy
Cardinal Distribution, Dublin, OH

ABSTRACT

SAS/GRAPH® includes many maps which are very useful. But what if you need a map that truly represents your company? Many companies combine several states into a region. My company does not even follow state boundaries so some states are split across more than one region. This presentation will show you how to turn the county-level map into a region-level map. Of course, with the advent of ODS, no presentation is complete without discussing ODS. The presentation will also cover the extra steps so you can have drill-down capabilities with your new regional map. This presentation uses SAS/GRAPH. It is assumed the audience has some SAS knowledge but may not have used SAS/GRAPH maps or ODS.

CREATING A REGIONAL MAP

My company currently divides the United States into 12 regions based on sales territories. Some states are split across regions. In reality there are some areas of the country which are shared by two regions. It was determined that all counties were to be assigned to a single, primary region for analytical purposes. However, when it came time to assign the states of New York and New Jersey, it was determined that it was easier to create a separate region for those states that to decide which was the primary region. Hence, our regions are unique so a pre-existing map would not work.

I already had a dataset with each state and county combination although I could have used the COUNTIES dataset provide with SAS/GRAPH as my starting point. The following code would create a unique list of states and counties from the map dataset.

```
PROC SQL;
  CREATE TABLE COUNTY_LIST AS
  SELECT UNIQUE STATE, COUNTY
  FROM MAPS.COUNTIES;
```

The dataset you use must use the same numeric state and county codes used in the SAS/GRAPH COUNTIES dataset. You simply need to add a variable indicating the region. As explained below, I used a numeric variable to represent region.

The COUNTIES dataset included with SAS/GRAPH is merged with my region dataset. You may think that is all you have to do but if you try to use this dataset in PROC GMAP you will discover a problem - the map is reversed. Some of the SAS/GRAPH maps are projected and others are not. If they are not, you need to run PROC GPROJECT which will convert latitude and longitude into Cartesian coordinates which are used by PROC GMAP. This PROC also has several options for controlling the angle of the maps.

In most cases, you'll also want to use PROC GREMOVE. This will remove the county and state boundaries within each region. When mapping the counties in the entire United States, the map will be very cluttered if you don't perform this step. There is one other step that you may want to consider. The map dataset is very detailed and shows the many curves the borders of the states have. This is particularly noticeable on states which have a body of water as a boundary. The COUNTIES dataset includes a density variable which can be used to eliminate some of the detail. This will smooth out the map which can be beneficial which when showing a large area. The code to do these steps is shown at the end of the paper.

ADDING DRILL-DOWN CAPABILITY

The JAVA applet requires a "jar" file or a text version of the map. Since you have modified the original map dataset, the supplied file is not usable. Therefore you need to create your own. Fortunately, SAS provided a macro for this conversion. The macro will create a text version of the map for use by the applet. The macro is shown at the end of the paper. The text file ends up in the following format.

```
REGION,SEGMENT,X,Y
numeric,numeric,numeric,numeric
1,1,0.2770831409,-0.005568895
…rest of coordinates for Region 1
2,1,0.2827390409,0.0396897128
…rest of coordinates for Region 2
…rest of coordinates for other Regions
```

So far you have created the custom map you need to show regions and saved it as a text version. The next step is to generate the map using ODS. The options/parameters on the ODS statement provide the information needed so it can be used with the JAVA applet. The DRILLPATTERN parameter specifies where to look for the web pages. There needs to be a separate page for each value on your map. The example shown uses files on your pc. On a web site, this would be a web address.

There were two problems I encountered when specifying the DRILLPATTERN. At the time this paper was written, I had not confirmed if these are indeed problems or if there was simply something about my data or my system causing the problems. First, when I used a character variable for my regions, the drill-down did not work. Switching to a numeric variable solved that problem. Second, I couldn't simply use c:\{regname}.htm. I had to have a prefix before the region value in the file names of the web pages.

I also encountered another problem with the map. Some of the states have lakes within the state boundaries. Depending upon the states selected, the output was either blank where the map should be or a distorted map appeared. Reducing the density eliminated the distortion problem. However, the output was still blank if I left in the states of Florida, Utah, New Jersey and Oregon regardless of what I used for density. At the time this paper was written, I was still investigating ways to eliminate the lakes in these states. So contrary to any ideas you might have based on the final map, Florida has not been dropped from the United States!

FINAL MAP

The final map is shown on the next page. Also shown is the original map with state boundaries to show you how things have changed (the one with county boundaries was far too cluttered to attempt to print here). When you move your cursor over each region you will get a pop-up box giving you the values of the variables used to created your map. When you click on a given region, you will go to the web page for that region.

CONCLUSION

SAS/GRAPH provides you with a variety of maps which can then be used to create your own custom map. ODS offers additional functionality beyond simply displaying the graph.

TRADEMARK INFORMATION

REFERENCES

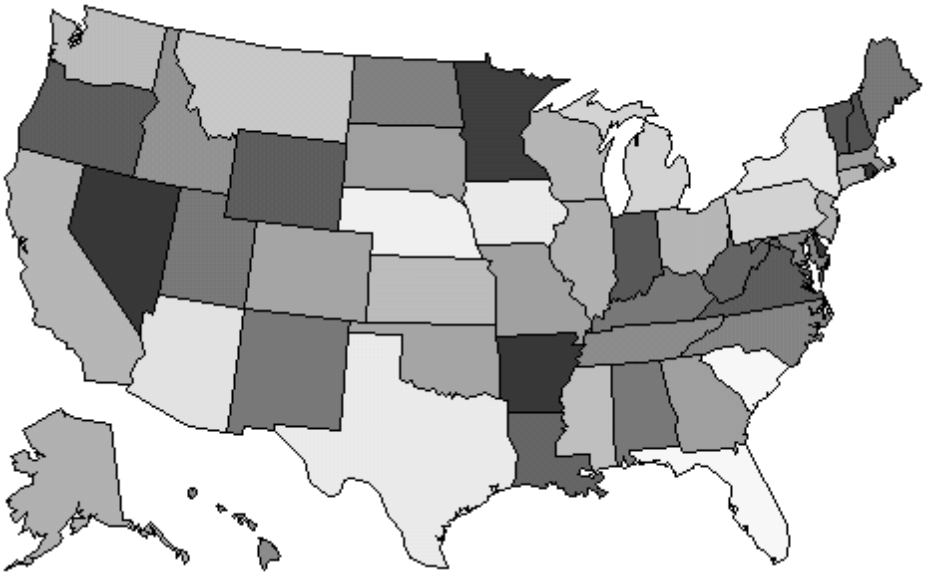
SAS Institute Inc., SAS/GRAPH® Software: Reference, Version 6, First Edition, Volume 2, Cary, NC: SAS Institute Inc., 1990. 664 pp.

SAS Institute Inc., TS-625, Cary, NC: SAS Institute Inc., 2000.

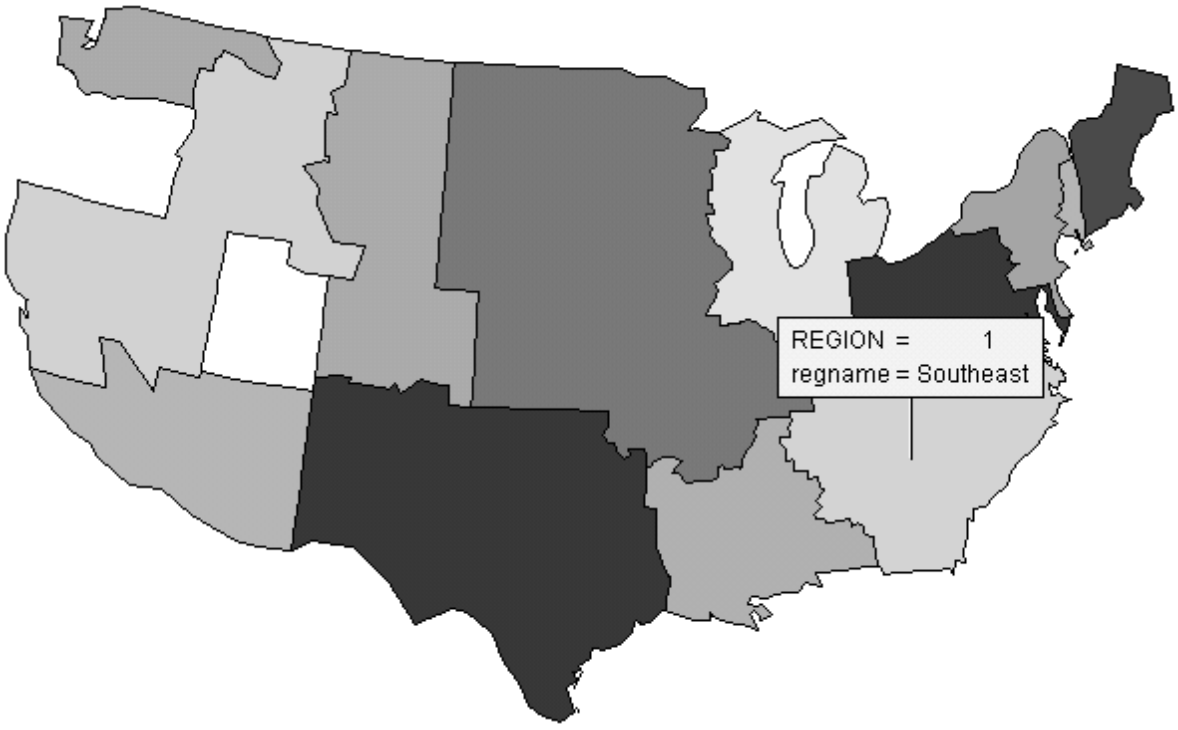
CONTACT INFO

Deb Cassidy
Cardinal Distribution
7000 Cardinal Place
Dublin, OH 43017
deb.cassidy@cardinal.com
614-757-7136

ORIGINAL STATE-LEVEL MAP



FINAL REGION-LEVEL MAP



CODE

*** ASSIGN ANY NECESSARY LIBNAME STATEMENTS TO ACCESS YOUR SAS/GRAPH MAPS OR YOUR DATA;

*** Specify the file location to store the text version of the map. This must be the same path as the JAR files which are supplied with SAS/GRAPH. A macro variable is used since this location is used more than once in the program;

```
%let dataloc=c:\program files\sas institute\shared files\applets\SSU_Paper.txt;
```

**** This is the macro which will create the text version of the map;

```
%macro exportds(dataset,file);
%let delimiter=,;
proc contents data=&dataset out=temp;
run;

/*write out the variable names first*/
data a;
  set temp end=last;
  file "&file" dsd dlm="&delimiter";
  put name @;
run;

/*write out the variable types next*/
data a;
  set temp end=last;
length ctype $9;
file "&file" mod dsd dlm="&delimiter";

if type=1 then ctype='numeric';
  else ctype='character';
put ctype @;
run;

/*write out the data*/
data a;
  set temp end=last;
file 'temp.out';
  if _n_ = 1 then do;
    put "data a; set &dataset;";
    %let out="file '&file' mod dsd dlm='&delimiter';";
    put &out;
    put 'put ' @;
  end;
put name @;
  if last then do;
     put ';run;';
  end;
run;
%inc 'temp.out';
```

```
%mend;

*** PROC FORMAT is used to assign names to each region I have created;
PROC FORMAT;
VALUE REGNAME
  1='Southeast'
  2='Mid-Atlantic'
  3='Mid-America'
  4='Gulf'
  5='Northwest'
  6='Pacific Northwest'
  7='Pacific Southwest'
  8='North'
  9='Northeast'
  10='Southwest'
  11='Midwest'
  12='NY/NJ';


*** INN.MY_REGIONS is my dataset which contains the region assignment for each state and county;
*** For my purposes, Hawaii, Alaska and Puerto Rico were to be eliminated.  Florida, Utah, New Jersey and;
*** Oregon were also deleted due to the problems encountered with the lakes.  These last 4 states will be included ;
*** once that problem is resolved.  The numeric codes for each state are in the SAS/GRAPH manual.;

data work.mydata ;
   length regname $1020;
   set inn.my_regions;
   if state not in (2,15,72,12,49,41,34);
   regname=put(region,regname.);
run;

*** This step merges the SAS/GRAPH supplied COUNTIES map with my data to add the region variable;
*** As noted above, some states were deleted.  The DENSITY was restricted to have a cleaner looking map;
*** I experimented with different density cutoffs before deciding to use <1;
data mymap;
   merge maps.counties inn.my_regions;
   by state county;
   if state not in (2,15,72,12,49,41,34);
   if density<1;
run;

*** The map dataset must be sorted before using GREMOVE to eliminate the county and state boundaries;
*** In PROC GREMOVE, the BY statement specifies the boundaries you want to keep while the ID;
*** statement specifies the original boundaries;
proc sort data=mymap;
   by region;
run;

proc gremove data=mymap out=mymapR;
   by region;
   id county state;
run;

***PROC GPROJECT will convert latitude and longitude to Cartesian coordinates. If you are using maps other;
```

```
*** than the US, you may wish to use some of the options to modify the projection;
proc gproject data=mymapR out=mymapP;
  id region;
run;

*** This macro will convert the map dataset to the text file which is required for the JAVA applet.
%exportds(mymapP,&dataloc)


*** Close ODS listing to be able to route the output to the ODS HTMLfile;
*** The HTML file and the DRILLPATTERN files can be a web address.;
*** The DRILLDOWNMODE is set to HTML so a click will drive the drilldown;
*** MapLocation is the location of the text version of the map;
ods listing close;
ods html file="f:\dcassidy\DRILL_DOWN_MAP.htm"
   parameters=("DRILLDOWNMODE"="HTML")
   parameters=("DRILLPATTERN"='file:///c:\REG{&regname}.htm')
   parameters=("BACKCOLOR"="WHITE")
   parameters=("MapLocation"="&dataloc");

***This statement sets the device to Java to produce an HTML file that that will appear in your web browser.;
***It also sets up customizations for the graph.;
goptions reset=all
        dev=java
        cback=white
        border
        gunit=pct
        htext=3
       colors=( green, red, orange, pink, blue, cyan, magenta, lime, purple, yellow, violet, indigo);

***This proc produces the graph.;
title1 "Acme Distribution Regions";
proc gmap map=mymapP data=mydata all;
  id region;
  choro regname /   discrete
                coutline=black
                nolegend
                des='Acme Distribution Regions'
                name='Regions';
footnote 'Click on region to go to regional page';
run;
quit;

***  These statements close the HTML file and open the channel to the graph output windows.;
ods html close;
ods listing;
```

Paper P612

Structuring Base SAS for Easy Maintenance

Gary E. Schlegelmilch, U.S. Dept. of Commerce, Bureau of the Census, Suitland MD

(Note: this is an update of the paper W2282-26, presented at SUGI 26.)

ABSTRACT

Computer programs, by their very nature, are built to be flexible. A program is no more than a series of versatile building blocks, stacked in such a manner to produce a desired result. However, the user requirements change over time, and so must the program change to reflect those new requirements. There are a number of ways to lay out a program, so as to make it easy to find the places where change is required. And as always; it is far easier to start with a solid foundation than to try and retool after the fact.

INTRODUCTION

There is no single "perfect" set of rules for structuring program code. In this paper, I have made a series of suggestions and offered a number of observations as to habits I have tried to form over the years. I have found them to be effective and helpful in creating code that is easy to read, maintain, and understand.

SAMPLE CODE: LOGGING INTO A REMOTE SITE

Imagine, if you will. A new program, named MERGERMT.SAS is to be built on a VAX Alpha platform. The program is to accept a file from a remote site to update a local master. Space is at a premium on the local site, so the decision is made to log into the remote site needed, and update from the remote transaction file without copying it to the local site.

The files for interface are all in a uniform format. The current requirement calls for the transactions to be coming in from a UNIX platform, but could be coming from a number of different platforms in the future. So, I started with a simple piece of code that could log into one remote site, using the TCP/IP method, and establish the SAS library containing the data I needed to access.

```
/*  Log into the UNIX
    platform            */

filename RLINK
         'LIBRARY:TCPUNIX.SCR';
%let UNIX01=184.131.137.13;

/* IP address for the node */

options comamid=TCP
        remote=UNIX01;
signon;
libname REMOTE
        '/sys/update'
        server=UNIX01;
```

Note: TCPUNIX.SCR is a script that contains the SAS/CONNECT SIGNON/SIGNOFF script for connecting to any UNIX host via the TCP access method. The script is copyright ©1990 by SAS Institute Inc. The script is associated with filename RLINK to connect with the SIGNON process. For purposes of this paper, I have assumed there to be a central library for utility files and programs, addressed by the VAX logical LIBRARY:.

The `%let` is a Macro statement, which sets a macro variable to the value specified. This is required to be set to the IP address of the platform to be contacted, and is used internally by SAS/CONNECT.

Tip: To determine the IP address of a given node or platform, enter `grep <node> /etc/hosts` on UNIX, or `PING <node name>` at a DOS prompt on a Windows platform. These commands will return the IP address for the requested node name, if it is connected to the network on which you are currently working. There is a utility program, Multinet, available on some VAX/Alpha systems; `multinet nslookup <node>` will yield a IP address as well.

Notice that the few lines of the module already reflect a few basic structuring items. Comments are included, both to indicate the function of the block, but also to point out uncommon features in the code. Blank lines are added to separate the code into small, readable blocks.

I also use both upper- and lower-case when writing my code. This, of course, is a matter of personal preference; SAS does not distinguish between upper- and lower-case lettering. However, you will note that I put all SAS constructs in lower case, and data items in upper-case. In Interspeak, the common dialect for Internet communication, common text is in lower-case, and items to be accentuated or "shouted" are in upper-case. To my way of thinking, the SAS code is set, checked by the interpreter, and as such does not need to be accented; the interpreter will flag any errors. On the other hand, user-defined items like data names and libraries are frequently things that deserve extra notice during debug and maintenance. So, I accent them by putting them in upper-case. You can see even from the small example here what stands out.

Another way you can help yourself in maintaining the program is to use field names that are as clear as possible. If a field is used only as a macro to carry a text message to the log in the event of an error, don't call it TXT or X2; call it ERRTEXT. That way, the program becomes more self-documenting. That's why the program takes the generic variable `sysparm`, and moves it into the more recognizable `node`.

In Version 6.12 and earlier, it could become difficult to clearly document data items solely by name. So, take the time to comment the field as soon as possible. An old documentation standard is to define a term as soon as it is used; it is helpful to do the same in your program. For instance, if you needed to write an inventory program for a retail store:

```
length NBRONHND 6.
  /*  Number of items
      on hand          */
       NBRONORD 6.;
  /*  Number of items
      on order         */
```

In Version 8 on, the programmer can make use of the 32-character field names, including underscores. So the same names in the example could be named NBR_ON_HAND and NBR_ON_BACK_ORDER, respectively.

Back to MERGERMT.SAS. Now, once this small login module is functional, let's take it to the next level; getting set up for different nodes. For the sake of brevity, we'll just use two nodes here. A DATA Step is used here since the `abort` statement and `in` operator must be used within a DATA Step.

```
/* ensure the NODE field is in
   uppercase to ensure correct
   comparisons  */

%let NODE=%upcase(&SYSPARM);
options comamid=TCP
        remote=&NODE;
%let UNIX01=184.131.137.13;
%let UNIX02=184.131.137.18;

data_null_;
  if "&NODE" in
    ("UNIX01","UNIX02")
  then do;
    signon;
    libname REMOTE '\sys\update'
            server=&NODE;
  end;
  else do;
    put 'INVALID NODE NAME '
'ENTERED.  LOGIN TO REMOTE '
'NODE ABORTED.';
    put "&NODE";
    abort;
  end;
run;
```

One of the main uses of indentation is to show a more detailed level of program flow. In the above example, it's easy to see at a glance that the `signon` and libname statements are performed within the boundaries of the `if`. The positive benefits become more noticeable in later steps with the `%macro` statements.

The initial requirement calls for the remote platform to be UNIX; so, the same `libname` statement can be used for all valid nodes. However, if the requirement changes later to include different platforms, the `libname`

statement will have to reflect the directory structure of the remote platform. Perhaps there would be a second `if...in` statement for all nodes of a different platform, reflecting the appropriate directory structure for the platform; or, in the event of different directories, there might be an `if` for individual nodes.

Of course, the `libname` statement must follow the `signon`; prior to successful completion of the `signon`, the requested node isn't available to the local host, and results in an error.

Notice that I always put each statement on a separate line. This not only makes each line easier to read, by making it more clearly defined; but it lends itself more easily to future expansion of the code.

If you use the Data Step Debugger to find flaws in your code, it becomes even more significant. If multiple statements are on a single line, it is more difficult to follow the flow of the code. With a separate line for each statement, you can readily see which line is being processed by the Debugger, and hence more easily find the flawed line.

Another time-saver; when an edit hinges on the value of a variable, display the value of the variable as a part of the abort message. It saves time when trying to find out why a process failed.

The code above was the second version of the program. As I was always running the program interactively, it was no problem to set the NODE variable manually at runtime. And an error routine was added in, just in case of a typo in entering the node I wished to use.

This fulfills the immediate requirement; it permits the program to log into either node, but requires a program change if login to a different node is desired. The next step is to make it more versatile, and not require the program change each time.

```
filename RLINK
'LIB:TCPUNIX.SCR';

%macro LOGNODE(NODE);
  %let UNIX01=184.131.137.13;
  %let UNIX02=184.131.137.18;

  data _null_;
    if "&NODE" in
      ("UNIX01","UNIX02")
    then do;
      ::
      ::
      ::
  run;;
%mend LOGNODE;

%LOGNODE(&SYSPARM);
```

Note: the :: here represents the code from the previous example, so as not to take up the space in this paper with repetitive code. Again, the code is indented one level, to show that it is subordinate to the `%macro` statement. To further document, the `%mend` statement includes the name of the Macro being used; it's not necessary, but it does clearly show where the macro ended.

Using a Macro to define repetitive code has a number of advantages. Compile time is reduced slightly, as the code within the Macro is only defined once. Your program will be smaller, for the same reason. Most importantly, the overall maintenance is reduced; in the event that future enhancements are required, there will only be one change to make, instead of searching the program to ensure they are all made in the same way in each place.

So, how does the name of the node get into the program? By adding a parameter to the runstream;

```
$ SAS/SYSPARM="UNIX01" -
    PROGRAM1.SAS
```

SYSPARM is an automatic macro variable; that is, it is already defined by SAS. A parameter passed in this manner is automatically passed into the system-defined macro variable SYSPARM, and requires no further definition by the user.

Notice here that the indentation becomes more significant. In each of the `if... do` blocks, the terminating `end` is aligned with the `if`. What happens if the statement is TRUE is quite evident. Moreover, in the case of debugging, it narrows down the focus of what is or is not happening in the program. As an example; if the "INVALID UNIX ID" prints in the log, you can be certain that the content of the variable &node is neither UNIX01 nor UNIX02.

Important note to all programmers; no matter how many times you say "It should (or shouldn't) be", it rarely changes the program code or its functionality. It's much better to start looking at blocks of code and following the flow of data. That is a great deal easier with a little structure.

So, now we have a program that successfully allows the program to establish login and a remote site of UNIX01 or UNIX02 to your session. It can be executed with different parameters without changing the regular version of the program. Then, it occurs to you; other people in your section are also creating remote sessions in their programs, plus four other programs on your development schedule will require the same type of code. You could cut-and-paste it into other programs, and e-mail the code to everyone; or create a reusable module.

Take the macro code, without the invocation, and place it in a separate file called LOGNODE.MAC. Add a block of comments to the top, perhaps something like this:

```
/*------------------------
   LOGNODE.MAC

   This routine will permit
   the user to pass a node
   name as an argument and
   log into that node as a
   remote host.
------------------------ */
```

For purposes of this example, we'll assume that all reusable code modules are stored in a centralized directory, and they are accessed via our VAX Alpha logical called LIBRARY:. The naming convention will of course change depending on the platform; it might as easily be /office/common/lib on UNIX, or C:\lib on the PC. To include that code into the program, use the `%include` macro, which reads the named code into the body of the program, and runs it as if it were part of the main program.

So now the program might look something like this:

```
/* ------------------------
   MERGERMT.SAS

   This program will merge
   a file on a remote node
   into the master dataset
   on the VAX Alpha.
------------------------ */

/*  use options that place
    the value of macro
    variables, and the
    macro code as it
    executes, into your
    LOG file.           */

options mlogic mprint
        symbolgen;

/*  incorporate the macro
    file to log into a remote
    node into the program. */

%include 'LIBRARY:LOGNODE.MAC';

%LOGNODE(%upcase(&SYSPARM));

/*  update the existing
    dataset, located in
    the master file
    directory, with the one
    from your remote host.
-------------------------*/

libname FILES
        'FILE_DEV:[MASTER]';

data FILES.MASTER;
  update FILES.MASTER
         REMOTE.TRANS;
  by ID CATEGORY;
run;
```

The final product takes advantage of the modular nature of SAS, laying out each function in a separate, testable module. The program is easy to read, enhance, and maintain. Moreover,

if someone in your area needs to log into UNIX99, VAX045, or UNISYS37, those can easily be added to the LOGNODE.MAC macro at no impact to the calling program. Everyone wins.

For example, it could easily be enhanced to read a file or dataset containing all available nodes and their corresponding IP addresses, and reduce the code overhead further.

More importantly for the immediate process, we can now expand on what was started by putting structured code in the macro. Now space and time is reduced for multiple programs, not just the originating one. Also, maintenance done in the .MAC routine is immediately accessible to any program using the module, without the compile or relink required by other languages.

Now, let's look at the same code without the benefit of the lines we added for readability and maintainability:

```
OPTIONS MLOGIC MPRINT SYMBOLGEN;

%LET ND=%UPCASE(&SYSPARM);
%IF &ND=UNIX01 %THEN %DO;
%LET UNIX01=184.131.137.13;
OPTIONS COMAMID=TCP
REMOTE=UNIX01; SIGNON; %END;
%ELSE %IF &ND=UNIX02 %THEN %DO;
%LET UNIX02=184.131.137.18;
OPTIONS COMAMID=TCP
REMOTE=UNIX02; SIGNON; %END;
%ELSE %DO;
%LET TEXT=INVALID UNIX ID
ENTERED.  LOGIN TO UNIX
ABORTED.;
%PUT &TEXT; %PUT &ND;
DATA _NULL_; ABORT; RUN;
%END;
LIBNAME REMOTE '\SYS\UPDATE'
SERVER=&SYSPARM;
LIBNAME FILES
'FILE_DEV:[MASTER]';
DATA FILES.MASTER;
UPDATE FILES.MASTER
REMOTE.TRANS;
BY ID CATEGORY; RUN;
```

This program is precisely the same, functionally, as the structured one above. It's fairly easy to see which would be the easier one to debug and maintain. Or, to put in another light; say you write this program, and it runs fine for a year. Then, the decision is made to add some new features. A programmer could tell at a glance where the changes might be required in the first program – and have to re-analyze the second.

In fact, that alone is a good test of a program's structure. Minor requirements changes should translate to minor changes to the program. If a minor requirements change means an overhaul of the program; then the program's basic structure should be examined. And there's no better time to do that, than when it's first laid out.

STRUCTURING FOR EFFICIENCY – WITHOUT LOSING READABILITY

Sometimes, you can make existing code clearer to read, and more effective at the same time.

Take this example. One of my programs was using a DATA step to reduce the size of a dataset prior to sorting. It seemed sensible to sort only the data I actually needed. So, I wrote a very simple DATA step prior to my sort:

```
data WORK.TEMP;
  set DATA00.TRANSACT;
  if TYPE='1' or TYPE='2';
run;

proc sort data=WORK.TEMP
          out=WORK.SORTED;
  by ID TYPE;
run;
```

Nice and simple. However, another member of my group suggested something even cleaner and easier to read;

```
proc sort
     data=DATA00.TRANSACT
     out=WORK.SORTED;
  by ID TYPE;
  where TYPE in ('1','2');
run;
```

Much better. The new version of the code (a) eliminated the storage of one temporary dataset (b) reduced the physical I/O of reading and writing from the input dataset twice, and (c) made more clear the intention of the routine. It also makes the routine easier to update. One or two IF statements aren't hard to read or follow, but if the new requirements call for 15 different TYPE's, the IN statement makes the intent of the WHERE clause very clear.

CONCLUSIONS

Any code, regardless of the language used, can be written for greater understanding. SAS tools lend themselves well to modular programming, permitting an ease in adding and changing functions of the programs in the future. Developing reusable modules saves coding time at the start of a project, and maintenance time in the future. Most importantly, building well-structured programs is a source of justifiable pride for the developer, because well-written code endures – while poorly written code that is difficult to maintain and use is inevitably discarded.

REFERENCES

Aster, Rick, Professional SAS Programmer's Pocket Reference, 2nd Edition, 1998, Breakfast Books.

SAS Companion for Microsoft Windows Environment, Version 6, 2nd Edition, SAS Institute Inc.

SAS/CONNECT Software, Usage and Reference, Version 6, 2nd Edition, SAS Institute Inc.

ACKNOWLEDGEMENTS

My thanks go to Ian Whitlock, for his insight and support in the preparation of this paper.

CONTACT INFORMATION

Gary E. Schlegelmilch
U.S. Dept. of Commerce, Bureau of the Census, ESMPD/MCDIB
Suitland Federal Center, Rm. 1200-4
4700 Silver Hill Road
Suitland MD 20746
Email Gary.E.Schlegelmilch@census.gov

Taming the Chaos: Managing Large SAS/AF Applications Using Programming Standards and the Source Control Manager of Version 8 of the SAS System

C. Michael Whitney, Motorola, Austin, TX

ABSTRACT

The use of programming teams offers both advantages and disadvantages when compared with individual programming efforts. The team approach allows a wide range of programming skills and problem-solving perspectives to be applied to a project, and may shorten development time. On the other hand, team-developed projects are often marred by differences in programming styles among developers, resulting in inconsistencies in the use of variable names, levels of documentation, and user interface design. Further, without some form of code management, the risk of inadvertently overwriting the work of other team members is always present. All of potential problems can make development and maintenance a chaotic and frustrating experience.

This paper will discuss how these problems can be minimized, if not eliminated, by applying some basic programming standards, and by making use of the new Source Code Management system that has been released with SAS version 8.

SAS products discussed in this paper include SAS/AF, SCL, and the Source Control Manager (SCM). The coding practices discussed in this article are applicable to all versions of the SAS System, on all platforms, and are aimed at developers with moderate to advanced SAS/AF & SCL experience. The code management portion is applicable only to the version of the Source Control Manager (SCM) that was released with SAS version 7. Earlier experimental versions of the SCM are not covered.

INTRODUCTION

Programming standards are essential for producing high quality, easily maintained software applications. The larger the application, the more this holds true. Using a code management system, such as the Source Control Manager provided with SAS version 8, further enhances maintainability. Such a system enables a programmer to check code modules in and out of a software library during development, or when the code needs updating, in much the same way a book is checked out from a library. Modules that have been “checked out” cannot be modified by others until they have been “checked in” by the current programmer.

The purpose of this article is to provide a framework for high quality, structured, and easily maintained SAS software. Uniformity of software development is essential in creating and maintaining computer systems that operate at peak efficiency. Due to space constraints, only the Top Down development approach is discussed. This is the method most applicable to the programming practices employed at most organizations. However, the principles discussed here are equally valid when applied to the Object Oriented development.

Recently, the Motorola Semiconductor Products Sector (SPS) unified its various Information Technology (IT) divisions into a global unit, rather than each SPS factory having its own IT team working independently of the others. One of the many benefits of this has been a pooling of knowledge as programmers from differing IT teams have been brought together. This has allowed us to take the best programming techniques used at the formerly independent sites, including standards, and come up with a good unified set of programming practices for our teams.

The guidance presented here has been collected by the author over the past eleven years, working for the US Air Force, a consulting firm, and the Motorola SPS IT Engineering Analysis Tools (EAT) team. It is applicable to any organization, be it governmental, corporate, or educational.

To be effective, programming standards should apply to all applications created in your organization. Applications created without standards applied to them should be brought up to the standard as maintenance is performed, if possible.

SOFTWARE DEVELOPMENT

SOFTWARE DEVELOPMENT LIFE CYCLE

The software development life cycle begins with a determination that a software application is required for customer support and ends with the cataloging of the application into the production libraries. In the SPS Engineering Analysis section, most projects begin when a device or product engineer at a factory requests a new statistical analysis tool, or the addition of new features to existing applications.

The development life cycle identifies important phases associated with the process of developing software. When completed, each phase should significantly reduce future maintenance costs and minimize the chance for software errors in the final product. The amount of detail required for each phase should be directly related to the size and complexity of the project. Large, complex projects require more extensive documentation than do small projects.

OVERALL LIFE CYCLE

There are several major phases in the software development life cycle. Figure 1 provides a summary of this cycle. The following is a brief description of the major phases:

1. **Gather and Define Requirements** - The first task is to determine if existing software can be used to fully or partially meet the customer's requirements. Analysis and research are the foundation on which the rest of the project lies.

2. **Prepare Initial Design** - If software development or modifications are necessary, the programmer formulates the initial requirements and designs documentation. The more detail that can be provided, the more easily understood the design will be. This design should be presented to the customer for approval, ensuring that it meets the customer's requirements.

3. **Formal Design and Design Review -** The detailed design is an effort to construct a concise, logical solution to the problem, which the programmer can easily translate into code. The design should be approved before coding begins.

4. **Test Plan, Coding, and Testing –** The appropriate coding standards should be followed for all production software. The development and execution of a test plan is essential to ensure that the customer receives a quality product (both software and data). Testing should occur any time that changes are made to the code. After testing is completed,

the software should undergo a review or walk-through prior to production.

5. Final Code Inspection - This is an intense, line-by-line review of the software prior to cataloging.

6. Cataloging - This process places approved software into the Production Software Libraries. It is extremely advantageous to place production code in libraries where only the software librarian has write authority. This will prevent accidental corruption of the source code and ensure that everyone is running the same version for production.

Top-Down Structured Design

Software should be developed in a top-down structured manner. Top-down design begins with formulating the solution in terms of generalized statements. After the general algorithm is developed, it can be refined by adding the details that are necessary to perform the general actions. For a complicated problem, this refinement process may be repeated several times, with each version containing more detail than the last. The design proceeds from the top (most general) to the bottom (most detailed), with the resulting design reflecting the nature of the problem. Top-down structure makes the program highly readable and easier to follow.

Structured software consists of separate functional modules. A module is a subportion of a program and is composed of a bounded group of instructions with a single identifier. A module may be a subroutine, function, or driver. In SCL terms, a module could be a either an SCL program, or a labeled section within an SCL program. Labeled subsections provide great modularity! Modules may call other modules, and in many cases, several modules may be required to complete a single function. Modules are designed with control flowing from the top to the bottom.

Top-down structured software should adhere to the following characteristics:

- **Cohesion** - All modules should perform single functions or small, related functions that require common data and pass information from one step to another.
- **Coupling -** Connections between modules must be obvious and should be minimized as much as possible for simplicity. Data will be passed as arguments between modules, when applicable.
- **Limited data exchange** - A minimum amount of data should be passed between modules. If only two variables out of a 100 variable data set are needed by a particular DATA step, use the DROP= or KEEP= data set option to eliminate those variables not needed. Similarly, only those SCL variables and list pointers that are required for a particular method to perform its given function should be passed between methods.
- **Exit and entry points** - Modules should have one entry point and one exit point.
- **Span of control** - Never let a module (except the program driver) directly control or call more than seven subordinates.
- **Scope of effect** - Subordinate modules are modules that make decisions so that other modules can complete a function. The scope of effect of any module includes all subordinate modules that are necessary for the completion of the module's function.
- **Module size** - Modules should consist of no more than four pages (200 lines) of executable code, minus comments. Modules should not perform more than one unrelated function, but may handle more than one related function for the program. If a module grows larger than 200 lines, check to see if the module is accomplishing more than a single function. If so, attempt to break it down into single-function modules. However, do not cut a single-function routine into multiple pieces simply for the sake of module size.
- **Independence -** The execution of the module is completely independent and does not depend on anything that occurred in previous invocations.

Top-down structured software is a direct result of the use of structured techniques and tools such as data flow diagrams, flow charts, pseudo code, structure charts, data dictionaries, and organized documentation. It requires forethought and work from everyone.

Top-down structured design has the following advantages:

- It allows for reusable modular coding, testing, and implementation.
- Design problems are detected early, when they are cheaper and easier to correct.
- Module development allows the programmer to concentrate solely on individual modules while treating other modules as black boxes.
- Fewer programming errors are likely to be made due to the module's significantly reducing program complexity.
- Modularity makes debugging easier. Problems are quicker to isolate in any particular module, shortening debug time.
- Modules are used more easily by other programs.

CODING STANDARDS

DOCUMENTATION

All program modules should be documented. The documentation must be standardized to promote uniformity in the program library.

At the top of each SCL or SAS program there should be a main program documentation section, that contains the following information in order to provide a complete reference of what's been done to the code in its lifecycle. Make sure that any additional information in the documentation section is pertinent.

1. **Program or Method Name -** name of the module.
2. **Support** - who, or what organization, 'owns' this module.
3. **Product -** which application does this code belong to?
4. **Purpose** - a brief, one-sentence description of what the module does.
5. **Usage** - how the module is called.
6. **Parameters** - what is passed into and out of the module.
7. **History** - details when the code was modified, who did it, and what was changed. The history section is the key to modifying or repairing the program properly. All entries in the history section must be as complete as possible.

   Each time the code is modified, an entry should be made in the history section. History entries should be completed for each re-cataloging of the program and should include the following information:
   - The date of the modification and project number. In the IT section of Motorola SPS, we use Rational's ClearDDTS™ defect tracking system for documenting bugs and enhancements requests. Each entry in that system has a unique tracking number. If your organization has a similar method of tracking requests, that number should go here, as well as the name of the person or persons who worked on the program.
   - A complete description of why the program was modified, what and where changes were made, and the results of the changes. Also, include any differences between the old and new versions.

- Additional information may be included in the documentation as needed, but a good rule of thumb is to limit the documentation to eight printed pages (400 lines). A general overview of the program is all that is needed to fulfill documentation requirements.

8. **Notes** - helpful notes for future maintenance. Typically you'll see warnings, or comments about future changes that should be made to this code. This is also a good place to list any references used for making the program. Notes may be entered anywhere in the documentation that is deemed necessary, but a grouping of the notes is easier to follow.
9. **Labeled Code Sections** - each labeled section of the SCL program should be listed here, in a logical order. Sections should be listed in the order in which they occur in the program. Whether it's alphabetical, or broken down as alphabetical for Frame widgets and non-widgets, or some other method, is up to you.
10. **Data Dictionary** - an alphabetical listing of all variables (including macro variables) used in a program is extremely useful. The SCL LENGTH code statements can often double for this purpose, if in-line comments are used after each LENGTH entry.

Within the body of the program, individual sections of code should be documented as well, describing the what and why of a particular code segment. Be as precise as you can -- you may be the one maintaining that code a year from now, when you've forgotten just exactly what that segment was supposed to be doing. Often, an explanation of **why** something was done is much more important than **what** was done – the **what** may be readily apparent from the code, but **why** a code section exists is not.

Without good documentation, maintenance becomes much more difficult. We have probably all heard the saying about not documenting the code being good job security, but even the best programmers cannot remember every detail about what they themselves wrote just a few months before. Poor documentation hurts everyone.

Please see Example 1: Sample Documentation Section

COMMENTING THE PROGRAM

Comments should precede the executable code and be set off in a uniform manner. In-line comments should be right-justified for readability. Each block of code should have a preceding block of comments pertaining to the workings of the code. Blocked comments ought to be set off with noticeable borders or blank lines. A continuous line of asterisks is common practice and provides unmistakable border. Single line comments should be avoided because they are easy to overlook; however, if necessary, they should also be set off with noticeable borders or blank lines to prevent code and comment confusion. Comments must be clear and concise with consideration for those who have to maintain or use the program. Having too many comments in a program is as bad as having too few. Use only enough comments to make the program understandable.

Please see Example 2: Sample Comments

CODE INDENTION AND COLORING

The purpose of code indention is to improve the readability and the logical structure of programs through a format that reflects the logic of the program:

1. Each base SAS DATA and PROC statement, and SCL entry labeled section should start in column 1.
2. Each subsequent line should be indented evenly.
3. Each DO group level should be indented, with DO statements starting on a new line for easy visibility.
4. The END statement used to terminate a DO loop should be indented at the same level as the starting DO statement.
5. Comments should not be indented to match corresponding code, and must precede the relevant code.

Coloring the SCL code can also improve readability. Our EAT team uses a gray background color, with code in black text, comments in blue, and green text for code in submit blocks. Comments in submit blocks are green as well. One of our factories IT teams used a black background, yellow text, cyan comments, and white submit blocks. Any set of contrasting colors that your programmers can agree upon will work.

NAMING CONVENTIONS

Without a standard set of variable names, code reusability means that each SCL program must often rename the variables or parameters of the calling program or macro in order to meet the naming convention of the calling program. It is much simpler to use a standard set of names for all programs written in your organization.

Further, it is useful to adopt a common naming convention for all of the individual SCL and FRAME components that make up each application. We've implemented the following naming standards at Motorola SPS IT:

- **Product Abbreviations** – All major products are identified by a unique two-character product identifier prefix. For example, the Engineering Data Analysis System[1] (EDAS) product uses "ED", while the Data Analysis Reporting Tool[2] (DART) uses "DA". Further, a few two-character codes are used for non-product-related systems, such as "C_" for common code, and "UT" for utilities.
- **Library and Catalog Names -** All libraries and catalogs associated with a given product use that products prefix. For instance, the EDAS project code library is "EDASLIB", while the EDAS beta code goes into the "ED_BETA" library, and the EDAS data management library is "ED_MANAG". Catalog entries should have the product revision ID in them: "EDAS50" for the 5.x release of EDAS.
- **Entry Names** – Use of the SCL SEARCH statement allows an application to call SAS/AF entries stored in other libraries, thus increasing the functionality of the application by taking advantage of a modular design.

  Entry names may be specified without specifying which catalog they are in. By providing a search path, the application will search a list of specified catalogs one-by-one until it finds the first entry matching the name specified. This feature aids greatly in development, since an experimental entry may be substituted for testing by merely adding a test catalog earlier in the SEARCH path for the tester. (The SAS Source Control Manager takes advantage of this ability).

  The drawback is that if you having SCL entries with the same name in the search path, regardless of the catalog names, may result in one of the products failing to work correctly.

  For this reason, a portion of the entry needs to reflect the product to which it belongs, using the same product identifier as its parent application. For instance, the FRAME entry and its associated SCL for the EDAS scatterplot module are called "ED_SCATR.FRAME" and "ED_SCATR.SCL". Non-FRAME SCL is differentiated by not having the underscore: "EDSCATR.SCL". All of the

[1] EDAS was presented at SUGI 16 by Leslie Fowler. (See reference section)

[2] DART is the successor of DevIS, presented at SUGI 21 by Larry Worley. (See reference section)

EDAS product code is stored in the EDAS50 catalog, in the EDASLIB library.

- **Macro Names and Macro Variables** –
  - Run-Time and Pre-Compiled Macros - Macros or macro variables which will exist for a user, such as run-time or pre-compiled macros, should start with an underline, followed by the two-character product identifier. I.e., "_EDMACRO". The underline makes it easier to differentiate a macro variable from a normal one. Since users may also create macros, it is important that the application's macros not interfere with their macros, and vice versa. Most users are warned not to begin their macros with an underline. (This is an important point to add to user documentation for most applications.)
  - Compile-Time Macros - Macros or macro variables which will exist only for the developer to aid in code generation - but which will not exist in the user's SAS environment when the user runs the application -- may follow any naming convention, since only the developer's environment will be affected. The convention discussed above may be followed, of course, if there is any doubt about the impact on the user or other developers.
- **Variable Names** – Here are a few recommended guidelines for variable names:
  - Avoid using variable names that duplicate SAS keyword or function names. Although SAS usually deals with these correctly, it makes reading the code very difficult.
  - Because list manipulation is so important in SCL, variables which are list identifiers stand out as such when the last part of the variable is either ...list or ...lst. These suffixes should be avoided, where possible, for non-list ids.
  - Dataset identifiers associated with a SAS dataset stand out as such when the last part of the variable is either ...id or ...dsid. These suffixes should be avoided, where possible, for non-dataset ids.
  - Temporary or holder variables can easily be designated as such by the use of temp, tmp, hold, or hld as part of their names.
  - Single-character variables such as i, j, k, etc., are fine for loop counters. Try to use more mnemonic names for important variables.

TESTING THE PROGRAM

CREATING AND TESTING THE TEST PLAN

Create an initial test plan based on the requirements and design documentation. This plan is a list of the tests to be made to verify the correctness of the results. The test plan identifies the test data to be used. Update the test plan as necessary during testing to reflect the actual testing done. All software requires testing by both the programming team and the customer before it is run in production or cataloged. Testing ensures that customer requirements are met or exceeded as far as possible, that coding and logic errors are discovered and corrected, and that each routine does what it was designed to do.

Testing should follow the test plan. Update the test plan when new or additional tests are required, or if tests described on the test plan become unnecessary or too difficult or impractical to perform. As a minimum, software should be tested as follows:

1. Test the program with both test and actual input data. Testing should include stressing the program with data both in and out of the testing parameters. This is done to ensure the program either stops or rejects the bad data.
2. Test all modules to the maximum extent possible with valid data. If possible, make sure each decision is executed at least once. Do hand calculations, if necessary, to verify that each module is functioning properly.
3. Have someone else test your program. Often, another person can discover awkward or cumbersome procedures or manage to break the program with erroneous data.
4. After modular testing, test the entire application as a whole to ensure that all of the modules work together properly. Be especially aware of the arguments and units being passed between the subroutines. Often, different arguments are required for different subroutines. Make sure the correct arguments with the correct units are passed to each subroutine.
5. Include an acceptance clause at the end of the test plan. This should be signed by the customer and the programmer at the conclusion of testing.

WALK-THROUGHS

A walk-through is a group evaluation of a product at various stages of its life cycle. Walk-throughs should be formal, properly structured, and well-documented. Proper structuring will make the walk-through more beneficial. Walk-throughs allow you to produce reliable, error free code. They can reduce the average from three to five errors per 100 lines to as few as three to five errors per 10,000 lines (Freedman 1990). They can help you correct design flaws and improve program documentation, as well as cut production time by as much as 50 percent. They also help increase the quality of system software. There are several types of walk-throughs:

- Design walk-throughs focus on the solution to the problem. This is critical in that it sets the guidelines on how a project is to be completed.
- Periodic walk-throughs are conducted whenever deemed necessary. Periodic walk-throughs will tell you where a project is.
- Final walk-throughs are necessary prior to submitting a program for a final code inspection. This walk-through will help find any discrepancies previously missed.

A minimum of three and a maximum of seven individuals should be involved in any type of walk-through. The size and scope of the project should determine the number of attendees. One individual moderates the walk-through. A second individual should record all pertinent information discussed during the walk-through, such as recommended changes to the material being presented. A third individual presents the material.

WALK-THROUGH STAGES

There are three stages in the walk-through process:

1. **Review stage -** This three to five day period prior to a walkthrough is used to acquaint each attendee with the product. Standards, checklists, material to be reviewed, and any relevant document from prior reviews or walk-throughs will be looked over during this stage.
2. **Walk-through stage** - In this stage, each detail of the product is reviewed. The presenter or moderator guides the meeting, which should last no longer than one to two hours.
3. **Follow-up stage** - This is where changes are implemented. All involved parties are informed of the changes and must agree to them.

HELPFUL HINTS

Here are a few very simple guidelines to remember while conducting a walk-through:

- The author is not on trial.
- The product is guilty until proven innocent.

- Choose walk-through participants carefully. Avoid personality conflicts if at all possible.
- Keep walk-throughs within the predetermined time limits. Schedule well in advance to ensure that everyone needed for the walk-through can attend.
- Create and follow a checklist of possible problems.

There are some people problems to watch out for in reviews and walk-throughs. Egos can play a factor in that people naturally do not like to be told they've made mistakes. Another problem area is inexperience at giving and receiving criticism. (Misdirecting comments at the creator rather than the code can turn the meeting into a defensive war.) The final problem is apathy - not trying hard to find errors. (Don't assume that others will find the same errors that you find.)

The team only has three decisions to choose from at the conclusion of a walk-through: accepting the product as is; accepting the product with revisions, trusting the creator to make the fixes; or determining that another walk-through is necessary after the errors have been corrected and the comments for improving the product have been implemented.

REVIEWS

A review is an informal check of a portion of a software program that can be conducted at any point in the development or maintenance processes. Very little documentation is needed, and structure is of no concern. Two or three individuals are sufficient for a review. Proper use of both reviews and walk-throughs will result in better software products and reduce long-range maintenance costs.

Program reviews may be made at any point in the program development cycle. The best times to review a program are after the design is developed, before any formal walk-through or code inspection, and before cataloging and production. Periodic reviews of all programs, either being developed or modified, are essential to ensure adherence to programming standards, that errors are detected (as undetected errors will haunt you later), and that documentation is correct. Make entries in the project log for each review. A properly kept log may help trace any problems that may arise later on in the project.

SOURCE CONTROL MANAGER

FEATURES

With the release of SAS version 7, SAS Institute has provided a new tool, the Source Control Manager (SCM), for managing the SAS/AF source and data files that make up your applications. Previously released as an experimental tool, the SCM is now fully integrated with SAS/AF. It provides a robust environment for developing and maintaining your SAS/AF applications, allowing you to check code in and out of the library, test changes before checking code modules back in, carry out revision control and version labeling, and easily distribute your application. The SCM environment contains a number of tools for generating reports on the development library and comparing file differences. Among these tools, the SCL Static Analyzer tool is especially notable for its ability to provide a wealth of information about the SCL in a given catalog.

The Source Control Manager (SCM) features a point-and-click interface, with pull-down or pop-up menus through which you can issue commands. The interface gives you the ability to browse the software libraries associated with the SCM, and the various catalogs and entries contained within them.

The SCM creates a control database that is associated with a given SAS library. When a developer checks code out of the SCM, the file is copied to his or her specified work area. The SCM then updates the control database by placing a lock on the file so that no one else using the SCM associated with that library can check out the code until it has been checked back in, thus preventing overwriting accidents.

By using the new CATNAME function in conjunction with the SEARCHPATH function, the SCM allows most code modules to be tested before it is checked back into the library.

Each time a file is checked back into the SCM, the previous version is archived. This provides an easy method of backing changes back out of the application if needed. The SCM administrator determines the total number of archives kept.

Further, the SCM's version labeling feature allows a 'snapshot' to be taken of the revision numbers of all the modules that make up an application. This way, if you need to rebuild an earlier release, the SCM will pull the correct revisions out of the main library and the archives to build a given release.

Once a version label has been created, you can copy that version of your application to a central distribution point, or to remote computers via SAS/CONNECT.

USING THE SCM

The SCM is located on the SOLUTIONS pull-down menu, in the DEVELOPMENT AND PROGRAMMING sub-menu. Or, you can type '`SCM`' on a SAS command line. Optionally, you can specify the location of the control database when using the command line, as well by using '`SCM SCMDATA=`*`CDBLibref`*'.

The documentation for the SCM is provided in the form of online help screens, from the SAS help menu. At the time of this writing, no documentation was available via the SAS Online Documentation CD, other than a brief mention of it.

When you start the SCM for the first time, you will need to associate a software library with it. Once you have done so, the SCM will create several datasets that make up the control database in the library. In this database are stored the list of all the files that are a part of the project, the location of the preference files for each person, who has which files locked, where the archives are to be stored, and more. Each developer can set preferences as to where his or her work library is located for each project library. The location of these preference files is determined when the SCM control database is created, and defaults to SASUSER. SAS Institute recommends that SAS/SHARE be used to access the control database library.

More than one project library can be assigned to a given control database. To do this, start the SCM with the control database you want. Then, from the TOOLS pull-down menu, select ADMINISTRATION UTILITIES. On the SOURCE DATA tab, new libraries or catalogs can be registered. Select the library or catalog, provide an archival location, and click the REGISTER button. You can choose whether or not you want all development libraries assigned to one SCM control database, or just those used for a particular project. However, it would appear to be best to include all libraries needed for a given application if you plan on using the version labeling feature.

Once the control database has been set up and configured, it's ready to be used by the developers. The programmer will start up the SCM, and tell it which control database library to use. The project library associated with that control database will then be displayed, and if any files have been checked out, those will appear with an icon of a lock in front of them. Double clicking on a file name will bring up that file in browse mode. The developer will need to tell the system where to copy the checked out files using the OPTIONS item from the pop-up menu, or TOOLS > OPTIONS from the pull-down menu. This is at a library, rather

than a catalog, level. The SCM will also ask for the developer's name, so that it can assign checked out files to the developer.

SCL STATIC ANALYZER

Another formerly experimental tool, the SCL Static Analyzer is now part of the SCM environment. It can tell you a great deal about the SCL that goes into your application, and works on a given catalog. When activated, it examines all the SCL in the catalog, and provides the following statistics: total lines, instructions, functions, attributes used, and the number of unique entries, labels and methods, variables, functions, etc. Further, it can detail the flow of your program, listing all the interrelations between the various modules that make up your application. All of the data the program collects is stored in datasets that can be viewed from within the system by clicking on the VIEW DATA button. All of the statistics and listings pop up in a compact tabular view window. Also, the Static Analyzer will provide a list of warning areas you should look at in your code…including areas of dead code and uncompiled entries, as well as areas where the Analyzer can't find the source entry being called, etc.

SCM NOTES

Several notable items:

- When checking out files, be sure not to have the catalog selected -- you **can** check out an entire catalog by accident.
- The current release of the SCM doesn't support creating copies of a version label if the version label contains multiple library references.

CONCLUSION

Programming standards provide a means of ensuring that software written at your organization is of high quality. But in order for standards to be effective, everyone must follow them.

The SCM provides an elegant solution to the problem of having program files accidentally overwritten, and provides additional functionality to the already excellent SAS development environment.

Unfortunately, due to size constraints, this article could offer only a brief overview of the software life cycle and the SCM. For more information on the life cycle and other aspects of programming standards, you might turn to the Freedman and Dunn's books listed below. For more information on the SCM, see the SAS Online Help files.

REFERENCES

Fowler, Leslie, et al (1991), *Data Analysis Applications for the Semiconductor Industry*, Proceedings of the Sixteenth Annual SAS User's Group International Conference, pg. 658-662.

Worley, Larry and Nelson, Jim (1996), *DevIS: Motorola's Near-Real-Time Device and Visualization System Using SAS/AF Software and SCL*, Proceedings of the Twenty-first Annual SAS User's Group International Conference, pg. 654-659.

Freedman, Daniel P. (1990), *Handbook of Walkthroughs, Inspections, and Technical Reviews*, New York: Dorset House Publishing.

Dunn, Robert (1982), *Quality Assurance for Computer Software*, New York: McGraw-Hill Book Company.

USAFCCC (1995), *Communications-Computer Systems Automated Data System Standards and Procedures*, Instructions 33-102, Volumes I-III, Scott AFB.

Motorola SPS IT Engineering Analysis Tools SAS Programming Standards (1998).

ACKNOWLEDGMENTS

Thanks go out to Les Bortner for a superb reviewing job and editing suggestions. Thanks also go to Chris Weyn, Leslie Fowler and Robert Smith for their reviews of the paper, as well.

CONTACT INFORMATION

(In case a reader wants to get in touch with you, please put your contact information at the end of the paper.)

Your comments and questions are valued and encouraged. Contact the author at:

C. Michael Whitney
Motorola SPS
2150 Woodward St.
Austin, TX 78744
Work Phone: (512) 996-7151
Fax: (512) 996-7148
Email: ra6952@email.sps.mot.com

EXAMPLE 1 – SAMPLE DOCUMENTATION SECTION

```
/*--------------------------------------------------------------------*/
/* Method:   EDASINIT                                                 */
/* Support:  Mike Whitney, SPS IT Solutions - Engineering Analysis    */
/* Product:  EDAS                                                     */
/* Purpose:  the method that starts EDAS50, site configuration        */
/* Usage:    call method('edasinit.scl','start','load_buffer');       */
/* Parameters:                                                        */
/*          what_to_do $20: method called by init file that starts EDAS */
/*          'load_buffer' : method called when build code for batch job */
/*          'script'      : method called when creating a script      */
/* History:                                                           */
/*    Sep 95 - M. Grover                                              */
/*           - Initial coding                                         */
/*                                                                    */
/*    Feb 97 - J. Nelson                                              */
/*           - added simple debug listing                             */
/*                                                                    */
/*    Oct 97 - E. Stokes                                              */
/*           - Added code to determine OS and pathname                */
/*             Created _edstart macro.                                */
/*                                                                    */
/* 20 Jan 98 - Mike Whitney - CIMcm0279                               */
/*           - modified OS pathname building routine, removed         */
/*             non-functioning SymPut calls.  Modified VMS section.   */
/*                                                                    */
/* 23 Jun 98 - Mike Whitney - CIMcm0280                               */
/*           - tweaked usrtr assignment to correctly reflect the VMS  */
/*             user path.                                             */
/*--------------------------------------------------------------------*/
/* Notes:                                                             */
/*--------------------------------------------------------------------*/
/* Labeled Code Sections: (listed in order of appearance)             */
/*  START - method that starts EDAS                                   */
/*  LET - Assign let statements for padas directories                 */
/*  SYMBOLN - build the SAS symbol statements                         */
/*--------------------------------------------------------------------*/

length
    ct            $3     /*  Session settings:  graphic char type   */
    dsname        $17    /*  dataset name                           */
    edasos        $8     /* edas' name for the current os           */
    fullpath      $80    /* physical path of 'padas' subdirectories */
    grcat         $17    /*  Session settings:  graphic catalog     */
    ht            $3     /*  Session settings:  graphic text height */
    let_libname   $200   /* Libname statement for padas libraries   */
    let_statement $200   /* %let macro definition statement         */
    listid        8
    ls            $8     /*  Session settings:  line size           */
    msg           $80    /*  Generic Return Message                 */
    OpSys         $35    /* substr of sysccp global macro           */
    parmtab       $17    /*  parameter table name                   */
    ps            $8     /*  Session settings:  page size           */
    rdsname       $17    /*  raw dataset name                       */
    sas_user      $80    /* physical path of the sasuser directory  */
    screenme      $17    /*  single node dataset name               */
    sys_command   $200   /*  Operating System Command               */
     ;
```

This is an older SCL entry, which wasn't documented as well as it could have been. The history was brought up to date during recent modifications

Note the issue tracking numbers.

Here the LENGTH statement does double duty as a data dictionary.

EXAMPLE 2 – SAMPLE COMMENTS

```
    control asis;

START:   /* Start EDAS */

    method
        optional=what_to_do  $20;

/*****************************************************************/
 The following submit block is here to make sure that SAM recognizes
 a new script from an older one. - 1/98 by Chris Weyn
******************************************************************/
```

An in-line comment used to briefly describe the purpose of the labeled section.

Commenting the code as to **why** something was done.

EXAMPLE 3 – CODE INDENTATION

Indented code is easy to read and follow. Only one comment has been left to show how they should be indented.

```
START:
   if symget('_eddebug')^='ON' then
      submit;
         options nosource nonotes nosource2;
      endsubmit;

/* Get the physical path of the sasuser directory    */

  select (edasos);
    when ('VMS')  sas_user= scan(pathname('padas'),1,']');
    when ('UNIX') sas_user= pathname('padas');
    when ('WIN')  sas_user= pathname('padas');
    when ('MAC')  sas_user= pathname('padas');
    otherwise;
  end;

  sas_user=lowcase(sas_user);

  if what_to_do ne ' ' then do;
    if what_to_do = 'START' then do;

      submit;
         %let _eddsnm = &dsname   ;
         %let _edrdsnm= &rdsname  ;
         %let _edsrnme= &screenme ;
         %let _edprmtb= &parmtab  ;
      endsubmit;

    end;
  end;
  else do;

    submit;
      %let _eddsnm=  ;
      %let _edrdsnm= ;
      %let _edsrnme= ;
      %let _edprmtb= ;
    endsubmit;
  end;
return;
```

Having the label in the first column, and the rest of the code indented, makes the labels more noticeable.

For the same reason, comments aren't indented, either.

Nested if-then-do loops

Submit block contents should be indented, as well.

Debugging Made Easy

Andrew Ratcliffe, Ratcliffe Technical Services Limited

If we cannot eliminate human error we are unlikely to be able to eliminate the creation of bugs in computer programs. However, the inevitability of bugs in all but the smallest of programs should not deter us from attempting to minimise the number of bugs; and those bugs that we do produce should be easy to investigate and resolve.

This paper focuses on making the process of bug investigation easier. It lists and discusses coding techniques, tools, and debugging techniques that the author has found to be successful. The paper focuses on Base SAS coding, but includes occasional mention of SAS/AF issues.

Introduction

All of my programs have bugs in them. And even if my programs appear bug free, you can be pretty sure that other aspects of the system on which they run have got bugs in them, i.e. SAS software and the operating system. So there's a fair chance that even the simplest of my programs will not run as designed on certain days of certain years under certain circumstances. It's a fact I have to live with!

Having come to terms with this fact, I have accrued and developed a number of tools and techniques for getting to the root cause of problems (potential bugs) as/when they occur. These tools and techniques are for use in a combination of situations and at a mixture of occasions in the software development life cycle. Most, however, are for use during the programming phase. If you accept the inevitability of bugs then you can see the benefit of making extra effort in the programming phase to make the bugs easier to investigate and resolve when they occur.

Expect The Unexpected

In my childhood days I was a member of the Boy Scouts. Their motto is "Be prepared" and I've kept that motto with me all these years. I try to write my programs to expect the unexpected and to handle that situation to a reasonable degree.

Select instead of if

To help me with trapping unexpected values, I generally use *select* statements instead of *if* statements when I am testing for specific values. For instance, if I have a variable that contains gender as 'M' or 'F', I might be tempted to code:

```
if gender eq 'M' then  ...do male things...
else    ...do female things...
```

But a safer option is to use the *select* statement:

```
select (gender);
  when ('M')   ...do male things...
  when ('F')   ...do female things...
end
```

The advantage of using the *select* statement is two-fold. Firstly, you are making the valid values of the gender variable very clear; secondly, the program will bomb if gender contains an invalid value. Thus, you get the earliest warning that something is wrong. If I had used the *if* statement, my program would have continued with an incorrect belief that we were dealing with a female observation.

I use the *select* statement's *otherwise* clause to trap unexpected values, but that presupposes that you can do something in the event that an unexpected value arises.

If it didn't work, do something

There's nothing worse than a program that continues after a problem has occurred. The program will inevitably destroy evidence about the original cause of the problem. It will do this by over-writing variables and data sets, etc. If you want to get to the root of a problem, you need as much good evidence as possible.

After doing something, check it worked. You can use low-level techniques such as the *select* statement I discussed above, and you can use the &syserr macro variable to check the success of a PROC or DATA step.

If it didn't work, use ABORT or STOP or ENDSAS, etc. to indicate failure, You can use ERRORABEND in a batch job.

The topic is discussed in great detail in "Taking Control and Keeping It" by Justina M Flavin et al, Proceedings of SUGI 26. See www2.sas.com/proceedings/sugi26/p074-26.pdf.

In the event of an error, you will want to prevent subsequent steps from being executed. A simple means of doing this is to use the *cancel* option on your *run* statements. *run cancel* tells SAS to just perform a syntax check on the step without executing it. I code *run &cancel* for each of my steps; I initialise &cancel to blank at the beginning of my program; I set &cancel to 'cancel' when I detect a problem and don't want subsequent steps to run. Here's an example.

```
178  options mprint;
179  %macro herbert;
180  %let cancel = ;
181
182  proc summary data=sashelp.class;
183    class sex;
184    var height wait;
185    output out=summ sum=;
186  run &cancel;
187
188  %if &syserr ne 0 %then
189    %let cancel = cancel;
190
191  data result;
192    set summ;
193    if height gt weight then put 'h>w';
194    else put 'w>h';
195  run &cancel;
196
197  %if &cancel ne %then
198    %put Program did not complete
successfully;
199  %mend herbert;
200
201  %herbert;
MPRINT(HERBERT):   proc summary
data=sashelp.class;
MPRINT(HERBERT):   class sex;
MPRINT(HERBERT):   var height wait;
ERROR: Variable WAIT not found.
MPRINT(HERBERT):   output out=summ sum=;
MPRINT(HERBERT):   run ;

NOTE: The SAS System stopped processing
this step because of errors.
WARNING: The data set WORK.SUMM may be
incomplete.  When this step was stopped
there were 0 observations and 0
        variables.
WARNING: Data set WORK.SUMM was not
replaced because this step was stopped.
NOTE: PROCEDURE SUMMARY used:
      real time           0.02 seconds
      cpu time            0.02 seconds



MPRINT(HERBERT):   data result;
MPRINT(HERBERT):   set summ;
MPRINT(HERBERT):   if height gt weight then
put 'h>w';
MPRINT(HERBERT):   else put 'w>h';
MPRINT(HERBERT):   run cancel;

NOTE: Data step not executed at user's
request.
NOTE: DATA statement used:
      real time           0.01 seconds
      cpu time            0.01 seconds


Program did not complete successfully
```

Notice how the data step did not run because of the *run cancel*. The execution of the %if means that the code must be part of a macro, which may not suit all purposes. The same technique can be used if you are calling base SAS code from SAS/AF SCL code.

Conditional test code

I include a lot of conditional test code in my work. For example, I include *put* statements that are only executed if the &debug macro variable has been set to a non-zero value beforehand. This is very useful for re-running code that you know is failing but for which you don't have enough information to figure-out the cause of the problem. I discussed this technique in greater detail in my SEUGI '99 paper entitled "Proactive Debugging". You can get a copy from www.ratcliffe.co.uk/res_papers.htm.

Make It Understandable

It may not be you

It may not be you that has to debug this program. So show some thought for your poor co-worker. Make the purpose of your program as clear as possible. Even if it is you that has to debug it later, you'll be grateful for your earlier consideration.

Neatness is a simple means of raising the level of clarity of your work. Don't underestimate the value of neatness in making your code more understandable for you and others.

No macho coding

We've all seen it! Functions embedded within functions embedded within functions, with buckets of parentheses thrown in for good measure; usage of esoteric bits of SAS functionality that were dropped from the documentation years ago but are still in the product; clever use of advanced mathematical functions to save using a larger number of basic mathematical functions.

Don't do it! Nobody will thank you when they have to maintain your program.

Write for clarity; don't try to impress. If you feel compelled to write complex code in order to achieve significantly better performance or some other goal, make sure you document your reasoning and the logic that you have applied. Use comments copiously.

Coding standards

Adopt coding standards. It is the simplest route to ensuring consistency of code and ease of maintenance for programmers. Gary E. Schlegelmilch's SUGI26 paper entitled "Structuring Base SAS for Easy Maintenance" is a good source of reference. You can get a copy at www2.sas.com/proceedings/sugi26/p220-26.pdf.

Make it explicit

I find it a tremendous help when I look at somebody else's program and I can easily trace back the source of some data. You can help to make data traceable in many ways. I'll discuss just a few here.

Scope your macro variables with %local, and by using methods in SCL. The "scope" of a variable is the range of sections within the program in which the variable is valid. In the following example, the macro variable named jaguar is globally scoped, i.e. it is accessible from any part of the SAS environment:

```
%macro rover;
  %put #2- &jaguar;
  %let jaguar = top;
  %put #3- &jaguar;
%mend rover;

%let jaguar=1;
%put #1- &jaguar;

%rover;
```

```
%put #4- &jaguar;
```

So when we run the macro, jaguar's values are as follows:

```
#1- 1
#2- 1
#3- top
#4- top
```

But if we tell SAS that jaguar is locally-scoped within the rover macro, the results are different:

```
%macro rover;
  %local jaguar;
  %put #2- &jaguar;
  %let jaguar = top;
  %put #3- &jaguar;
%mend rover;

#1- 1
#2-
#3- top
#4- 1
```

The results differ because we have two macro variables named jaguar in the second example. Within the rover macro we have a second variable who exists only for the life of the macro.

By restricting the life of information in this way, it becomes easier to trace the origin of information. Clearly it's not good practice to have variables with the same name in both the global and local macro environment, but what the examples demonstrate is that if we have a problem with jaguar inside the rover macro, the %local statement tells us that the invalid value for jaguar can only have been set within the macro. So we don't have to look too far for the cause of the problem.

In class-based SCL code, *do*/*end* blocks have their own scope too.

Another means of making it clear where information has come from is to make it clear what information is being passed between sections of code. In base, I never use _LAST_ (the last created data set), I always code the data set name.

I avoid using global macro variables and the local/global SCL list.

Parameter lists (for macros and SCL methods) should be treated as your friends. Parameter lists form excellent documentation of what is being passed into and out of a sub-routine, i.e. a macro or SCL method.

Make It Easy On Yourself

Preserve evidence

If you have to investigate a problem, you will want to follow the chain of events backwards to find the original cause. You won't be able to do this if some of the information in that chain is missing.

I never overwrite intermediate (temporary) data sets. Those data sets may not be important in terms of your final results, but they are important if you are debugging. If you call your intermediate data sets something like "temp" then it's preferable to create temp1 from your input data set, and create temp2 from temp1. If you create temp and then over-write it then you have lost the "paper trail".

This approach can result in a large number of TEMPnn data sets, so you should delete these data sets when they are no longer required, e.g. delete temp1 after temp2 has been created from it, otherwise your work library may run our of space. But if you delete these data sets you've lost your paper trail! So, I make the deletion conditional upon a macro variable that indicates whether I am in debug/development mode or in production mode.

Good syntax

Make your syntax good: remove warnings such as type-mismatches. Consider removing messages like uninitialised variables. Compare the following two DATA steps.

```
1    data _null_;
2      numeggs = 12;
3      msg = 'Old Macdonald has '
4            !! compress(numeggs)
5            !! ' eggs' ;
6      put msg= never=;
7    run;

NOTE: Numeric values have been converted to
character values at the places given by:
(Line):(Column).
      4:21
NOTE: Variable never is uninitialized.
msg=Old Macdonald has 12 eggs never=.

8
9    data _null_;
10     numeggs = 12;
11     msg = 'Old Macdonald has '
12       !! compress(putn(numeggs,'BEST.'))
13       !! ' eggs' ;
14     never = .;
15     put msg= never=;
16   run;

msg=Old Macdonald has 12 eggs never=.
```

The second DATA step shows how to remove the type mis-match and the uninitialised variable message.

The V8 SAS/AF SCL compiler has compile-time binding, thereby offering more validation at compile time

Write Things Down

A specification

Write it down in words before you code it. If you doubt yourself later (or lose focus) you'll be able to judge if you're doing what you were intended to do.

The simplest of specifications is better than none at all. You may be grateful for it if you loose track of things whilst coding. And if anybody has to maintain your code at a subsequent time, they will certainly thank you for the documentation.

Compile time

If I am distributing compiled code, I have found it useful for some documentation on compile dates/times so that I can cross-reference the distributed code that is causing trouble with the version of the source that I hold. I have a small macro that does this for me in my SAS/AF SCL code. The macro is described in my aforementioned “Pro-Active Debugging” paper.

The macro could be adapted to be used in stored macros and stored data steps in addition to its existing use with SCL.

Build a Good Foundation

Clean the environment

At the start of my programs I try to clean the environment and to validate it. This helps give me a predictable environment in which to execute.

To clean the environment I begin by clearing-out the work library. You can do this with PROC DATASETS:

```
proc datasets lib=work kill;
run;
```

In addition, I often delete any permanent data sets and/or files that I expect my program to create. However, it is not always appropriate to do this. You have to consider whether you would prefer to retain those old data sets and files in the event of the program crashing.

You can validate the environment in a number of ways. Depending upon the type of application that you’re producing, you might want to make sure you’re running on a platform that your application supports; make sure that the necessary SAS modules are licensed; make sure you have enough

spare disk space for the work library and/or permanent libraries.

You can check to see what platform you're running on by querying the &sysscp or &sysscpl macro variables. Use PROC SETINIT to see what modules are licensed.

On Windows, you can find the amount of free disk space using the getDiskFreeSpaceA DLL routine as follows:

```
libname samples 'C:\Program Files\SAS
Institute\SAS\V8\core\sample\';

filename sascbtbl catalog
'samples.pcsamp.sascbtbl.source';

data;
  success = modulen('*e'
                   ,'GetDiskFreeSpaceA'
                   ,"C:\"
                   ,numSectorsPerCluster
                   ,numBytesPerSector
                   ,numFreeClusters
                   ,numTotalClusters );
  freeSpace = numFreeClusters
              * numSectorsPerCluster
              * numBytesPerSector ;
  put success= freeSpace= comma16.;
run;
```

Incremental is best

I much prefer to develop incrementally. It allows me to discover problems one at a time. I can resolve problems as I go. I know that other developers prefer the big-bang approach where they do a great tranche of coding and then try to run and try to debug it. But to my mind, having more than one bug to resolve at the same time serves to complicate the process.

Old is best

Re-use code. Leverage your investment in earlier debugging. If I were looking for an example and I were to take this to an extreme, I would say use PROC REPORT instead of DATA _NULL_.

Sources of Information

The log is your best friend

If a problem occurs, the log could be your best source of information. In fact it may be your only source of information!

In a batch situation, make sure you are retaining the log file. And try to avoid using one log file that is over-written each time the program is run – if the program is re-run after a problem has occurred you may lose valuable information about why the problem occurred on the first occasion.

The log will be a crucial source of information in your debugging session. Write information to the log yourself, don't just rely on SAS. Paul D Sherman's SUGI26 paper "Intelligent SAS Log Manager" has a lot of good information on this topic. See www2.sas.com/proceedings/sugi26/p108-26.pdf.

Bug Mining

So maybe you've written a program, and maybe you've followed some of my recommendations. And now you find you need to investigate a bug. Frank DiIorio's SUGI26 paper entitled "The SAS Debugging Primer" is a very good source of information and guidance. You can find it at www2.sas.com/proceedings/sugi26/p054-26.pdf.

Start at the very beginning

Start at the ***top*** of the log and look for the ***first*** warning or error. Find the root cause of your problem. After one problem has occurred, subsequent sections of code are likely to fail. But don't focus on those subsequent sections, start back at the beginning and focus on the first warning/error.

Fix the first problem and you'll probably find that your subsequent problems disappear.

One step at a time

Don't try to rush what you're doing. Take your time and think carefully. Don't jump to conclusions. Try to remove one possible cause of the problem at a time. Keep removing possible causes until the code works. At that point you know that the thing you just removed is your culprit.

When you begin to drill-down into the problem, if you find it difficult to pin-down the problem, take into consideration that it might not be the fault of the SAS program. The fault may lie with any of the other components of your program, e.g. SAS itself or the operating system.

Data step debugger

Be on intimate terms with the DATA step debugger! To find out more, refer to "How to Use the Data Step Debugger", a SUGI25 paper by S. David Riba. See www2.sas.com/proceedings/sugi25/25/btu/25p052.pdf

Code generated by macros

If you have a problem with a macro then things can get extra tricky because the macro debugging tools are so limited. I find it often helps to remove the macro code from the "equation" by copying the generated Base SAS code from the log to the editor (having set *options mprint* before running the code).

However, if you have a lot of code being generated it can be tiresome to remove all those copied line numbers and other paraphernalia from the log. In this case you can use *options mfile* (*options reservedb1* in V6). Used in conjunction with the *mprint* option and an mprint filename statement, mfile gives you a "clean" copy of the code generated by your macro. See below:

```
options mfile mprint;

filename mprint
'c:\temp\mprint.txt';

%macro jimbo;
  data;
    length
  %do i=1 %to 10;
    fred&i
  %end;
  8;
  run;
%mend jimbo;

%jimbo;
```

In the log we see:

```
MPRINT(JIMBO):   data;
NOTE: The macro generated output from
MPRINT will also be written to external
file c:\temp\mprint.txt while OPTIONS
      MPRINT and MFILE are set.
MPRINT(JIMBO):   length fred1 fred2 fred3
fred4 fred5 fred6 fred7 fred8 fred9 fred10
8;
MPRINT(JIMBO):   run;
```

And in the external file we see:

```
data;
length fred1 fred2 fred3 fred4 fred5 fred6
fred7 fred8 fred9 fred10 8;
run;
```

Information sources

MSGLEVEL

Using o*ptions msglevel=i* will offer you additional information such as when indexes are being used.

```
95   options msglevel=i;
96
97   data billy (index=(demo));
98     do demo=1 to 1e4;
99       output;
100    end;
101  run;

NOTE: The data set WORK.BILLY has 10000
observations and 1 variables.
NOTE: Simple index demo has been defined.
NOTE: DATA statement used:
      real time           0.13 seconds
      cpu time            0.10 seconds
```

```
103  data subset;
104    set billy;
105    where demo lt 20;
INFO: Index demo selected for WHERE clause
optimization.
106  run;

NOTE: There were 19 observations read from
the data set WORK.BILLY.
      WHERE demo<20;
NOTE: The data set WORK.SUBSET has 19
observations and 1 variables.
```

In the example shown above, the msglevel=I option has caused two extra informational messages to be written to the log. The first tells us that the simple index was created; the second tells us that the index was used.

SASTRACE

The SASTRACE option displays detailed information about the commands that SAS/ACCESS sends to your DBMS. Use it in conjunction with the SASTRACELOC option.

DEBUGLEVEL

The “internal” SAS start-up option – debuglevel is something that offers plenty. I’ve not had time to make use of it yet! It has five potential values: testing, normal, debug, fulldebug, and demo. Invoking SAS with debuglevel=fulldebug will result in informational debugging dialog boxes.

More interesting tips like this can be found in Phil Mason’s SUGI26 paper entitled “SAS Tips I Learnt While At Oxford”. See www2.sas.com/proceedings/sugi26/p020-26.pdf.

Conclusion

Bugs are an inevitability of contemporary programming. In this paper I have tried to demonstrate some “defensive” programming techniques that I use in order to make the debugging process as painless as possible.

I encourage you to experiment with these techniques and to develop your own. In my experience it is an investment worth making.

Biography

Andrew Ratcliffe is a freelance SAS software consultant with over 15 years experience of SAS software. He specialises in object-oriented application development. Through his company (Ratcliffe Technical Services Limited) he is able to offer services including analysis and design, consultancy, and mentoring. Andrew is editor of the NOTE: free e-newsletter.

Email: andrew@ratcliffe.co.uk

Web: www.ratcliffe.co.uk

Newsletter subscription: www.ratcliffe.co.uk/note_colon

STATISTICS & DATA ANALYSIS

SECTION CHAIRS

Maribeth Johnson
Medical College of Georgia

Lita Rosario
Marquee Associates

STATISTICS & DATA ANALYSIS

Modeling Data with Nonparametric Methods Using SAS Software

Robert Cohen , SAS
Dong Xiang, SAS

Abstract: Nonparametric analysis is widely used to model data for which knowledge of the underlying model is limited. This workshop is intended for a broad audience of statisticians and data analysts who are interested in nonparametric methods. The objectives are to describe the use of new nonparametric tools in SAS software for feature identification, data compression, curve fitting, and surface modeling. In the first part of this workshop, the following methods and tools are described: fitting local regression models with the LOESS procedures, fitting thin plate smoothing spline models with the TSPLINE procedure, fitting generalized additive models with the GAM procedure, and performing wavelet analysis with new SAS/IML subroutines. The second part of the workshop illustrates the use of these tools with several examples. Topics include curve fitting and surface modeling including smoothing parameter selection, prediction confidence limits, robustness to outliers, and feature identification using wavelet tools.

Paper P-702

Individual Growth Analysis Using PROC MIXED

Maribeth Johnson, Medical College of Georgia, Augusta, GA

ABSTRACT

Individual growth models are designed for exploring longitudinal data on individuals over time. PROC MIXED allows the growth parameters for each individual to be examined as random effects in the model. Individual-level covariates can be entered into the model as fixed effects to determine their impact on the dependent variable alone and in interaction with the growth parameters. The structure of the variance-covariance matrix of the repeated measurements can also be examined and entered into the model. A model building exercise will be demonstrated using up to eight systolic blood pressure measurements of youths aged 7-22.

INTRODUCTION

Essential hypertension (EH) is a major risk factor for coronary heart disease, which remains the leading cause of death in the USA. Prior to middle age, the prevalence of EH in African Americans (AAs) is approximately twice as in European Americans (EAs), and EH-related mortality rates are 10 times higher in AAs compared to EAs.

The early natural progression of the development of high systolic blood pressure (SBP) from childhood to adulthood within the context of ethnicity and sex is not completely understood. A better understanding might contribute to a better identification of people at risk and, in turn, lead to improved prevention of hypertension.

This study evaluated the effect of sex and ethnicity upon the development of SBP over an eight year period. The ages of the children over this eight years ranged from 7 to 22 years. The data consisted of 3187 records on 524 individuals (129 EA males, 128 EA females, 122 AA males, 273 AA females). To be included in the analysis an individual had to have at least 4 out of 8 possible SBP measurements denoted as WAVEs which are measurements taken a year apart. The frequency distribution of the number of measurements is seen in Table 1. Over half of the individuals had 7 or 8 measurements and this was consistent within the ethnicity-sex subgroups.

Table 1. Measurement count distribution

Number of SBP measurements	Frequency	Percent
4	156	29.8
5	37	7.1
6	65	12.4
7	140	26.7
8	126	24.1

Plots of mean SBP across AGE for the four ethnicity-sex subgroups can be seen in the Appendix along with the AGE by WAVE frequency table. WAVE is positively correlated with AGE but they are not mutually exclusive.

THE MODEL

Individual growth models are used for exploring longitudinal data on individuals over time. Random effects or multilevel modeling allows for investigation of two levels of variability of the response variable, SBP: within and between subjects. In longitudinal data, observations taken over time are nested within subjects drawn from some population of interest giving a two-level hierarchical structure. The variation of responses within subjects over time is at the lowest level (level one) and the variation of the underlying mean responses between subjects is at level two (Singer, 1998). Measurements made on the same individual are correlated and it is this dependency that leads to the inadequacy of simple estimation procedures based on ordinary least squares. Sometimes in longitudinal data the interest lies as much in the covariance matrix estimates as in the average growth parameters. Using this technique allows us to examine both.

UNCONDITIONAL GROWTH

Unconditional growth for SBP was modeled as a function of age as a deviation from 15. Expressing age as a deviation from its mean reduces the multicollinearity between linear and quadratic regression coefficients if a quadratic model is shown to best fit the data. The intercept and slopes are fit as random effects which vary across subjects.

Linear Growth

The following SAS code is used to fit an unconditional linear growth model:

```
proc mixed noclprint covtest;
  class id;
  model sbp = age15 / solution ddfm=bw;
  random intercept age15 / sub=id type=un
gcorr;
run; quit;
```

The NOCLPRINT option on the PROC MIXED statement prevents the printing of the CLASS level information giving the number of children involved in the analysis. THE COVTEST option tells SAS that you would like hypothesis tests for the variance and covariance components.

The MODEL statement is used to indicate the fixed effects and the RANDOM statement is used to indicate the random effects. The /SOLUTION option asks SAS to print the estimates for the fixed effects. The random effects, here the intercept and the linear slope, are estimated within each subject and the variance components estimate the variation of the underlying mean responses between subjects (SUB=ID). The structure of this 2X2 variance-covariance (V-C) matrix is unstructured (UN) and the GCORR option tells SAS to print the estimated correlation matrix amongst the random effects.

Quadratic Growth

Since the plots of mean SBP across AGE appeared to be curvilinear a quadratic model was applied to the data. The code is as follows:

```
proc mixed noclprint covtest;
  class id;
  model sbp = age15 age15*age15 / solution
ddfm=bw;
  random intercept age15 age15*age15/sub=id
type=un gcorr;
run; quit;
```

The only addition is that of the quadratic age term both in the model and as a random effect. The dimension of the unstructured V-C matrix is now 3X3.

Quadratic growth: V-C grouped by ethnic-sex subgroup

The pattern across age for SBP appears quite different for the ethnicity-sex subgroups (see plots and tables in Appendix). The

AA males show more increase in SBP and much more variation in their SBP over time. The V-C matrix of the random effects for each individual quadratic growth curve was grouped by ethnicity and sex to see if these differences are significant. The similarity of the V-C matrix among subgroups was examined. Differences in the variance between members of the same subgroup might indicate sampling problems or an inherent difference between groups.

The SAS code that groups the V-C matrix of the random effects of the intercept and slopes follows:

```
proc mixed noclprint covtest;
  class id race sex;
  model sbp = age15 age15*age15 / solution
ddfm=bw;
  random intercept age15 age15*age15/sub=id
type=un gcorr group=race*sex;
run; quit;
```

RACE and SEX must be added to the CLASS statement and the GROUP=RACE*SEX option is added to the RANDOM statement.

Unconditional Growth Model Comparisons

A likelihood ratio test (LRT) for the significance of a more general model can be constructed if one model is a submodel of another by computing -2 times the difference between their residual log likelihoods (-2RLL). Then this statistic is compared to the chi-square distribution with degrees of freedom equal to the difference in the number of parameters for the two models. Models are preferred where the -2RLL is smaller. Model comparisons can also be made using Akaike's Information Criterion (AIC) or Schwarz's Bayesian Criterion (SBC). In both cases, the model that has the largest value is the preferred model. SBC penalizes models with more covariance parameters more than AIC so the two criteria may not agree when assessing model fit.

Table 2 shows the estimates for the fixed effects of these three models and Table 3 shows the comparisons between models for unconditional growth. A model with a random linear and quadratic effect provided a significantly better fit than the model with the random linear component alone based on all three model comparison methods. Adding a cubic component did not significantly improve the fit of the model (data not shown).

Table 2. Unconditional Growth Model: Fixed Effect Estimates (SE)

Model	Intercept	Age15	Age15^2
Linear	109.9 (.35)	.92 (.06)	---
Quadratic	110.3 (.37)	.85 (.06)	-.055 (.01)
Quadratic: V-C grouped	109.8 (.36)	.80 (.06)	-.058 (.01)

The quadratic model that grouped the V-C matrix on ethnic-sex subgroup provided a significantly better fit over the model where the where the V-C matrix was ungrouped (Table 3) although the estimates of the fixed effects were similar for the two models (Table 2).

Table 3. Unconditional Growth Model: Model Comparisons

Model	Param	AIC	SBC	-2RLL	Chi-square / df
Linear	3	-11014	-11022	22020	----
Quadratic	6	-10998	-11013	21982	38 / 3 *
Quadratic: V-C grouped	24	-10994	-11048	21940	42/18 *

*p<.001

The variances for both quadratic models are shown in Table 4. When broken out into groups it can be seen that the variation in SBP growth between AA males, both in the average SBP at age 15 and in the linear rate of growth, is much larger than the other subgroups. The between subject variation for EA males and females and for AA females appear similar. The covariances between the random effects are not shown but there is a significant positive correlation between the intercept and the linear slope showing that those with higher average SPB at age 15 are also increasing at a faster rate across age. The correlation between the linear and quadratic slope coefficients is 0.

There is variation in both the intercepts and the linear slopes that potentially could be explained by a level 2 (subject-level) covariate and this variation is different within ethnicity-sex subgroups. The unconditional growth model where the V-C matrix of the random effects is grouped by ethnicity-sex is used to investigate the effects of covariates on SBP changes over time.

Table 4. Unconditional Quadratic Growth Model: Variance (SE) of the Random Effects

Model	Subgroup	Intrcpt	Age15	Age15^2	Resid
V-C ungrouped		60.4 (4.5)	.47 (.12)	.010 (.004)	37.7 (1.2)
V-C grouped	EA male	47.8 (7.4)	.40 (.21)	.005 (.007)	37.6 (1.1)
	EA female	56.2 (9.0)	.37 (.23)	.001 (.008)	
	AA male	100.6 (15.2)	.79 (.29)	.020 (.011)	
	AA female	43.1 (6.3)	.30 (.22)	.020 (.010)	

ADDITION OF SUBJECT-LEVEL COVARIATES

Models were considered that investigated the fixed effect of different covariates on SBP as well as whether the variation in intercepts and slopes of the individual growth curves was related to these covariates. When investigating different covariates it is best not to group the V-C matrix due to computational and time constraints. Once a covariate is shown to have a significant effect then the V-C matrix can be grouped to show which subgroup, if any, was most effected by the addition.

Ethnicity

The code used to fit ethnicity in the model follows (AA is the reference group in the solution vector):

```
proc mixed noclprint covtest;
  class id race sex;
  model sbp = age15 age15*age15 race
race*age15 race*age15*age15 / solution
ddfm=bw;
  random intercept age15 age15*age15/sub=id
type=un gcorr group=race*sex;
  lsmeans race / pdiff;
run; quit;
```

The results are presented in Table 5. The interaction between ethnicity and the quadratic term is not included since it was not a significant effect.

Table 5. Addition of Ethnicity to the Quadratic Growth Model for SBP

	Ethnicity: AA is reference group
Fixed Effect Estimates (SE)	
Intercept	111.2 (.49)

Age15	.95 (.08)
$Age15^2$	-.055 (.012)
Covariate	-3.1 (.66)
Covariate*Age15	-.34 (.11)
Random Effect Estimates (SE)	
Intercept : EA male	56.3 (9.2)
EA female	44.9 (7.6)
AA male	88.2 (13.6)
AA female	45.6 (6.9)
Age15: EA male	.51 (.24)
EA female	.21 (.21)
AA male	.65 (.28)
AA female	.46 (.25)

There is a significant mean SBP difference due to ethnicity. The average SBP at age 15 is 108.1 for EA and is 111.2 for AA. There is also a significant interaction between ethnicity and the linear age coefficient where the rate of increase in SBP across age is .61mmHg/year for EA and .95mmHg/year for AA. The variation in the intercepts and slopes was slightly reduced for EA females (15%) and AA males (12%), slightly increased for EA males (18%), and unchanged for AA females. Adding ethnicity to the quadratic growth model helped the estimation of the curves for EA females and AA males.

Sex

The code used to fit sex in the model follows (Female is the reference group in the solution vector):

```
proc mixed noclprint covtest;
  class id race sex;
  model sbp = age15 age15*age15 sex sex*age15
sex*age15*age15 / solution ddfm=bw;
  random intercept age15 age15*age15/sub=id
type=un gcorr group=race*sex;
  lsmeans sex / pdiff;
run; quit;
```

The results are presented in Table 6. The interaction between sex and the quadratic term is not included since it was not a significant effect.

Table 6. Addition of Sex to the Quadratic Growth Model for SBP

	Sex: Female is reference group
Fixed Effects Estimates (SE)	
Intercept	107.6 (.44)
Age15	.38 (.08)
$Age15^2$	-.053 (.011)
Covariate	5.1 (.64)
Covariate*Age15	.90 (.10)
Random Effects Estimates (SE)	
Intercept : EA male	46.5 (7.3)
EA female	42.6 (6.9)
AA male	79.0 (12.0)
AA female	47.2 (7.0)
Age15: EA male	.36 (.21)
EA female	.09 (.18)
AA male	.46 (.23)
AA female	.18 (.21)

There is significance due to the main effect of sex where the average SBP at age 15 for males is 112.7 and for females is 107.6. There is also a significant difference in the rate of SBP change across age where males increase at 1.28mmHg/year and females at .38mmHg/year. The addition of sex to the model resulted in a reduction in the intercept variance for EA females (24%) and AA males (20%) while this variance was unchanged for EA males and AA females. The variation in slopes was greatly reduced between EA females (76%), AA males (42%) and AA females (40%) and relatively unchanged between EA males. The addition of sex to the quadratic growth model helped the estimation of the curves for all subgroups except EA males.

Maximum Household Education

The maximum education level attained by either the mother or the father of the child was used to try to assess socioeconomic status. This variable was coded as LOW for those whose parents had a high school or less education and HIGH otherwise.

The code used to fit education level in the model follows (Low parental education is the reference group in the solution vector):

```
proc mixed noclprint covtest;
  class id race sex ed;
  model sbp = age15 age15*age15 ed ed*age15
ed*age15*age15 / solution ddfm=bw;
  random intercept age15 age15*age15/sub=id
type=un gcorr group=race*sex;
  lsmeans ed / pdiff;
run; quit;
```

The results are presented in Table 7. The interaction between education and the quadratic term is not included since it was not a significant effect.

Table 7. Addition of Maximum Household Education to the Quadratic Growth Model for SBP

	Education: LOW is reference group
Fixed Effects Estimates (SE)	
Intercept	110.9 (.62)
Age15	1.08 (.10)
$Age15^2$	-.056 (.011)
Covariate	-1.5 (.72)
Covariate*Age15	-.40 (.12)
Random Effects Estimates (SE)	
Intercept : EA male	47.4 (7.4)
EA female	55.9 (9.0)
AA male	98.7 (15.0)
AA female	43.1 (6.4)
Age15: EA male	.39 (.21)
EA female	.39 (.22)
AA male	.69 (.28)
AA female	.31 (.22)

Those subjects whose parents had achieved a level of education beyond high school had a slightly lower average SBP at age 15 (109.4) and lower rate of SBP increase across age (.68mmHg/year) than those whose parents had a lower level of education (110.9 and 1.08mmHg/year, respectively). The variation in intercepts was unchanged for all groups by the addition of maximum household education. The linear slope variation was unchanged for EA and AA females but was slightly reduced, and therefore aided the slope estimation, for EA (17%) and AA (13%) males.

Full Model

We then looked at the effect of ethnicity, sex and education together on SBP growth. Height in centimeters centered at the mean of 150 (HTCM150) and BMI centered at its mean of 30 (BMI30) were included in the model to control for differences in growth and adiposity. It was thought that some of the ethnicity and sex differences might be due to these factors. The parameter

estimates and the random effect variances are shown in Table 8.

Table 8. Full model

	Fixed Effect Estimates (SE)	Type III test p-value
Intercept	109.3 (.81)	-
Age15	.29 (.13)	.37
$Age15^2$	.034 (.016)	.03
Ethnicity *	-3.2 (.60)	<.0001
Sex **	2.4 (.55)	<.0001
Education ***	-1.1 (.64)	.10
Ethnicity*Age15	-.41 (.10)	<.0001
Education*Age15	-.35 (.11)	.0013
HTCM150	.16 (.03)	<.0001
Sex*HTCM150	.14 (.02)	<.0001
BMI30	.30 (.04)	<.0001
	Random Effects Estimates (SE)	
Intercept: EA male	33.6 (5.5)	
EA female	33.5 (5.5)	
AA male	65.2 (9.9)	
AA female	44.6 (6.6)	
Age15: EA male	.16 (.18)	
EA female	.10 (.17)	
AA male	.55 (.25)	
AA female	.22 (.20)	

*AA is the reference group
**Female is the reference group
***Low parental education is the reference group

HTCM150 and BMI30 are both time dependent covariates that are highly correlated with age. The linear effect of age is no longer significant in this model since HTCM150 and BMI30 have entered the model as significant effects. There is still a significant quadratic age effect. EA have significantly lower average SBP at age 15 (106.1) and males have higher average SBP at age 15 (111.7) than AA or females (109.3). There was not a significant interaction between ethnicity and sex for means or linear slopes. The rate of increase across age in SBP, adjusted for differences in growth and adiposity, was significantly less for EA and for those whose parents achieved a level of education beyond high school. As all subjects grew taller their SBP increased, although males increased at a significantly faster rate than females. Increases in body mass were also related to increases in SBP for all subjects.
This model provides a 30-40% reduction in the variation of the intercepts (average SBP at age 15) for EA males and females and AA males. The variance of the intercepts remained unchanged from the unconditional growth model for AA females.

The between subject variance in the linear slopes has been reduced from 30-70% by this model. The estimate of the variance does not exceed its standard error for EA males and females and AA females. There is still significant variation in the slope estimates for AA males that is not explained by this model.

WITHIN-SUBJECT COVARIANCE STRUCTURE

One of the strengths of PROC MIXED is that it allows the comparison of different structures for the within subject, or error, covariance matrix. So far we have been concerned with the between-subject variances of the intercepts and slopes of the growth model but have not been concerned about the within-subject variances for the up to eight SBP measurements (WAVEs). In this context, the intercepts and growth rates are assumed to be constant across individuals but the model introduces a different type of complexity: the residual observations within subjects (after controlling for the linear and quadratic effects of AGE) are correlated through the within-subject error V-C matrix. This error matrix is called the **R** matrix and is modeled using the REPEATED statement.

```
proc mixed noclprint covtest;
  class id wave;
  model sbp = age15 age15*age15 / s notest;
  repeated wave /sub=id r rcorr type=cov-
structure;
 run; quit;
```

WAVE has been added as a CLASS variable to indicate the time structured nature of the data within subjects and WAVE is also used on the REPEATED statement. The SUB=ID option is the mechanism for block diagonalizing **R** since subjects are considered independent. The R option of the REPEATED statement requests that the first block of the **R** matrix be printed, the RCORR option prints the correlation matrix corresponding to **R**. If the first subject does not have all of the measurements (WAVEs) then you will need to specify a subject who does.

The TYPE= option is what determines the V-C structure. Five different *cov-structures* were considered, these are:

1) Compound symmetric (CS): This is the most specific structure, the variance within waves is constant and there is a common correlation between waves. This is the assumption if only the intercepts vary across individuals. There are two parameters estimated.
2) Heterogeneous compound symmetric (CSH): This structure assumes a common correlation between waves but allows for different variances along the diagonal. For these data there are 9 parameters estimated.
3) Toeplitz (TOEP): This structure assumes a common variance across waves but produces a banded covariance structure such that the correlations between waves separated by the same amount of time are equal. There are 8 parameters estimated.
4) Heterogeneous Toeplitz (TOEPH): This structure allows for different variance parameters and produces a banded covariance structure such that the correlations between waves separated by the same amount of time are equal. There are 15 parameters estimated.
5) Unstructured (UN): This structure produces estimates of all eight variances and 28 covariances in each subject block of **R**. Therefore, all of the correlations between waves may be different.

Table 9 shows the comparisons between these models. Since the same fixed effects were included in all models a likelihood ratio test (LRT) was used to compare models for which one is a special case of the other. CSH and TOEP were compared to CS, TOEPH was compared to TOEP, and UN was compared to TOEPH.

Table 9. Within-Subject Model Comparisons

Type	Parameters	AIC	SBC	-2RLL	Chi-square / df
CS	2	-11053	-11057	22102	---
CSH	9	-11042	-11061	22066	36 / 7 *
TOEP	8	-11022	-11039	22028	74 / 6 *
TOEPH	15	-11015	-11045	22000	28 / 7 *
TOEPH: grouped	60	-11001	-11128	21881	119/45 *
UN	36	-11015	-11092	21959	41 / 21

*p<.001

Recall that we prefer models in which the AIC and SBC are larger and the -2RLL is smaller. Allowing for heterogeneous variances for each wave along the diagonal provides a significantly better fit for both the CS and the TOEP models. Allowing for a banded covariance structure (TOEP) provides a better fit than the assumption that all of the correlations between waves are equal

(CS). The LRT between UN and TOEPH did not show a significant difference in fit so the preferred model for these data is one where the error matrix has a TOEPH structure.

Since the variances of the intercepts and the slopes were significantly different within the AA male subgroup, a model was constructed that looked into possible subgroup differences in the **R** matrix using the TOEPH structure.

```
proc mixed noclprint covtest;
  class id wave race sex;
  model sbp = age15 age15*age15 / s notest;
  repeated wave /sub=id r rcorr toeph
group=race*sex;
 run; quit;
```

The model comparison results of the TOEPH model grouped by ethnicity and sex are also shown in Table 9. The grouping provides a significantly better fit over not grouping, even though there are 45 more parameters to estimate.

The variance and correlation estimates for each subgroup are listed in Table 10. The TOEPH(X) values are the correlation between values that are X waves apart, so TOEPH(1) is the estimated correlation between waves 1 and 2, 2 and 3, 3 and 4, 4 and 5, 5 and 6, 6 and 7, and 7 and 8.

Table 10. V-C Parameter estimates (SE) from the TOEPH model: Grouped by Ethnicity and Sex

	EA Male	EA Female	AA Male	AA Female
Var (1)	58 (9)	75 (12)	120 (18)	103 (15)
Var (2)	88 (13)	89 (14)	128 (18)	83 (12)
Var (3)	79 (12)	82 (12)	130 (18)	55 (7)
Var (4)	59 (8)	68 (9)	147 (21)	72 (9)
Var (5)	80 (10)	83 (11)	161 (21)	86 (10)
Var (6)	86 (11)	70 (9)	172 (23)	84 (10)
Var (7)	94 (12)	75 (10)	170 (23)	101 (12)
Var (8)	93 (16)	92 (16)	170 (27)	139 (21)
TOEPH(1)	.59 (.04)	.66 (.04)	.72 (.03)	.49 (.04)
TOEPH(2)	.54 (.04)	.63 (.04)	.68 (.04)	.48 (.04)
TOEPH(3)	.51 (.05)	.59 (.04)	.67 (.04)	.46 (.04)
TOEPH(4)	.47 (.06)	.59 (.05)	.62 (.05)	.43 (.05)
TOEPH(5)	.42 (.07)	.46 (.07)	.64 (.05)	.38 (.06)
TOEPH(6)	.38 (.08)	.48 (.07)	.62 (.06)	.39 (.07)
TOEPH(7)	.48 (.10)	.52 (.10)	.71 (.07)	.55 (.08)

The AA Male group also shows much larger within WAVE variation in SBP than do the other groups. Their values also seem to track slightly better across WAVE but these correlations are only marginally higher when compared to the other the other groups. The major difference in subgroups appears to be within-WAVE variation in SBP.

COMBINED MODEL

When the within-subject error covariance matrix specification was combined with the model where the intercepts and slopes were considered as random effects the new model did not converge. This occurred even when neither V-C matrix was grouped by ethnicity and sex.

This new model is probably over specified. The error covariance structure within subjects describes the behavior of the errors - what remains after removing other fixed and random effects in the model. The specification of the TOEPH error matrix may no longer be needed after the intercepts and slopes are considered as random effects and additional covariates are added to the model.

CONCLUSION

PROC MIXED allows for the investigation of the changes over time within and between individuals. The intercepts and slopes of the individual growth curves can be considered as random effects and the effects of adding covariates to the model can be evaluated as changes in variances as well as the fixed effects estimates of those growth parameters and covariates.

Many different models can be considered and comparisons made in order to try to determine the best fit for the data. A limitation of this analysis technique for these data is that no one subject has measurements across the entire AGE spectrum. The change in SBP across AGE is derived from up to 8-year accumulations from many individuals.

REFERENCES

Littel, RC, Milliken, GA, Stroup, WW and Wolfinger, RD (1996) *SAS System for Mixed Models*, Cary, NC: SAS Institute, Inc.

SAS Institute Inc. (1989), *SAS/STAT User's Guide: Version 6, Fourth Edition, Volume 2,* Cary, NC: SAS Institute Inc.

SAS Institute Inc. (1992), *SAS Technical Report P-229, SAS/STAT Software: Changes and Enhancements, Release 6.07*, Cary, NC: SAS Institute Inc.

SAS Institute Inc. (1994), *SAS/STAT Software: Changes and Enhancements, Release 6.10*, Cary, NC: SAS Institute Inc.

Singer, JD (1998), *Using SAS PROC MIXED to Fit Multilevel Models, Hierarchical Models, and Individual Growth Models*, J Educational and Behavioral Statistics, 24(4): 323-355. (Also found at http://gseweb.harvard.edu/~faculty/singer/)

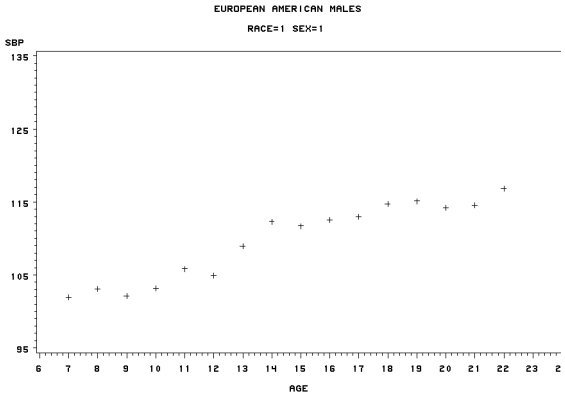
Wolfinger, RD (1992), "A tutorial on mixed models", Cary, NC: SAS Institute, Inc.

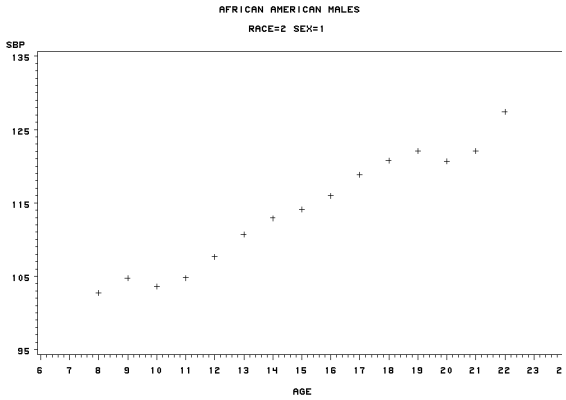
CONTACT INFORMATION

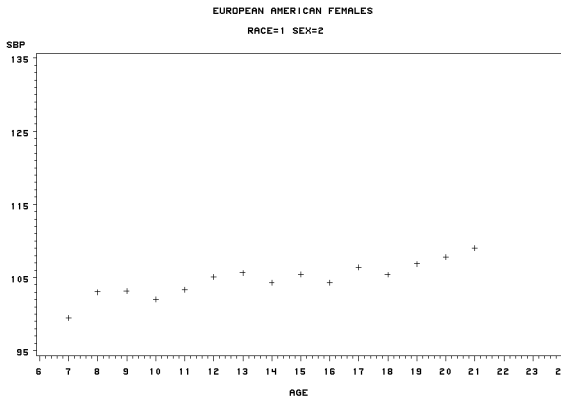
Your comments and questions are valued and encouraged. Contact the author at:

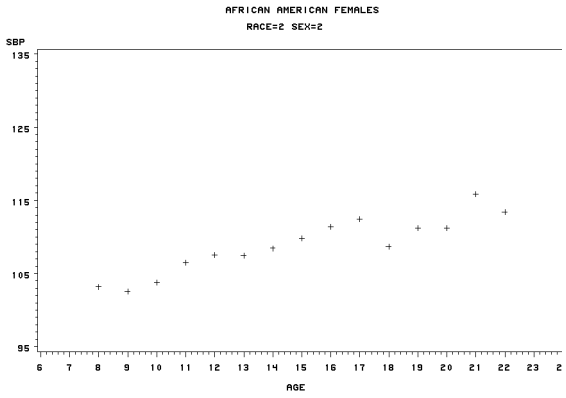
Maribeth Johnson
Office of Biostatistics AE-3035
Medical College of Georgia
Augusta, GA 30912-4900
Work Phone: (706) 721-3785
Fax: (706) 721-6294
Email: maribeth@stat.mcg.edu

APPENDIX









Subgroup Means Across Age: SBP

	EA Male		EA Female		AA Male		AA Female	
AGE	N	Mean (SD)	N	Mean (SD)	N	Mean (SD)	N	Mean (SD)
7	12	102 (7)	12	100 (7)	0	---	0	---
8	25	103 (7)	31	103 (7)	18	103 (7)	23	103 (13)
9	23	102 (6)	22	103 (7)	23	105 (8)	30	103 (8)
10	45	103 (7)	37	102 (7)	37	104 (7)	39	104 (9)
11	52	106 (9)	53	103 (7)	45	105 (8)	50	107 (8)
12	64	105 (7)	65	105 (8)	56	108 (9)	67	108 (8)
13	78	109 (9)	83	106 (7)	67	111 (10)	84	107 (8)
14	97	112 (9)	93	104 (8)	72	113 (10)	113	108 (9)
15	105	112 (9)	102	105 (9)	85	114 (10)	122	110 (9)
16	84	113 (9)	90	104 (8)	87	116(10)	111	111(9)
17	65	113 (8)	72	106 (9)	87	119 (11)	84	112 (12)
18	49	115 (10)	39	105 (8)	70	121 (13)	62	109 (9)
19	22	115 (11)	34	107(8)	46	122 (13)	42	111 (10)
20	22	114 (9)	24	108 (9)	32	121 (12)	31	111 (9)
21	16	115 (11)	12	109 (14)	19	122 (13)	21	116 (10)
22	14	117 (13)	0	---	11	127 (14)	11	113 (15)

The FREQ Procedure

Table of AGE by WAVE

AGE WAVE

Frequency	1	2	3	4	5	6	7	8	Total
7	23	1	0	0	0	0	0	0	24
8	85	12	0	0	0	0	0	0	97
9	53	36	8	1	0	0	0	0	98
10	44	75	35	3	1	0	0	0	158
11	36	78	60	21	4	1	0	0	200
12	28	37	83	65	33	4	1	1	252
13	25	26	37	109	76	35	3	1	312
14	25	25	24	67	121	76	33	4	375
15	17	23	35	49	76	116	76	22	414
16	11	15	21	34	63	80	113	35	372
17	3	11	23	38	45	64	81	43	308
18	0	4	16	18	39	40	63	40	220
19	0	0	3	13	17	41	41	29	144
20	0	0	1	4	15	23	36	30	109
21	0	0	0	2	3	12	23	28	68
22	0	0	0	1	2	3	12	18	36
Total	350	343	346	425	495	495	482	251	3187

Power and Sample Size Determination for Linear Models

John M. Castelloe, SAS Institute Inc., Cary, NC
Ralph G. O'Brien, Cleveland Clinic Foundation, Cleveland, OH

Abstract

This presentation describes the steps involved in performing sample size analyses for a variety of linear models, both univariate and multivariate. As an analyst you must gather and synthesize the information needed, but you should be able to rely on the analytical tools to accommodate the numerous ways in which you can characterize and solve problems. Examples illustrate these principles and review relevant methods. User-written, SAS® software-based programs already handle a wide variety of problems in linear models. Now, SAS Institute itself is developing software that will handle a rich array of sample size analyses, including all those discussed in this paper.

Introduction

Power and sample size computations for linear models present a level of complexity greater than that required for simple hypothesis tests. A number of steps are involved to gather the required information to perform these computations. After settling on a clear research question, the analyst must (1) define the *study design*, (2) posit a *scenario model*, a mathematical model proposing a general explanation for the nature of the data to be collected, and (3) make specific conjectures about the parameters of that model, the magnitudes of the *effects and variability*. Because developing the scenario model is typically a technically difficult and subjective process, various strategies and simplifying formulas exist to make matters more feasible, and software for sample size analysis should exploit them. Once the scenario modeling is done, you must still (4) delineate the primary *statistical methods* that will best address the research question. Finally, the (5) *aim of assessment* must be clearly expressed to ensure that the power and sample size computations accomplish the intended goal in study planning. In hypothesis testing, you typically want to compute the powers for a range of sample sizes or vice-versa. All of this work has strong parallels to ordinary data analysis. The section "Components of a Sample Size Analysis" explains these steps in more detail.

User-developed SAS-based applications, such as UnifyPow (O'Brien 1998) and the SAS/IML® program of Keyes and Muller (1992), already handle a wide variety of problems in linear models. SAS Institute is developing new software for power and sample size analyses to cover the methods discussed in this paper, along with a variety of other models discussed in Castelloe (2000).

This paper describes different strategies for power and sample size analysis for linear models in a series of examples, starting with the *t*-test and progressing through one-way analysis of variance (ANOVA), multiple regression, and multi-way ANOVA. In each example, you will first learn about the specific ingredients required for the power or sample size computation for the linear model being considered. Then the example will proceed to illustrate the implementation of a power or sample size analysis following the five-component strategy. Later sections describe unified approaches for multivariate models with fixed effects and suggest guidelines for extensions such as multiple comparisons, mixed models, and retrospective analyses.

A Review of Power Concepts

Before explaining more about the five components of a sample size analysis and proceeding through examples in linear models, a brief review of terminology used in power and sample size analysis is in order. Refer to Castelloe (2000) for a more thorough treatment of these concepts.

In statistical hypothesis testing, you typically express the belief that some effect exists in a population by specifying an alternative hypothesis H_1. You state a null hypothesis H_0 as the assertion that the effect does *not* exist and attempt to gather evidence to reject H_0 in favor of H_1. Evidence is gathered in the form of sample data, and a statistical test is used to assess H_0. If H_0 is rejected but there really is *no* effect, this is called a *Type I error*. The probability of a Type I error is usually designated "alpha" or α, and statistical tests are designed to ensure that α is suitably small (for example, less than 0.05).

If there really is an effect in the population but H_0 is *not* rejected in the statistical test, then a *Type II error* has been made. The probability of a Type II er-

ror is usually designated "beta" or β. The probability $1 - \beta$ of avoiding a Type II error, that is, correctly rejecting H_0 and achieving statistical significance, is called the *power*. An important goal in study planning is to ensure an acceptably high level of power. Sample size plays a prominent role in power computations because the focus is often on determining a sufficient sample size to achieve a certain power, or assessing the power for a range of different sample sizes. Because of this, terms like *power analysis*, *sample size analysis*, and *power computations* are often used interchangeably to refer to the investigation of relationships among power, sample size, and other factors involved in study planning.

Components of a Sample Size Analysis

Even when the research questions and study design seem straightforward, the ensuing sample size analysis can seem technically daunting. It is often helpful to break the process down into five components:

Study Design: What is the structure of the planned design? This must be clearly and completely specified. What groups and treatments ("cells" and "factors" of the design) are going to be assessed, and what will be the relative sizes of those cells? How is each case going to be studied, i.e., what are the primary outcome measures ("dependent variables"), and when will they be measured? Will covariates be measured and included in the statistical model?

Scenario Model: What are your beliefs about patterns in the data? Imagine that you had unlimited time and resources to execute the study design, so that you could gather an "infinite data set." Characterize that infinite data set as best you can using a mathematical model, realizing that it will be a simplification of reality. Alternatively, you may decide to construct an "exemplary" data set that mimics the infinite data set. However you do this, your scenario model should capture the key features of the study design and the main relationships among the primary outcome variables and study factors.

Effects & Variability: What exactly are the "signals and noises" in the patterns you suspect? Set specific values for the parameters of your scenario model, keeping at most one unspecified. It is often enlightening to consider a variety of realistic possibilities for the key values by performing a sensitivity analysis. Alternatively, construct two or three exemplary data sets that capture the competing views on what the infinite data set might look like. For linear models and their extensions an important component is the "residual" term that captures unexplained variation. The standard deviation (SD) of this term plays a critical role in sample size analysis. What is this value? A sensitivity analysis is usually called for in positing SD.

Statistical Method: How will you cast your model in statistical terms and conduct the eventual data analysis? Define the statistical models and procedures that will be used to embody the study design and estimate/test the effects central to the research question. What tests will be done? What significance levels will be used? Will one- or two-tailed tests be used?

Aim of Assessment: Finally, what needs to be determined in the sample size analysis? Most often you want to examine the statistical powers obtained across the various scenarios for the effects, the statistical procedures (tests) to be used, and the feasible total sample sizes. Some analysts find sample size values that provide given levels of power, say 80%, 90%, or 95%. Other analysts compute the value for some key effect parameter (e.g., a given treatment mean) that will provide a given level of power at a given sample size. You might even want to find the α-level that will provide a given power at a given sample size for a given effect scenario.

The following examples illustrate how these components can provide a guiding structure to facilitate more rigorous planning of studies involving linear models.

Preview of Examples

The examples are organized by different types of linear models. The main distinction is in the type of predictors (or independent variables) in the model. All examples are univariate, involving only one continuous response variable. The first two examples are simple cases involving only categorical predictors, the *t*-test and one-way ANOVA. Multiple regression involves predictors treated as continuous variables, although some may be dummy (0/1) variables representing categories. The multi-way ANOVA with covariates contains categorical predictors of interest *and* additional continuous predictors used as covariates to reduce excess variability. Finally, power computations for the multivariate general linear model (GLM) are discussed, without a detailed example, and guidelines are given for some common linear models extensions not covered by examples.

The format of each example is as follows. First the type of linear model is briefly explained, and the problem situation of the study planner is revealed. The five-component strategy explained in the "Components of a Sample Size Analysis" section is applied to solve the problem, and then the power computation details are explained, including relevant equations and references. The equations are used to solve the problem at hand, but they are sufficiently general

for the variety of different possible goals (e.g., solving for power, sample size, etc.).

Ordinary Two-Sample *t*-test

The two-sample (pooled) *t*-test is equivalent to an ANOVA with two groups and thus is a special case of a linear model. Although power computations for *t*-tests are widely understood and implemented, a characterization following the five-component strategy provides a useful framework for more complicated examples. In addition, the last part of the example illustrates the consideration of unbalanced designs and costs.

Suppose an industrial chemist is researching whether her firm should switch to a new grade of ammonium chloride when producing an organic compound, SHHS-01. A more expensive, finely ground grade is touted to give a higher yield than the standard coarse grade, but this needs to be tested. A 5% increase in yield would offset the extra cost. The chemist's goal is to determine an appropriate sample size to have adequate power in a comparison of the two grades using a *t*-test.

Study Design: Under laboratory conditions designed to mimic production conditions, equal numbers of mini-batches of SHHS-01 will be made using either the coarse or fine grade of ammonium chloride. The primary outcome measure will be the yield, measured simply as weight in grams.

Scenario Model: The conjectured "infinite data set" has two groups of yields, each containing independent and normally distributed data. The mean for the fine group exceeds the mean for the coarse group. There is no reason to suspect that the variability differs between groups.

Effects & Variability: Based on numerous previous laboratory studies with the coarse ammonium chloride, the chemist knows that this yield averages about 160g/batch. Because this is an organic process, there is significant variability as well, with a standard deviation of 20g. For the fine ammonium chloride, biological modeling predicts that the yield could be at least 10% greater, at 176g. The 20g standard deviation should apply. This scenario is illustrated in Figure 1.

Statistical Method: The statistical question is whether the fine grade ammonium chloride will produce at least 5% (8g) more SHHS-01 than the coarse grade (to offset the extra cost). This situation conforms to an ordinary, one-tailed *t*-test, with hypotheses $H_0 : \mu_2 - \mu_1 = \mu_{\text{diff}} < 8$ and $H_1 : \mu_2 - \mu_1 = \mu_{\text{diff}} \geq 8$. Making the mini-batches in the lab is inexpensive, but making a Type I error could lead to establishing a production process that is substantially more costly. Hence, alpha will be set at 0.001 or, at most, 0.005.

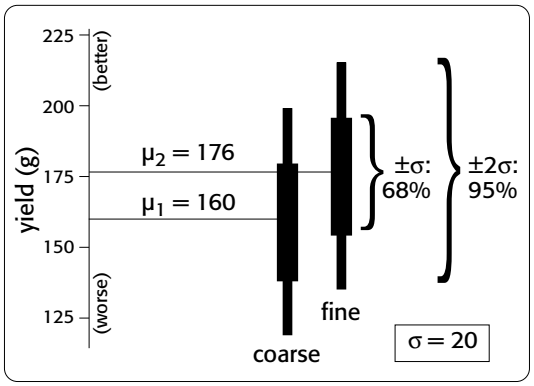


Figure 1. Conjectured Scenario for Coarse and Fine Grades

Aim of Assessment: If the fine grade is really 10% more effective, as believed, then it will save the company a lot of money. Thus, failing to discover this would be so costly that the chemist decides to produce enough mini-batches to achieve a statistical power of 99%. Remember, making and weighing the mini-batches is relatively inexpensive. So the chemist wants to determine the required sample size to provide 99% power.

This required sample size is determined by solving the following equation for n:

$$\text{power} = P\left(t(2n-2,\delta) \geq t_{1-\alpha;2n-2}\right)$$

where $t(u,\delta)$ is distributed as noncentral t with u d.f. and noncentrality $\delta = \sqrt{n/2}\,(\mu_{\text{diff}} - \mu_0)/\sigma$, and $t_{p;u}$ is the p^{th} quantile of the central t distribution with u d.f.

With 99% power, the required sample size per group is n = 303 for α = 0.005 and n = 370 for α = 0.001. Hence, using N = 740 total will achieve outstanding control of Type I and Type II errors. N = 606 is also feasible.

Unbalanced Designs and Cost Suppose that in the lab it costs 75% more to produce a mini-batch using the experimental fine grade of ammonium chloride. The chemist wonders if an unbalanced design with fewer fine mini-batches than coarse ones would produce as much power but at less cost. Consider a 3:2 sampling ratio, i.e., cell weights of w_1 = 3/5 and w_2 = 2/5.

Power computations can be performed in terms of

group weights and total sample size:

$$\text{power} = P\left(t(N-2,\delta) \geq t_{1-\alpha;N-2}\right)$$

where $\delta = \sqrt{N w_1 w_2}(\mu_{\text{diff}} - \mu_0)/\sigma$.

To achieve 99% power using α = 0.001 requires 462 + 308 = 770 cases versus 370 + 370 = 740 cases for the balanced design. The unbalanced study design is 4.1% larger, but it would cost about 1.6% less to run. While the ordinary two-group *t*-test has optimum *statistical* efficiency with a balanced design, it can have sub-optimum *budgetary* efficiency if the cost per sampling unit differs between the groups.

Note that two-tailed versions of the above formulas are available, using the noncentral $F = t^2$ distribution (with noncentrality $\lambda = \delta^2$).

One-Way ANOVA with One-d.f. Contrast

This example extends the previous one by increasing the number of groups from two to three. An appropriate sample size will be determined for a comparison (represented this time as a linear contrast) between one group and the average of the other two.

Suppose the chemist introduced in the previous section implements the comparison between the fine and coarse grades of ammonium chloride and concludes that the fine grade is advantageous, giving a yield of about 176g as predicted. Plans to purchase the fine grade chemical from Producer A are interrupted when Producer B offers a package deal of "special-fine" bundled with "super-fine" in a 1:1 ratio, for about the same cost. Engineers at Producer B claim that the special-fine grade yields 172g of SHHS-01, while the super-fine yields 190g, with the same variability as the fine grade from Producer A (SD = 20g). The chemist is asked to compare the yield using Producer A's fine grade to the average yield of special fine and super fine (used separately) supplied by Producer B, with enough mini-batches to achieve 90% power if the engineers are correct.

Study Design: The chemist will conduct experiments using the three different grades of ammonium chloride, with the two varieties from Producer B used equally often and each twice as often as the Producer A variety. In other words, the weights are 2 special-fine : 2 super-fine : 1 fine (w_1 = 0.4, w_2 = 0.4, w_3 = 0.2). The dependent variable is the yield in grams.

Scenario Model: The chemist will assume that the chemicals from the two producers will produce mean yields as previously surmised for Producer A and claimed by engineers for Producer B.

Effects & Variability: The mean yields are conjectured to be μ_1=172g (special-fine), μ_2=190g (super-fine), and μ_3=176g (fine). The standard deviation is assumed to be about 20g for each grade.

Statistical Method: A 1-way ANOVA will be conducted to test a contrast of fine with average over special-fine and super-fine, using the usual F statistic with $\alpha = 0.05$. The contrast can be written as a CONTRAST statement for PROC GLM, for example, as

```
contrast grade -1 -1 2
```

Aim of Assessment: The chemist wishes to calculate the required sample size to achieve 90% power.

The required total sample size N can be calculated from the following equation:

$$\text{power} = P\left(F(1, N-G, \lambda) \geq F_{1-\alpha;1;N-G}\right)$$

where

$$\lambda = N \frac{\left(\sum_{i=1}^{G} c_i \mu_i - c_0\right)^2}{\sigma^2 \sum_{i=1}^{G} \frac{c_i^2}{w_i}}$$

and $\{c_i\}$ are the contrast coefficients. The required sample size is found to be 735 (rounded up from 732 to avoid fractional group sizes).

Multiple Linear Regression

Instead of categorical predictors as in the *t*-test and 1-way ANOVA, multiple regression involves continuous and dummy independent variables. Although as a special case multiple regression could be used with dummy variables to conduct an ANOVA, the intention here is to demonstrate the more typical usage focusing on tests of individual predictors controlling for other predictors. Such a test is planned in this example, and the goal is to compute the power.

One of the important considerations in multiple regression and correlation analysis is whether to treat the predictors as fixed or random. There are also many alternative ways to characterize the effects, using various forms of correlations and regression coefficients. The example uses fixed predictors and involves an effect specification in terms of partial correlation. Following the example is a discussion of the ramifications of the distinction between fixed and random predictors and a collection of equations showing the alternative ways to specify effects.

A team of preventive cardiologists is investigating whether elevated serum homocysteine levels are

linked to atherosclerosis (plaque build-up in coronary arteries). The analysis will use ordinary least squares regression to assess the relationship between total homocysteine level (tHcy) and a plaque burden index (PBI), adjusting for six other variables: age, gender, and plasma levels of folate, vitamins B_6 and B_{12}, and a serum cholesterol index. The group wonders whether 100 subjects will provide adequate statistical power.

Using the five components, the power analysis breaks down as follows:

Study Design: This is a correlational study at a single time. Subjects will be screened so that about half will have had a heart problem. All eight variables will be measured during one visit.

Scenario Model: Most clinicians are familiar with simple correlations between two variables, so the collaborating statistician decides to pose the statistical problem in terms of estimating and testing the partial correlation between X_1 = tHcy and Y = PBI, controlling for the six other predictor variables ($R_{YX_1|X_{-1}}$). This greatly simplifies matters, especially the elicitation of the conjectured effect.

The statistician uses partial regression plots like that shown in Figure 2 to teach the team that the partial correlation between PBI and tHcy is the correlation of two sets of residuals obtained from ordinary regression models, one from regressing PBI on the six covariates and the other from regressing tHcy on the same covariates. Thus each subject has "expected" tHcy and PBI values based on the six covariates. The cardiologists believe that subjects who are relatively higher than expected on tHcy will also be relatively higher than expected on PBI. The partial correlation quantifies that adjusted association just like a standard simple correlation does with the unadjusted linear association between two variables.

Effects & Variability: Based on previously published studies of various coronary risk factors and after viewing a set of scatterplots showing various correlations, the team surmises that the true partial correlation is likely to be at least 0.35.

Statistical Method: Regress PBI on tHcy and the six other predictors, plus the intercept. Use an ordinary *F* test to assess whether tHcy is a significant predictor in this model with seven predictors. The test presumes that the residuals come from a normal distribution.

Aim of Assessment: Compute the statistical powers associated with N = 80 and 100, using α = 0.05 and 0.01.

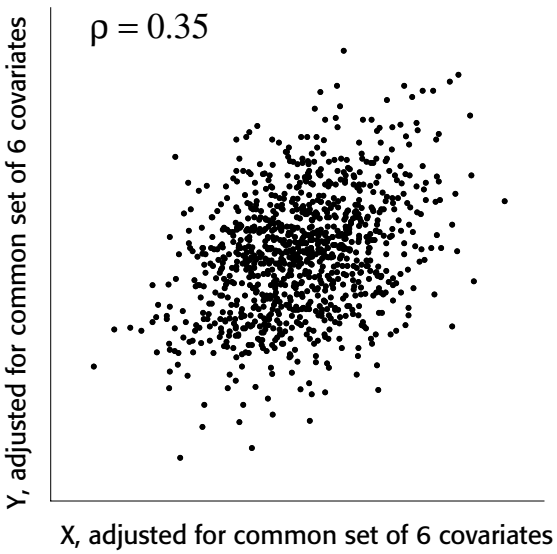


Figure 2. Partial Correlation Plot

The exact power can be computed from the equation

$$\text{power} = P\left(F(p_1, N-p-1, \lambda) \geq F_{1-\alpha;p_1;N-p-1}\right) \quad (1)$$

where p is the total number of predictors in the model (including the predictor of interest, but not the intercept), p_1 is the number of predictors being tested simultaneously (here, p_1 = 1), and

$$\lambda = N\frac{R^2_{YX_1|X_{-1}}}{1-R^2_{YX_1|X_{-1}}} \quad (2)$$

The calculated powers range from 75% (N = 80, α = 0.01) to 96% (N = 100, α = 0.05). The latter result is almost balanced with respect to Type I and Type II error rates. The study seems well designed at N = 100.

Fixed vs. Random Predictors The computations in the example assume a *conditional* model, as typically used in multiple linear regression. The predictors (represented collectively as X) are assumed to be fixed, and the responses Y are assumed to be independently normally distributed conditional on X. The usual test statistic considered is the Type III F test where the null hypothesis states that all coefficients of the p_1 predictors of interest are zero.

A related approach is the *unconditional* model, typically used in multiple correlation analysis, in which predictors are assumed to be random. The variables in Y and X are taken to have a joint multivariate normal distribution. Power computations differ for the conditional and unconditional models. Gatsonis and Sampson (1989) outline an exact power computa-

tion method for the unconditional model due to Lee (1972).

It is important to note, however, that the usual test statistics for conditional and unconditional models are equivalent, having exactly the same null distribution. "The conceptual difference between them is primarily one of interpretation and generalizability of the conclusions" (Gatsonis and Sampson 1989, p. 516). Thus the strategies for describing effects in each of the two approaches can be used interchangeably in sample size analysis. For example, the cardiologists conjectured effects in terms of partial correlation but planned to use multiple regression.

Alternative Effect Specifications The remainder of this section describes the various ways you can describe effects using different types of correlations and regression coefficients. You can use the same parameterizations for either conditional or unconditional models in power computations. The well-known method for the conditional framework is outlined explicitly here, and you can refer to Gatsonis and Sampson (1989) for analogous computations in the unconditional framework.

Consider the general situation in which you are interested in testing that the coefficients of $p_1 \geq 1$ predictors in a set X_j are zero, controlling for all of the other predictors X_{-j} (comprised of $p - p_1 \geq 0$ variables). For the conditional model, the power can be computed using equation (1), where the noncentrality λ is defined differently for various alternative specifications of the effects. You can choose whichever one is most convenient for expressing the conjectured effects in your situation.

One such specification involves the multiple partial correlation $R_{YX_j|X_{-j}}$:

$$\lambda = N\frac{R^2_{YX_j|X_{-j}}}{1 - R^2_{YX_j|X_{-j}}}$$

You can also express the effects in terms of the multiple correlations in full ($R_{Y|(X_j,X_{-j})}$) and reduced ($R_{Y|X_{-j}}$) nested models:

$$\lambda = N\frac{R^2_{Y|(X_j,X_{-j})} - R^2_{Y|X_{-j}}}{1 - R^2_{Y|(X_j,X_{-j})}} \quad (3)$$

The numerator of (3) is equivalent to the squared multiple semipartial correlation $R^2_{Y|(X_j|X_{-j})}$. Thus

$$\lambda = N\frac{R^2_{Y|(X_j|X_{-j})}}{1 - R^2_{Y|(X_j,X_{-j})}} \quad (4)$$

You may find it easier to work in terms of standard (zero-order) correlations, even though there are more parameter values to specify. A form of λ involving the correlations between Y and variables in $X = \{X_j, X_{-j}\}$ (labeled as vectors ρ_{XY} and $\rho_{X_{-j}Y}$), and between pairs of variables in X (labeled as correlation matrices S_{XX} and $S_{X_{-j}X_{-j}}$), is given by the following:

$$\lambda = N\frac{\rho'_{XY}S^{-1}_{XX}\rho_{XY} - \rho'_{X_{-j}Y}S^{-1}_{X_{-j}X_{-j}}\rho_{X_{-j}Y}}{1 - \rho'_{XY}S^{-1}_{XX}\rho_{XY}} \quad (5)$$

The remaining specifications apply only to cases in which X_j consists of a single predictor.

You can express λ in terms of the standardized regression coefficient ($\tilde{\beta}_j$) of X_j; the tolerance of X_j, computed as $1 - R^2_{X_j|X_{-j}}$ in a regression of X_j on the other predictors; and the multiple correlation $R_{Y|(X_j,X_{-j})}$ for the full model:

$$\lambda = N\frac{\tilde{\beta}^2_j(1 - R^2_{X_j|X_{-j}})}{1 - R^2_{Y|(X_j,X_{-j})}} \quad (6)$$

Or, you can posit the unstandardized coefficient β_j along with the tolerance of X_j, SD of X_j (σ_{X_j}), and SD of residual (σ):

$$\lambda = N\frac{\beta^2_j(1 - R^2_{X_j|X_{-j}})\sigma^2_{X_j}}{\sigma^2}$$

If an exchangeable correlation structure is deemed reasonable, equation (5) can be simplified to include only the common correlation between Y and each predictor (ρ_{XY}) and the common pairwise correlations between predictors (ρ_{XX}):

$$\lambda = N\frac{\rho^2_{XY}(1 - \rho_{XX})}{[1 + (p-1)\rho_{XX} - p\rho^2_{XY}][1 + (p-2)\rho_{XX}]}$$

A useful compromise between the exchangeable correlation structure and the necessity of specifying all correlations is a *relaxed* exchangeable correlation structure (Maxwell 2000), which allows different correlations ρ_{X_jY} between Y and X_j, and $\rho_{X_jX_{-j}}$ between X_j and the other predictors, in addition to common correlations $\rho_{X_{-j}Y}$ between Y and components of X_{-j}, and $\rho_{X_{-j}X_{-j}}$ between elements of X_{-j}:

$$\lambda = N\frac{R^2_{Y|(X_j,X_{-j})} - R^2_{Y|X_{-j}}}{1 - R^2_{Y|(X_j,X_{-j})}}$$

where

$$R^2_{Y|(X_j,X_{-j})} = \Big\{\rho^2_{X_jY}[1 + (p-2)\rho_{X_{-j}X_{-j}}] +$$

$$(p-1)\rho^2_{X_{-j}Y} - 2(p-1)\rho_{X_jY}\rho_{X_{-j}Y}\rho_{X_jX_{-j}}\Big\} \cdot$$
$$\Big\{1 - \rho_{X_{-j}X_{-j}} + (p-1)\rho_{X_{-j}X_{-j}} - \rho^2_{X_jX_{-j}}\Big\}^{-1}$$

and

$$R^2_{Y|X_{-j}} = \frac{(p-1)\rho^2_{X_{-j}Y}}{1 + (p-2)\rho_{X_{-j}X_{-j}}}$$

If you want to test contrasts of the regression coefficients, you can use the more general formulation discussed in the section "The Univariate GLM with Fixed Effects."

Multi-Way ANOVA with Fixed Effects and Covariates

This next example features an ANOVA model that is an extension of the kind of model considered in the section "One-Way ANOVA with One-d.f. Contrast" in the sense of having two factors (instead of one) and additionally a continuous covariate. The planned tests are also more complicated, involving several contrasts. The goal of the scientists in the example is to assess whether their largest possible sample size will provide adequate (possibly excessive) power for these tests.

The discussion in this section follows the five-component layout as used in the previous examples. Details regarding the mathematical power computations, and other alternative ways of describing the components, are covered in the following section, "The Univariate GLM with Fixed Effects."

Suppose a team of animal scientists hypothesizes that dietary supplements of the trace element sugianimum (fictitious) increase the growth rate in female newborn rabbits. Standard rabbit chow contains 5 ppm sugianimum. The scientists want to study four other supplemental formulations, +10, +20, +40, and +80 ppm (with the standard chow designated as +0). This will allow them to conduct an ANOVA to test models with thresholds, ceiling effects, and/or dose-response effects. They have sufficient facilities and funding to study at most 240 rabbits but would be pleased if fewer would seem to suffice.

Rabbits are eight to nine weeks old when they arrive from the commercial breeder, and their body weight is 1.5 $\pm$ 0.25 kg (mean $\pm$ SD). After eating only standard chow for the next 24 weeks, female rabbits of this breed have a body weight of about 4.2 kg $\pm$ 0.56 kg.

Study Design: The primary outcome measure will be each rabbit's body weight after 24 weeks on the study.

Sugianimum level is represented by a factor called Sugianimum Supplementation Level or SugiSupp, with five levels (+0, +10, +20, +40, and +80 ppm). The scientists will include rabbit feed from all five of the major U.S. manufacturers (Gamma, Epsilon, Zeta, Eta, and Theta) to enable greater generalizability of the results. Call this factor Company. Thus, if all manufacturers supplied all formulations, the design would be a 5 $\times$ 5 factorial, Company $\times$ SugiSupp.

Suppose each company produces only two formulations besides the standard one (+0 ppm), thus making a complete factorial design impossible. The scientists will use a randomized design with cell weights as displayed in Table 1. Thus, they are planning a 2:1:1 ratio for the standard and two supplemental formulations for each company.

		SugiSupp				
Company		+0	+10	+20	+40	+80
<1>	Gamma	2	1	1	0	0
<2>	Epsilon	2	1	0	1	0
<3>	Zeta	2	0	1	0	1
<4>	Eta	2	0	0	1	1
<5>	Theta	2	1	0	0	1

Table 1. Cell Weights for Design

Rabbits that are larger at baseline tend to gain more body weight during the study period. Because of this correlation, the rabbits' initial body weight RabbitWgt00 could serve as a useful covariate by accounting for extra variation in body weight at 24 weeks. So the scientists plan to include the measurement of RabbitWgt00 in the study protocol.

Scenario Model: The scientists envision two different scenarios for the means of body weights at 24 weeks across SugiSupp and Company. Both scenarios assert a monotonically increasing dose-response relationship until 40 ppm but a ceiling effect after that, and the average weight gains differ by company. "Scenario 1" conjectures that the pattern of sugianimum effects is the same across companies, i.e., the Company $\times$ SugiSupp interaction is null. "Scenario 2" involves essentially the same main effects but reflects the suspicion that Gamma's +10 formulation is less effective than its own +0, and Epsilon's +10 formulation is unusually effective compared to its own +0.

Effects & Variability: For scenario 1, the scientists conjecture means for the 5 $\times$ 5 factorial as shown in Table 2. The means are displayed graphically in Figure 3.

One can confirm that these means conform perfectly

to the main-effects-only linear model,

$$\mu_{\{\text{Company,SugiSupp}\}} = 4.2 + A_{\{\text{Company}\}} + B_{\{\text{SugiSupp}\}}$$

where $A_{\{1\}}$ = 0.0, $A_{\{2\}}$ = -0.2, $A_{\{3\}}$ = 0.2, $A_{\{4\}}$ = -0.1, $A_{\{5\}}$ = 0.1; and $B_{\{+0\}}$ = 0, $B_{\{+10\}}$ = 0.1, $B_{\{+20\}}$ = 0.4, $B_{\{+40\}}$ = 0.5, $B_{\{+80\}}$ = 0.5.

		SugiSupp				
Company		+0	+10	+20	+40	+80
<1>	Gamma	4.2	4.3	4.6	4.7	4.7
<2>	Epsilon	4.0	4.1	4.4	4.5	4.5
<3>	Zeta	4.4	4.5	4.8	4.9	4.9
<4>	Eta	4.1	4.2	4.5	4.6	4.6
<5>	Theta	4.3	4.4	4.7	4.8	4.8

Table 2. Conjectured Means for Scenario 1

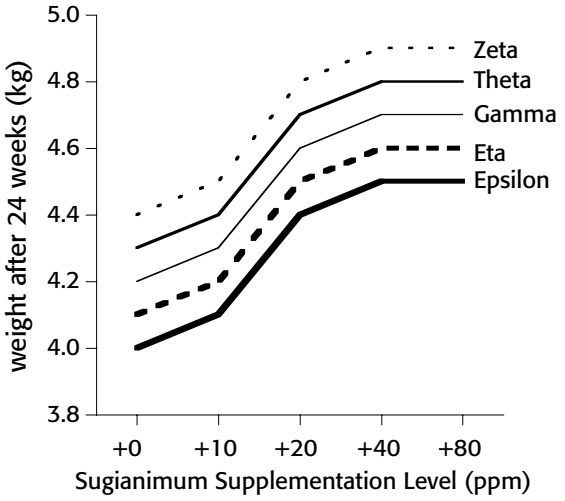


Figure 3. Conjectured Means for Scenario 1

In scenario 2, the scientists consider the same main effects but also a small interaction involving only the 2 $\times$ 2 cells in the top left corner of the table:

$$\mu_{\{1,+0\}} = 4.3,\ \mu_{\{1,+10\}} = 4.2,$$
$$\mu_{\{2,+0\}} = 3.9,\ \mu_{\{2,+10\}} = 4.2$$

All of the specifications for this problem can be incorporated into a single SAS data set, as follows.

```
proc plan ordered;
  factors Company=5 SugiSupp=5 / noprint;
  output out=Design Company  cvals=('Gamma'
    'Epsilon' 'Zeta' 'Eta' 'Theta')
    SugiSupp nvals=(0 10 20 40 80);
run;

data CellWeights;
   input CellWgt @@;
datalines;
2 1 1 0 0
2 1 0 1 0
2 0 1 0 1
2 0 0 1 1
2 1 0 0 1
;

data CellMeans; keep Scenario1 Scenario2;
  array A{5} (0.0 -0.2 0.2 -0.1 0.1);
  array B{5} (0.0  0.1 0.4  0.5 0.5);
  do i = 1 to 5; do j = 1 to 5;
    Scenario1 = 4.2 + A{i} + B{j};
    Scenario2 = Scenario1;
    if      ((i=1)&(j=1))
      then Scenario2 = Scenario2 + 0.1;
    else if ((i=1)&(j=2))
      then Scenario2 = Scenario2 - 0.1;
    else if ((i=2)&(j=1))
      then Scenario2 = Scenario2 - 0.1;
    else if ((i=2)&(j=2))
      then Scenario2 = Scenario2 + 0.1;
    output;
  end; end;
run;
data rabbits5x5;
  merge Design CellWeights CellMeans;
run;
```

The scientists consider 0.56 to be a reasonable guess of the error SD, but they would also like to assess the power assuming this SD is as high as 0.73. They believe there is a correlation of about ρ=0.45 between baseline body weight and body weight after 24 weeks. The design is randomized, and so there is no underlying relationship between RabbitWgt00 and the design factors, Company and SugiSupp. The team also presumes that the Company and SugiSupp effects are not moderated by RabbitWgt00, i.e., there is no RabbitWgt00 $\times$ Company or RabbitWgt00 $\times$ SugiSupp interaction. Accordingly, the only effect of adding RabbitWgt00 to the linear model will be to reduce the error standard deviation to $(1-\rho^2)^{\frac{1}{2}}$ of its original value. Thus, ρ = 0.45 reduces the SD values by $100\left[1-(1-\rho^2)^{\frac{1}{2}}\right]$% = 10.7%. So the conjectured values for error SD (originally 0.56 and 0.73) become 0.5 and 0.65.

Statistical Method: The team assumes normality for the distribution of rabbits' body weights (conditional on the explanatory variables). Variables such as body weight tend to be positively skewed and may need to be transformed prior to analysis. In addition, without such a transform, the SDs for the two groups may not be equal, because there tends to be a positive relationship between groups' means and their SDs. Both problems are often greatly mitigated by using a log transform, i.e., by assuming that the original data is lognormal in distributional form. But for the purposes of this example, assume that the normality assumption for the body weights is reasonable.

This study could be analyzed in numerous ways. The

strategy chosen should be incorporated into a sample size analysis that conforms to the data analysis plan. The scientists decide to compare each of the four supplemental formulations with the control in an ANOVA, using a Bonferroni correction for multiple testing with overall α = 0.05, or α = 0.05/4 = 0.0125 per test. Alternatively, they could use Dunnett's test. They will also test for a dose-response relationship, assuming that the essential component of that relationship is captured using just the linear trend across the five levels of SugiSupp. Formally, this assumes that +0, +10, +20, +40, and +80 ppm of sugianimum are equally spaced in terms of the potential effect on body weight. The appropriate contrast is

```
contrast "linear trend" SugiSupp -2 -1 0 1 2
```

The model is a two-way ANOVA with main effects only and a covariate, which can typically be specified with SAS code as

```
freq CellWgt;
class Company SugiSupp;
model Scenario1 Scenario2 = Company SugiSupp
                            RabbitWgt00;
```

Note that scenario 2, with its small interaction effect, does not satisfy this statistical model. But power computations are still perfectly valid. It may be of interest to investigate how a model misspecification affects power.

Contrasts between the standard formulation and each of the alternatives can typically be specified with SAS code as

```
contrast "+0 vs +10" SugiSupp 1 -1;
contrast "+0 vs +20" SugiSupp 1 0 -1;
contrast "+0 vs +40" SugiSupp 1 0 0 -1;
contrast "+0 vs +80" SugiSupp 1 0 0 0 -1;
contrast "linear trend" SugiSupp -2 -1 0 1 2;
```

Significance will be judged at α = 0.0125 for the pairwise comparisons and α = 0.05 for the test of a linear trend in dose response.

Aim of Assessment: The scientists want to ascertain whether 240 rabbits are sufficient to provide adequate power for their planned tests, according to their two scenario models. They wonder whether fewer rabbits might suffice. To investigate the effect of sample size on power, they will also consider a design with only 160 rabbits.

The approach used to calculate power for this situation is explained in the next section, "The Univariate GLM with Fixed Effects." The results computed using UnifyPow (O'Brien 1998) are displayed in Table 3.

α = .0125 for comparisons and .05 for linear trend		Standard Deviation 0.5		0.65	
		Total N		Total N	
Scenario	Test	160	240	160	240
1	+0 vs +10	.047	.067	.032	.043
2	+0 vs +10	.047	.067	.032	.043
1	+0 vs +20	.573	.788	.332	.515
2	+0 vs +20	.529	.746	.301	.473
1	+0 vs +40	.804	.948	.532	.749
2	+0 vs +40	.833	.961	.566	.782
1	+0 vs +80	.942	.994	.737	.912
2	+0 vs +80	.942	.994	.737	.912
1	linear trend	.996	.999	.941	.991
2	linear trend	.996	.999	.946	.992

Table 3. Power Values

So, the small degree of interaction (in scenario 2) barely affects the power. Assuming these scenarios are reasonable, a main-effects-only model should suffice. The study is likely to find significance for the linear trend in dose/respose and the higher formulations of SugiSupp, but it is also very likely that +10 will be deemed "below threshold." The scientists should be conservative in reporting non-significant results for the threshold comparisons, since the power is quite low until +40. If the SD is 0.65, then perhaps only +80 would see a threshold effect. Given the mediocrity of the power values, the scientists realize that they should use all 240 rabbits.

As a side note, a power analysis ignoring the covariate RabbitWgt00 reveals that the maximum possible power with SD = 0.73, for tests other than the linear trend contrast (which has very high power in all cases), is 82.4% for the contrast between +0 and +80 with N = 240.

The next two sections outline power computations for the general framework of linear models with fixed effects (univariate and then multivariate) and alternative strategies for specifying the relevant components.

The Univariate GLM with Fixed Effects

The example in the previous section involves an ANOVA with two factors and a covariate, a special case of the univariate GLM with fixed effects.

Methods for computing power for the general linear model with fixed effects have been developed in a series of papers, providing exact results for univariate models, as well as good approximations for both multivariate models and univariate approaches to repeated measures. Muller et al. (1992) summarize results for these situations, and O'Brien and Shieh

(1992) develop a slightly improved power formula for multivariate models.

The univariate GLM is discussed in this section, with special emphasis on the alternative ways in which you can specify the quantities involved in power computations. These quantities are encompassed by the five-component layout as demonstrated in the examples in this paper.

The multivariate GLM is discussed in the next section. Computations for the univariate approach to repeated measures (with sphericity or without, using Greenhouse-Geisser or Huynh-Feldt corrections) are not discussed here but are similar in spirit to the ones outlined in this section; details can be found in Muller and Barton (1989) and Muller et al. (1992).

The univariate GLM is represented as follows:

$$Y = XB + \epsilon \quad \text{where} \quad \epsilon \sim \mathrm{N}(0, \sigma^2)$$

where Y is the vector of responses, X is the design matrix, B is the vector of effect coefficients, and ϵ is the vector of errors.

The independent variables represented in X may be either categorical or continuous. Consequently, the univariate GLM covers *t*-tests, fixed-effects ANOVA and ANCOVA, and multiple linear regression, which have been discussed along with examples in previous sections. This section outlines a more general framework and expounds on the various ways of expressing the components required for a sample size analysis.

Typically, hypotheses of interest in these models have the general form of a linear contrast

$$H_0 : CB = \theta_0$$

where C is a matrix of contrast coefficients and θ_0 is the null contrast value. Note that this formulation covers the overall test and tests of individual effects as special cases.

The components involved in power computations can be broken down as follows, showing alternative formats for how some of the quantities can be specified:

Study Design:

- design profiles: {essence design matrix} or {exemplary X} or {empirical mean and covariance of X rows}
- sample size: {total sample size and weights of design profiles} or {number of replications of design profiles}

Effects & Variability:

- model parameters: {cell means} or {model parameters using another coding scheme} or {exemplary X and Y}
- error variance: {error standard deviation (root MSE)} or {exemplary X and Y}
- multiple correlation between Y and continuous covariates (if applicable)

Statistical Method:

- model equation
- contrast coefficients
- test statistic (including multiple comparison information, if applicable)
- null contrast value
- significance level

Aim of Assessment:

- various (compute power, sample size, etc.)

The (exact) computation of power is intuitive in the sense that it involves the noncentral F distribution whose noncentrality is computed in exactly the same way as the F test statistic except with estimates ($\hat{B}$ and $\hat{\sigma}$) replaced by conjectured true values. For the equations and other computational details, see O'Brien and Shieh (1992). Although the equations express power as a function of the other components, solutions for sample size and other quantities can be obtained via iteration.

There are several different ways in which you can specify the components required to compute power. You may choose to specify some or all quantities directly, such as the design matrix (X), error standard deviation (σ), and model parameters (B). Recall that in the rabbit example in the previous section, SD was posited directly.

Instead of the full design matrix, you can provide the *essence* design matrix (the collection of unique rows in X) along with the weights or frequencies of each row. This has the benefit of expressing the design profiles and sample sizes independently of each other, since the number of rows in the full design matrix X varies with sample size.

Even if you don't code the X matrix as a cell-means model, you can express the model parameters B as cell means, the collection of mean response values at each factor-level combination. This is often the most familiar coding scheme.

The CONTRAST statement in GLM and other SAS/STAT® procedures can be a handy shorthand way for specifying contrasts of interest in complicated models.

As an alternative to specifying quantities directly, you can formulate an *exemplary data set*, a hypothetical data set having the same format as the one that will eventually be used in the data analysis. Instead of gathering real data, however, you fill the exemplary data set with "pretend" observations that are representative of the scenario for which you want to perform power computations. It summarizes the mean scenarios and cell weights under consideration. Often this approach is easier (than direct parameter specification) for inferring the design, effect values, and (optionally) error standard deviation. Recall that in the rabbit example in the previous section, an exemplary data set was used to specify the design and means.

To provide a minimally useful amount of information, an exemplary data set must contain each design profile that will be used, and the response values must be indicative of conjectured effects. This allows the design structure and effect values to be inferred. If the design profiles occur in the same proportion as they will in the actual study, then the profile weights and error standard deviation can also be inferred. Since the model and statistical test cannot be inferred from exemplary data, they must be specified separately.

Special considerations apply in the presence of continuous independent variables ("covariates"), depending on whether they are involved in the statistical tests. In a randomized design where the covariates X_c are measured at baseline (before randomization) and are *not* included in the contrast, you can compute an approximate power as demonstrated in the rabbit example in the previous section. Conjecture the (multiple) correlation $R^2_{Y|X_c}$ between Y and X_c. Reduce the standard deviation of the residual term to $\sigma(1 - R^2_{Y|X_c})^{\frac{1}{2}}$. Proceed as if the covariates are not in the model, except that the degrees of freedom for the residual is reduced by the number of covariates. This simplification holds only if X_c is uncorrelated with the variables already in the model.

If the covariate distribution differs across groups, then the contrasts apply to the least square means (LSMEANS) rather than to the simple means.

If covariates are included in the statistical tests, then you have two feasible (albeit complicated) strategies to choose from. For contrasts amounting to tests of individual effects, you can re-cast the contrast and effects in terms of correlations and use one of the approaches described in the "Multiple Linear Regression" section. Or for any contrast, you can specify X in its full form or in terms of its empirical mean and covariance.

The Multivariate GLM with Fixed Effects

The multivariate GLM is an extension of the univariate GLM in the sense of having more than one response variable, i.e., Y is a matrix instead of a vector. Important special cases include repeated measures and MANOVA. Although exact power computations are not availiable except in the case of one-d.f. contrasts, O'Brien and Shieh (1992) develop good approximate formulas.

As an example, the model used for the rabbits in the "Multi-Way ANOVA with Fixed Effects and Covariates" section could be extended to a multivariate model by including body weight measurements at a number of different times, say, 12, 24, and 36 weeks.

The multivariate GLM is represented as follows:

$$Y = XB + \epsilon \quad \text{where} \quad \epsilon_i \sim \mathrm{N}(0, \Sigma)$$

where Y is the matrix of responses, X is the design matrix, B is the matrix of effect coefficients, ϵ is the matrix of errors (with rows $\{\epsilon_i\}$), and Σ is the covariance matrix of the Y columns (varying over "within" factor levels). The matrix Σ is often referred to as the covariance of repeated measures.

The hypothesis under consideration is the contrast

$$H_0 : CBA = \theta_0$$

where C is a "between" contrast matrix (involving effects specified by X), and A is a "within" contrast matrix (involving the columns of Y).

All of the sample size analysis components discussed for the univariate GLM also apply for the multivariate model. In addition, you must specify the test statistic, the covariance of repeated measures, and the within contrast matrix. Special forms of the within contrast matrix give rise to special cases such as classical MANOVA, growth profile analysis with time polynomials, and between-trend analysis. Here is a summary of the *additional* required components in a multivariate GLM sample size analysis:

Study Design:

- number of repeated measurements (i.e., number of columns in Y)

Effects & Variability:

- covariance of repeated measures: {covariance matrix} or {type of covariance matrix and relevant parameters (for example, compound symmetry or AR(1))} or {exemplary X and Y}

Statistical Method:

- "within" contrast coefficients
- test statistic: Wilk's likelihood ratio, Hotelling-Lawley trace, Pillai trace, etc.

A Survey of Other Situations

There are many types of linear models, and other approaches to the multivariate models with fixed effects, not covered in the previous sections. This section presents a brief summary for sample size analysis with other popular types of linear models.

Lognormal Data When the data are lognormally distributed in an ANOVA, you can specify effects in terms of mean ratios, supply a conjecture for the coefficient of variation (assumed common across design profiles), and proceed with the same approaches already developed for standard ANOVA models. Lognormal outcomes occur in many situations, such as when the response variable is a probability or growth measurement. Often you can express cell means conveniently in this paradigm, as a fraction of the reference or baseline level.

Multiple Comparisons When a study involves multiple inferences or multiple comparisons, power considerations require specifying precisely which inferences you want power for. Westfall et al. (1999) discuss the issue and give some computational tools.

Mixed Models Currently there is no accepted general standard for power computations in *mixed* linear models, with both fixed and random effects, although methods have been developed for some special cases. It is an active area of research, and currently simulation remains the recommended approach.

Some classes of models with random effects (for example, simple split-plot designs) can be re-cast as multivariate linear models, with the random effect modeled instead as multiple response values. Power analysis can thus be performed using the methods discussed in this paper.

O'Brien and Muller (1993, section 8.5.2) show an exact power computation for a one-way random-effects ANOVA using a multiple of a central F distribution. Lenth (2000) computes approximate power for a wide variety of balanced ANOVA designs with fixed and random effects (where all of the random effects are mutually independent, inducing a compound symmetry correlation structure) by constructing F tests as ratios of expected mean squares and applying Satterthwaite corrections for degrees of freedom. Effects are specified as variance components for random factors and sums of squares for fixed factors.

Power analysis is particularly complicated for mixed models, due to the wide variety of statistical tests that are available. Helms (1992) develops a method for computing approximate power for contrasts of fixed effects, for the approximate F test involving REML estimators of the model coefficients and covariance. The non-null distribution is approximated by a noncentral F with noncentrality estimated much in the same way as O'Brien and Shieh (1992) do for the multivariate GLM, by replacing estimates with conjectured true values.

Simulation Regardless of the availablility of exact or approximate formulas for power computations, simulation remains a viable approach for conducting a sample size analysis for any linear model (indeed, any statistical model). You must be able to simulate realizations of the model (in other words, data sets generated according to the conjectured model, design, effects, and variability), compute the test statistic, and determine when the null hypothesis is rejected. You can repeat this process and estimate power as the percentage of rejections.

Retrospective Analysis This paper has focused on *prospective* power calculations, performed as part of study planning and not based directly on actual data. *Retrospective* calculations attempt to infer the power of a study already performed or estimate power from pilot data. Bias corrections should be used in such retrospective analyses. In addition, since the variability in the observed data can be characterized and propagated through the power analysis, confidence intervals for power can be constructed. These issues are discussed thoroughly in Muller and Pasour (1997), Taylor and Muller (1996), and O'Brien and Muller (1993).

Conclusion

Power and sample size determination has been illustrated for several varieties of linear models, ranging from simple *t*-tests to multivariate models. For any of these situations, you can gather the information required for power computations by considering five aspects of study planning: the design, scenario models representing beliefs about the data, specific conjectures about the effects and variability, the statistical method to be used in data analysis, and the aim of the assessment. The ensuing power computations reveal important aspects about the planned study, such as adequate choices for sample sizes or the likelihood of significant results. Future SAS software will provide analytical tools to help you characterize and solve such problems in power and sample size analy-

sis.

Acknowledgments

We are grateful to Virginia Clark, Keith Muller, Bob Rodriguez, Maura Stokes, and Randy Tobias for valuable assistance in the preparation of this paper.

References

Castelloe, J.M. (2000), "Sample Size Computations and Power Analysis with the SAS® System," *Proceedings of the Twenty-Fifth Annual SAS Users Group International Conference,* Paper 265-25, Cary, NC: SAS Institute Inc.

Gatsonis, C. and Sampson, A.R. (1989), "Multiple Correlation: Exact Power and Sample Size Calculations," *Psychological Bulletin*, 106, 516–524.

Helms, R.W. (1992), "Intentionally Incomplete Longitudinal Designs: I. Methodology and Comparison of Some Full Span Designs," *Statistics in Medicine*, 11, 1889–1913.

Keyes L.L. and Muller, K.E. (1992), "IML Power Program," available at ftp.bios.unc.edu/ pub/ faculty/ muller/power01/distrib/.

Lee, Y.S. (1972), "Tables of the Upper Percentage Points of the Multiple Correlation," *Biometrika*, 59, 175–189.

Lenth, R.V. (2000), "Java Applets for Power and Sample Size," www.stat.uiowa.edu/ ˜rlenth/Power/.

Maxwell, S.E. (2000), "Sample Size and Multiple Regression Analysis," *Psychological Methods*, 5, 434–458.

Muller, K.E. and Barton, C.N. (1989), "Approximate Power for Repeated Measures ANOVA Lacking Sphericity," *Journal of the American Statistical Association*, 84, 549–555 (with correction in volume 86 (1991), 255–256).

Muller, K.E., LaVange, L.M., Ramey, S.L., and Ramey, C.T. (1992), "Power Calculations for General Linear Multivariate Models Including Repeated Measures Applications," *Journal of the American Statistical Association*, 87, 1209–1226.

Muller, K.E. and Pasour, V.B. (1997), "Bias in Linear Model Power and Sample Size Due to Estimating Variance," *Communications in Statistics – Theory and Methods*, 26, 839–851.

Muller, K.E. and Peterson, B.L. (1984), "Practical Methods for Computing Power in Testing the Multivariate General Linear Hypothesis," *Computational Statistics and Data Analysis*, 2, 143–158.

O'Brien, R.G. (1998), "A Tour of UnifyPow: A SAS Module/Macro for Sample-Size Analysis," *Proceedings of the Twenty-Third Annual SAS Users Group International Conference*, Cary, NC: SAS Institute Inc., 1346–1355. Software and updates to this article can be found at www.bio.ri.ccf.org/ UnifyPow/.

O'Brien, R.G. and Muller, K.E. (1993), "Unified Power Analysis for t-Tests Through Multivariate Hypotheses," In Edwards, L.K., ed. (1993), *Applied Analysis of Variance in Behavioral Science*, New York: Marcel Dekker, Chapter 8, 297–344.

O'Brien, R.G. and Shieh, G. (1992). "Pragmatic, Unifying Algorithm Gives Power Probabilities for Common F Tests of the Multivariate General Linear Hypothesis." Poster presented at the American Statistical Association Meetings, Boston, Statistical Computing Section. Also, paper in review, downloadable in PDF form from www.bio.ri.ccf.org/UnifyPow.

SAS Institute Inc. (1999a). *SAS/IML® User's Guide, Version 8*, Cary, NC: SAS Institute Inc.

SAS Institute Inc. (1999b). *SAS/STAT® User's Guide, Version 8*, Cary, NC: SAS Institute Inc.

Taylor, D.J. and Muller, K.E. (1996), "Bias in Linear Model Power and Sample Size Calculation Due to Estimating Noncentrality," *Communications in Statistics –Theory and Methods*, 25, 1595–1610.

Westfall, P.H., Tobias, R.D., Rom, D. Wolfinger, R.D. and Hochberg, Y. (1999), *Multiple Comparisons and Multiple Tests Using the SAS® System*, Cary, NC: SAS Institute Inc.

Contact Information

John M. Castelloe, SAS Institute Inc., SAS Campus Drive, Cary, NC 27513. Phone (919) 531-5728, FAX (919) 677-4444, Email John.Castelloe@sas.com.

Ralph G. O'Brien, Department of Biostatistics and Epidemiology/Wb4, 9500 Euclid Avenue, Cleveland, Ohio, 44195. Phone (216) 445-9451, FAX (216) 444-8023, Email robrien@bio.ri.ccf.org.

Version 1.2

Paper P704

Using the SAS® System to Estimate Sample Size Requirements for Small Sample Confidence Intervals

Jim Penny, Center for Creative Leadership, Greensboro, NC

ABSTRACT

This paper presents an iterative procedure to compute the minimum sample size required for the construction of small sample confidence intervals. The explanation of the process uses the context of computing confidence intervals around the means of item ratings from 360-surveys. The results suggest that the common number of 3 to 5 raters in a group may not provide stable item means, and that the resulting feedback may lack sufficient stability for making valid developmental decisions using item-level feedback.

INTRODUCTION

The use of 360-assessments for developmental purposes has become increasingly popular in the past several years, and the question occasionally arises about the number of raters needed for reliable feedback in 360-assessments. However, to speak to the question about the number of raters, we must first decide what we mean by "reliable." There are many indices of reliability, and they all use, more or less, the idea of replication (Feldt & Brennan, 1989). That is, if we repeat the process, would we achieve the same result? Using the context of replication, the reliability of a 360-assessment is particularly important because people receiving 360-feedback are likely to make behavioral changes in an effort to improve their performance on the job. (London & Beatty, 1993; London & Smither, 1995; Van Velsor & Leslie, 1991)

Much of the information in many 360-feedback reports is rater group means of items and scales. Using the context of replication, we can phrase the question of reliability as "If we repeat the process, will we achieve the same group means?" Unfortunately, if there is one constant in this universe, it is sampling variability, and, even if we repeat the process with the same raters, not just similar raters, we are likely to receive slightly different ratings because the context in which the raters are thinking has likely changed.

However, barring a large contextual shift, it is reasonable to expect that the raters would produce similar ratings at two different times, and the degree of internal consistency that we have seen in the many 360 surveys tends to support this expectation. One could argue further, then, that the question of how many raters we need for reliable feedback is a question of sampling variability, and, given that the 360-feedback often consists of item and scale means broken down by rater groups, that question becomes "How many raters do we need to produce stable means?" That is, how many raters do we need before the random effect of sampling variability begins to negate itself?

We can rephrase this question in terms of the confidence interval; however, to do so changes the question from simply one of "how many raters do I need" because whenever two or more raters are gathered one can always compute a confidence interval around the sample mean. Rather, the question becomes one of interpreting the confidence interval and of how the width of that interval is indicative of the stability of the mean. The resultant question is more intriguing, though less tractable. That question is "How narrow do we need the confidence interval for the range to be practical?" where "practical" is defined in the context of narrow enough to be important though wide enough to be obtainable.

THEORETICAL FRAMEWORK

The confidence interval gives us a numerical range around a parameter estimate, where the confidence represents the likelihood that a repeated estimation of that parameter would produce a result within that range (Neter, Wasserman, & Kutner, 1985, p. 71). Alternatively, we can say that we know the likelihood that this interval captures the population value of the parameter (Glass & Hopkins, 1984, p. 183). Using the mean as the statistic of interest, we call the sample mean a point estimator, and refer to it as our one best guess at the true mean. However, the one thing that we can say for sure is that the point estimator is wrong; it may be close to the true value, but it is not the true value, and this is where the confidence interval enters this discussion, providing a margin of error for the point estimator.

The first step, then, in computing a confidence interval is to compute the mean. The next step is to choose the confidence of the interval, where confidence represents the likelihood that the interval contains the true value. A common value is 95%, which is analogous to the statistical significance level of .05. In addition to the selection of the probability, the computation of the confidence interval also requires the sample standard deviation and the sample size.

We can write the 95% confidence interval with the following equation (Neter, Wasserman, & Kutner, 1985, p. 11)

$$95\%CI = \bar{x} \pm t_{df,.975}\left(\frac{s_x}{\sqrt{n}}\right)$$

where *t* is the *t*-statistic. The *df* in the equation refers to the degrees of freedom and is equal to *n-1* where *n* is the number of subjects. We call the *x* with the bar overhead "x-bar," and it represents the sample mean. The symbol s_x represents the sample standard deviation, and the symbol *n* represents the sample size.

We sometimes call the value after the plus-and-minus in the above equation the "margin of error," and we use it as an indicator of precision in the point estimator. The smaller the margin of error, the more certainty we have in the precision of the point estimator, where precision suggests that the point estimator is close to the true value. Using the notation from the last equation, we can write an expression for the margin of error, using the symbol MOE to represent the value.

$$MOE = t_{df,.975}\left(\frac{s_x}{\sqrt{n}}\right)$$

If we ignore the dependency of *t* on the value of *n*, we can rearrange this equation and solve it for *n*.

$$n = \left(\frac{MOE}{t_{df,.975}s_x}\right)^2$$

Now, all that we need to estimate the sample size is an acceptable margin of error, a value for *t,* which unfortunately depends on *n*, and the sample standard deviation.

ESTIMATING THE SAMPLE STANDARD DEVIATION

The problem here, and the reason for the word "estimating" in the section heading, is that we often do not have a sample from which to compute the standard deviation, and we do not have it because we have not yet decided how many people go into that sample. We could compute the standard deviation using data we have already collected, and use that value to estimate the value that we might expect in the sample that we do not yet have. However, it is likely that 360-assessment participants are going to select small numbers of raters, perhaps in groups as small as three to five, and we know that the means and standard

deviations in such small samples can be unstable, rendering any estimation based on extant data questionable, so we will likely benefit from a bit of theoretical reasoning.

It is nearly common knowledge that the standard deviation of Likert-type items such as appear on many 360-surveys is more-or-less one point (Church & Bracken, 1997; Gregarus & Robie, 1998; Rothstein, 1990). This phenomenon is the direct result of assuming a normal distribution underlies the discrete rating scale. For example, if you assume that a normal distribution centrally underlies a 5-point Likert-type rating scale, then the central six standard deviations of the distribution which subtend approximately 99% of the data have a range from 1 to 5, suggesting an approximate standard deviation of four-sixths of a point.

Unfortunately, many factors can influence the distribution of ratings produced by the use of Likert-type items. For example, a scale that does not have symmetrical anchors may produce distributions that differ from those produced by scales with symmetrical anchors (Penny & Johnson, 1999). In addition, one could anticipate that different groups of raters could produce different distributions. For example, the raters of one group may have more contact with the person they are rating than do the members of another group who have less opportunity to observe (Rothstein, 1990). Moreover, the ratings from supervisors are likely to exhibit less variability than the ratings of direct reports and peers (Conway & Huffcutt, 1997). If the targets of the survey function toward one end of the scale, then it is likely that the ratings will not span the entire scale Penny & Johnson, 1999). For instance, it is unlikely that a high performing executive is going to receive low ratings on items having to do with being a successful executive. Lastly, the purpose of the survey is likely to have an effect on the distribution of the ratings (Zedeck & Cascio, 1982).

Hence, there are many reasons why some items may elicit a tighter distribution of responses from raters while others may elicit a looser distribution. However, the point remains that the usual range of standard deviations of Likert-type items is about .5 point to about 1.5 points and usually rounds to 1 point. Hence, when existing data do not exist to guide one's selection of a standard deviation, or when it seems dubious to use existing data, it may be reasonable to expect a value of about 1 point for the standard deviation of item ratings. For this research, we chose to consider five values of the rating standard deviation. There values were .50, .75, 1.0, 1.25, and 1.5 points.

ESTIMATING THE MARGIN OF ERROR

What span of points on a 5-point Likert-type scale constitutes a sufficiently narrow range for an average? If one receives an average rating of 3.0 from a group of raters, then would an interval of 3.0 plus or minus 1.0 be sufficient? To answer that question, we might ponder for a moment just what that interval suggests. Recall that one interpretation of the 95% confidence interval is that there is a 95% probability that the interval contains the true mean. This interpretation then suggests that the true mean could be any number between the ratings 2 and 4 on this scale, and given 2 to 4 spans 50% of the rating scale, it is not difficult to argue that such a range is nearly meaningless.

What we need, then, is some idea for how small the margin of error needs to be, and it is likely that different distributions of ratings will suggest different margins of error. The expected distribution of the ratings will surely influence the choice of the margin of error, but it may be that the purpose of the survey has a greater effect on this choice. For instance, a survey used for selection may require smaller margins of error than would a survey used for developmental purposes. The explicit choice of a margin of error seemed beyond the scope of this discussion, so we examined four values of the margin of error. These values were .25, .5, .75, and 1 point.

COMPUTATIONAL METHODOLOGY

We have values, now, for the standard deviation of the rating and the margin of error. We could use the prior equation to compute the sample size if we knew the value of t. The difficulty is that t is dependent on the sample size, and, because we are computing the sample size, we cannot look up the appropriate value of t. Moreover, we cannot readily solve t for n. We have to use numerical methods to estimate the value of n for the chosen values of the standard deviation and the margin of error.

One iterative numerical procedure that we can consider with this problem begins with an initial value of t corresponding to the smallest value of n that is possible. For a 95% confidence interval with $n=2$, $t=12.71$, and we can use this value of t to estimate the next value of n. Obviously, this new value of n will be incorrect, but we can call it our first best guess. We then use this best guess for n to update the value of t, and that update enables our re-estimation of n. We continue this process until the values of n are not changing substantially. Acton (1970, pp. 41-86) and Press, Flannery, Teukolsky, & Vetterling (1989, pp. 270-308) describe such an iterative procedure for finding the roots of transcendental equations. Appendix 1 presents the code to implement this procedure using the SAS® System.

RESULTS

Table 1 provides a list of minimum sample sizes for both 90% and 95% confidence intervals using the five values of the standard deviation and the four values of the margin of error presented in the preceding text. We rounded the values of n to the nearest greater integer. The numerical procedure generally converged in about 5 to 6 iterations; it did not converge when the margin of error exceeded the standard deviation of the ratings. The time required for the program to reach convergence was negligible, and we allowed the program to write its few lines of output to the SAS log file from which we could copy the final value of the sample size. Additional code to route the output from the log to an external file would be easy to incorporate, and would provide the ability to collect the output from several programs executions in a single file.

DISCUSSION

The sample sizes given in Table 1 are a bit larger than those frequently encountered in many 360 surveys where one generally finds 3 to 5 raters in a group. Moreover, Greguras & Robie (1998) and London & Smither (1995) suggested that the usual number of raters selected for 360-surveys may be too small for the estimation of stable means of most items and some scales. In addition, Bozeman (1995) suggested that feedback based on small samples might not be useful, and Kluger & DeNisi (1996) presented evidence to suggest that perhaps as many as one-third of those who made behavioral changes after reviewing 360-feedback actually experienced decreased performance. However, Van Velsor & Leslie (1991) suggested that some circumstances exist in which such ratings may be sufficiently stable for use in feedback reports.

Our study, the context of which was the presentation of means based on responses to single items, appeared to agree more with the suggestion of Greguras & Robie (1998). It is without doubt that the use of more raters will generally produce distributions with smaller standard errors due to the increase in sample size. However, using more raters adds to the resource required to complete the 360-survey, and, given the growth in the use of 360-assessment, some managers may feel the need to use fewer raters. That response is understandable. However, doing so may produce means that lack sufficient stability, or accuracy, for use in making useful developmental decisions.

One might suggest there are other means by which we can reduce the standard errors of the rating means. For example, the computerized administration of the survey might decrease the time required to complete the items, suggesting the possibility of more raters for the same amount of resource currently used. The techniques of computer adaptive testing (CAT) could substantially reduce the time required to complete a survey by presenting only

as many items as we need to estimate a scale score (Thissen, 1990). However, the use of CAT not only presumes large samples for the estimation of item characteristics but also likely precludes, or at least makes difficult, the presentation of item level feedback owing to the nature of CAT to use different items with different raters while estimating the latent trait represented by the items (Wainer & Mislevy, 1990).

Alternative methods may exist to reduce the standard errors of the rating means. For instance, the selection of raters can influence the standard deviation of the ratings, and raters who have additional opportunity to observe the behaviors assessed by the survey will likely respond with reduced measurement error (Rothstein, 1990). Hence, if circumstances necessitate a small number of raters, then the selection of raters who have more opportunity to observe the person they are rating is likely to reduce the magnitude of the standard errors of the rating means. Moreover, coaching raters on the behaviors assessed by the survey seems likely to heighten the awareness of the raters, to make them efficient observers, and, subsequently, to reduce the measurement error in their ratings. However, coaching requires time, and the more sophisticated ratings that coached raters may provide likely will require more time on the part of the raters.

The importance of performance and behavioral information from the people with whom one works is without question. However, the manner in which we gather that information may influence the reliability of the interpretation of the feedback report, which, in turn, limits the validity of that interpretation, resulting in some developmental decisions that could be sub-optimal.

REFERENCES

Acton, F. S. (1970). *Numerical methods that work*. New York: Harper & Row.

Bozeman, D. P. (1995). Interrater agreement in multi-source performance appraisal: A commentary. *Journal of Organizational Behavior*, *18*, 313-316.

Church, A. H., & Bracken, D. W. (1997). Advancing the state of the art of 360-degree feedback. *Group and Organizational Management*, *22*, 149-161.

Conway, J. M., & Huffcutt, A. I. (1997). Psychometric properties of multisource performance ratings: A meta-analysis of subordinate, supervisor, peer, and self-ratings. *Human Performance*, *10*, 331-360.

Feldt, L. S., & Brennan, R. L. (1989). Reliability. In Linn, R. l. (Ed.) *Educational measurement, 3rd Edition*. New York: Macmillan.

Greguras, G. J., & Robie, C. (1998). A new look at within-source interrater reliability of 360-degree feedback ratings. *Journal of Applied Psychology*, *83*, 960-968.

Glass, V. G., & Hopkins, K. D. (1984). *Statistical methods in education and psychology, 2nd edition*. Needham Heights, MA: Allyn and Bacon.

Kluger, A. N., & DeNisi, A. (1996). The effects of feedback interventions on performance: A historical review, a meta-analysis, and a preliminary feedback intervention theory. *Psychological Bulletin*, *119*, 254-284.

London, M., & Beatty, R. W. (1993). 360 degree feedback as competitive advantage. *Human Resource Management*, *32*, 353-373.

London, M., & Smither, J. W. (1995). Can multi-source feedback change perceptions of goal accomplishment, self-evaluation, and performance-related outcomes? Theory-based applications and directions for research. *Personnel Psychology*, *48*, 803-839.

Neter, J., Wasserman, W., & Kutner, M. H. (1985). *Applied linear statistical models: Regression, Analysis of variance, and experimental designs*. Homewood, IL: Richard D. Irwin.

Penny, J., & Johnson, R. L. (Nov. 1999). *The Effect of Rating Augmentation on Likert-type Responses: An Exploratory Empirical Study*. Paper presented at the annual meeting of AEA, Orlando, Florida.

Press, W. H., Flannery, B. P., Teukolsky, S. A., & Vetterling, W. T. (1989). *Numerical recipes in Pascal*. Cambridge, MA: Cambridge University Press.

Rothstein, H. R. (1990). Interrater reliability of job performance ratings: Growth to asymptote level with increasing opportunity to observe. *Journal of Applied Psychology*, *75*, 322-327.

Thissen, H. (1990). Reliability and measurement precision. In Wainer, H. (Ed.) *Computerized adaptive testing: A primer*. Hillsdale, NJ: Lawrence Erlbaum.

Thissen, H., & Mislevy, R. J. (1990). Item response theory, item calibration, and proficiency estimation. In Wainer, H. (Ed.) *Computerized adaptive testing: A primer*. Hillsdale, NJ: Lawrence Erlbaum.

Van Velsor, E., & Leslie, J. B. (1991). *Feedback to managers: Vol 1. A guide to evaluating multi-rater feedback instruments*. Greensboro, NC: Center for Creative Leadership.

Zedeck, S., & Cascio, W. F. (1982). Performance appraisal decisions as a function of rater training and purpose of appraisal. *Journal of Applied Psychology*, *67*, 752-758.

Table 1: Minimum sample sizes for 90% and 95% confidence intervals

Margin of Error	Standard Deviation	Sample Size if Confidence=.90	Sample Size if Confidence=.95
0.25	1.50	100	141
0.50	1.50	27	38
0.75	1.50	13	18
1.00	1.50	9	12
0.25	1.25	70	99
0.50	1.25	19	27
0.75	1.25	10	14
1.00	1.25	7	9
0.25	1.00	46	64
0.50	1.00	13	18
0.75	1.00	7	10
1.00	1.00	5	7
0.25	0.75	27	38
0.50	0.75	9	12
0.75	0.75	5	7
1.00	0.75	-	-
0.25	0.50	13	18
0.50	0.50	5	7
0.75	0.50	-	-
1.00	0.50	-	-

Note: The numerical procedure used in this analysis does not converge if the margin of error exceeds the standard deviation of the ratings.

Appendix 1: SAS code to estimate minimum samples sizes for small confidence intervals

```
/* This program estimates the number of raters necessary to achieve
a 95% confidence of a minimum width

confi  = confidence
sx     = standard deviation of ratings
moe    = acceptable margin of error
n_hat  = current estimate of minimum sample size
t_hat  = t-statistic corresponding df=n_hat-1 and p=1-alpha/2
n_prev = value of n_hat in prior iteration

Note:  The TINV function returns the t associated with the
       probability in the left tail.  This code contains an adjustment
       to produce the positive t needed for the confidence interval.  For
       example, if CONFI=.95 and n=2, then the program passes the value
       .975 to TINV, which then returns 12.71

*/

data _null_;

   confi = .90;
   sx = .75;
   moe = .25;

   /* begin iterations with n=2 raters */
   n_hat = 2;
   t_hat = tinv(confi+(1-confi)/2, n_hat-1);

   put 'Beginning estimation procedure';
   do until ((n_hat - n_prev) = 0);
      put 'Current estimate =' n_hat;
      n_prev = n_hat; /* retain prior estimate of n */
      n_hat = ((t_hat*sx)/moe)**2; /* compute new estimate of n */
      t_hat = tinv(confi+(1-confi)/2,n_hat-1); /* compute t for ne n */
      n_hat = int(n_hat) + 1; /* truncate decimals and round up by 1 */
   end;
   put 'Convergence achieved at n =' n_hat;
run;
```

Contact Information

Your comments and questions are valued and encouraged. Contact the author at:

Jim Penny
Center for Creative Leadership
One Leadership Place
Greensboro, NC 27438-06300
Work Phone: 336-286-4442
Fax: 336-286-4434
Email: pennyj@leaders.ccl.org
Web: www.ccl.org

Optimal Solution of Discrete Resource Allocation Problems with SAS/OR ®Software

by LTC Doug McAllaster, US Army Logistics Management College, Fort Lee, VA

ABSTRACT

This paper is a tutorial on how to use SAS/OR PROC LP to solve integer programming problems. SAS/OR (operations research) software includes PROC LP which solves linear programming (LP) problems. This same procedure also includes the capability to solve integer programming (IP) problems, a very important class of problems which models discrete (binary: yes or no) decisions. This paper describes some classic integer programming models, all of which PROC LP can solve. Although the paper is primarily a verbal description of classic LP & IP models (and includes their algebraic formulations in the appendix), my presentation of this paper will focus on graphical representations (which are much more readily intelligible) like those in the appendix.

OVERVIEW

We begin with easier problems and work our way up to the more difficult ones. First, we describe three foundational LP models which are classic network flow problems: (1) assignment model, (2) transportation model, & (3) transshipment model. These problems have a special structure and are easier to solve using PROC NETFLOW rather than with the more general PROC LP. Then we examine three important and special applications of the transshipment model: (1) shortest path, (2) maximal flow, (3) maximal matching.

Second (having warmed up), we describe some pure integer programming models which are classic location models. These models are concerned with optimal placement of facilities to serve customers. We describe three classic models: (1) complete cover, (2) maximum cover, and (3) median placement.

CLASSIC NETFLOW MODELS

The assignment model gives an optimal assignment of people to jobs. Here we have N people and N jobs. There is some cost associated with a person performing a job. The model will assign a person to each job at minimal cost. We represent this model using a network where each node on the left represents one person and each node on the right represents one job. See the assignment model panel.

The transportation model has a very similar structure and network. This model is concerned with shipping goods from supply nodes (plants or warehouses) to demand nodes (customers) in order to minimize the transportation cost of meeting all the demand. Left hand nodes are plants, right hand nodes are customers. The numbers represent the supply of "widgets" at each plant and the demand for "widgets" at each customer. See the transport model panel.

Finally, the transshipment model is the most general network flow model. It permits a more general structure for the network. These networks can include intermediate nodes. Furthermore, the intermediate nodes may be supply, demand, or transshipment nodes. The latter is a node which simply sends out whatever comes in, i.e., it's neither a supply nor a demand, it's just a node at which branching takes place, a "middleman," if you will. See the transship model panel.

The obvious application of the transshipment model is to meet customer demands from supplies at minimum cost. Thus, this is often called the minimum cost network flow problem (MCNFP). Less obvious, however, is that the MCNFP has very wide applicability. In fact, Glover, Klingman, & Phillips (see references) wrote an entire book describing the practical application of these models. For example, optimization analysts often use it to solve multiperiod production and inventory planning problems.

INPUT TABLES

PROC NETFLOW requires two data tables to solve this model, nodedata and arcdata. The nodedata table requires two columns: node and supdem. Node is character, supdem is numeric, where positive integers are the supply at a node and negative integers are demands. The arcdata table requires three columns: from, to, and cost. Columns from and to are character and are the beginning and ending nodes of the arc, respectively. (Naturally, each from and to value in arcdata must match a node value in nodedata.) The cost column gives the cost to ship a unit along the arc.

From these two tables, PROC NETFLOW internally generates a single equation (or constraint) for each node. Each node's equation requires: (the sum of flows out of it) minus (the sum of flows into it) equal to (its supdem). The appendix has the algebraic formulation of the transshipment model. Both the assignment and transportation models are special cases of the transshipment model.

The output table is a listing of the flow on each arc. This flow minimizes total cost while meeting all of the nodal (supply and demand) constraints.

SEMINAL APPLICATIONS

We now consider two seminal applications of the MCNFP. The shortest path problem solves the problem of finding the shortest path from some source node to some demand node, through a set of intermediate transshipment nodes. We model this problem using the MCNFP with a supply of unity at the source node and a demand of unity at the sink node. Note that the solution to this problem is a single continuous path through the network from source to sink.

The maximal flow problem is similar, except that now our problem is to find out how many "widgets" we can get through our network from the source to the sink. This requires a simple (but not obvious) modification to the network. We add a new "return" arc from the sink back to the source and simply tell the model to maximize the flow along it. Note that all of the nodes are transshipment nodes, that is whatever flows in also flows out. One can think of the model as increasing the flow along the return arc until we achieve the maximum capacity of the network.

We conclude with a special and important application of the maximal flow problem to solve the maximal (bipartite) matching problem. In this situation, we have two disjoint sets, e.g., men and women. We indicate compatibility of man to woman with the existence of an arc between them. We introduce the return arc and simply maximize its flow to give us the matching which maximizes the number of couplings.

NETWORK ADVANTAGES

Network flow models have some great advantages over the more general linear programming models. These models are easy to comprehend, largely because they have a readily intelligible, graphical representation. Furthermore, these models are quite easy to code using SAS. The programmer simply provides a table of nodes and a table of arcs.

Next, these models enjoy a special internal structure which makes the solution times very fast (computationally easy). This structure also insures that the optimal flow values are integer. We get integer answers naturally; that is, without specifically constraining the optimizer to use much more complicated and potentially intractable integer programming solution (branch and bound) methods . That is, we get integer answers for free.

Finally, these models are very widely applicable to many business situations. Analysts often model a physical situation in which the main constraint is simply that "widgets" do not disappear. This "conservation of flow" is a fundamental fact in much of the physical universe.

CLASSIC LOCATION MODELS

We now describe some pure integer programming models which are classic location models. Location models enable us to determine the optimal placement of facilities to service some customers. Again, these models are quite easy to comprehend due to their physical and visual representation on the Euclidian plane. We consider three classics: complete cover, maximum cover, and median placement.

COMPLETE COVER

Consider the problem of locating fire stations to service cities. Here we obviously require a complete cover of all cities. We have a list of candidate sites for fire stations, a list of cities we must cover, distances between sites and cities, and some critical distance within which each city must have a fire station. We need to minimize the number of fire stations while providing complete coverage of the cities. This problem is a discrete one. That is, our solution must tell us whether or not to place a fire station at a candidate site. Also, note that we need to know both how many and where to put the fire stations.

We model this situation with a bipartite network. One side has the candidate fire station locations and the other has the cities we must cover. We derive the existence of an arc between a site and city from the distance table and the critical distance. See the complete cover panel in the appendix.

In this model we have several binary unknowns to consider simultaneously. That is, we first must consider the sites as being either "on" or "off." Then, we consider that turning a site node "on" enables all the arcs which emanate from it, thus covering the respective cities.

An example of a covering problem on a map of the US is in the appendix. This example has 24 customer cities which must be covered by up to 8 candidate sites. Sites can cover cities which are within 999 miles. The solution shows that we can cover all 24 cities using only three sites. Also note that five cities are covered twice.

There are a few important differences between the former network flow models and this covering model. Here the fire stations do not exist yet! Furthermore, we are not flowing known quantities through the network. For example, one fire station can cover multiple cities. Likewise, a city may be covered by multiple fire stations. We cannot us network flow methods since we do not have fixed supplies and demands. Thus, we cannot use PROC NETFLOW.

The algebraic version of the model is in the appendix. It is fairly straightforward. Our binary analysis decision variable, Xi, is whether or not we place a facility at a candidate site. Our objective is to minimize the total number of facilities and our constraint is to cover each city at least once.

MAXIMUM COVER

Now we consider the less critical situation of locating libraries to provide service to cities. Here we still have candidate sites and cities to service. However, we do not have to completely cover each and every city (nobody burns without books). Rather, we consider the number of citizens in a city as the measure of importance for covering it. Our payoff is pleasing the most patrons. Another difference here is that we have some fixed number of libraries we will place. We still have the distance matrix between sites and cities as well as the some cutoff distance within which we consider a city covered. Thus, the only addition to our basic sites to cities graph is that we now have payoff values associated with the city nodes. See the maximum cover panel in the appendix.

The algebraic version of the model is also in the appendix. This model is more complicated in that we have an additional set of binary decision variables, Zi, indicating whether a city is covered by some site. The objective is to maximize the grand total sum of covered patrons. The simple constraint is to place no more than the given (P) number of libraries. The other linking constraint insures that we only get credit for covering a city if it is within the cutoff distance of some site.

MEDIAN PLACEMENT

Our third classic location problem is concerned with minimizing the distance traveled from service sites to customer cities. Thus, the major difference is our method for handling distance. In the previous models, there was a critical or cutoff distance within which a city was covered, outside of which it was naked. Now, we consider distance in a linear (not binary) fashion, i.e., any service site can cover any city, but our objective is now to place our given (P) sites in the "weighted middle" of the cities. Thus, we are back to meeting the demand at every customer city, it's just that some folks have to make a long trip for service.

Of course, we know this is true in practice; for example, that medical facilities are located near population centers, small towns don't have retail super-centers. There are a plethora of applications of this model: for locating health clinics, post offices, grocery stores, et cetera.

Our basic network requires only slight modifications. We still have candidate sites on the left and cities (customers) on the right. Furthermore, we retain the demand weight (population) for each city as in the maximum cover problem. The major difference is that our network has more arcs since every site can service every city. (That is, our we cannot reduce our original dense distance matrix to a sparse binary one, as in the covering problems.) See the median placement panel in the appendix.

The algebraic model for this problem is also in the appendix. The algebraic model is significantly different from covering problems in that we now have a set of Yij binary decision variables for the assignment of sites to cities. We require this more extensive set of variables since we now must keep track of who goes where for service in order to accurately account for the total travel distance of customers. The objective minimizes the population weighted distance from cities to sites. The first constraints simply insures we build P sites. The second constraints insures that each city (customer) has a (service) site. The third linking constraint ensures that if a site doesn't exist, then no customer can go there for service.

NATIONAL GUARD

The Army National Guard (ARNG) annually faces a discrete location problem. I have simplified both the scope and details of the problem for this presentation. See Murty and Djang for a complete description. Each spring ARNG places a half-dozen mobile combat trainers at armories for the coming summer training cycle. The is the median placement problem; we we have a given number of trainers (P), candidate sites (large ARNG centers capable of housing incoming troops and supporting the training), and customer cities (consider each state's centroid, weighted by its ARNG population). See the appendix for a SAS/GRAPH map of this problem and its solution.

PROC LP

Although we were able to display the above pure integer programming problems on a network, these are not network flow models since we cannot fix the supply & demands a priori. As mentioned above, this is unfortunate since PROC NETFLOW is amenable to very easy coding.

Rather, we have to solve these IP models with PROC LP. Now, PROC LP can solve any general linear and integer programming problem. However, as a result of its general problem solving capability, the analyst must specify the objective and constraint equations explicitly. This is not a trivial task. But, of course, DATA STEP or PROC SQL programming provides adequate capability to generate the input table.

INPUT TABLE

PROC LP can accept the input table in either dense or sparse format. The appendix has both formats for our complete cover model. The dense format has a row for each equation in the model. Each variable (column) is an analysis decision variable. This format directly reflects the algebraic equations. However, since an LP may have hundreds of decision variables, the more common format is the sparse (or coefficient) format. Here we have an observation for each non-zero coefficient and identify the coefficient's location in the model equations with a row and col variable. The type variable identifies the type of equation.

CONCLUSION

SAS/OR software can solve integer programming problems. This paper describes some classic integer programming applications, their formulations, graphical representations, and the input data required for PROC LP to solve a problem.

REFERENCES

The Wiley-Interscience Series in Discrete Mathematics and Optimization includes two excellent books which are readily intelligible to the interested reader.

Glover F., Klingman D., and Phillips N., *Network Models in Optimization and their Applications in Practice*, 1992, John Wiley and Sons.

Daskin, Mark. *Network and Discrete Location: Models, Algorithms, and Applications*, 1995, John Wiley and Sons.

SAS Institute Inc. *SAS/OR User's Guide Version 6 First Edition*, Cary, NC: SAS Institute Inc. 1989. 479pp.

Murty, K. and Djang, P., 1999. The US Army National Guard's Mobile Training Simulators Location and Routing Problem. *Operations Research* 47(2) March- April, 175-182.

TRADEMARK

AUTHOR CONTACT

You may contact me at: McAllasterD@Lee.Army.Mil

TRANSSHIPMENT MODEL determine flows thru network to meet demands but not exceed supplies

analysis decision variables and input data

X_{ij} – advar	– # units to flow from node i to node j thru arc(i,j)
B_i – input	– net supply or 'balance' at node i
C_{ij} – input	– cost per unit flow thru arc (i,j)

objective minimize grand sum of all flows*costs

$$\sum_{ij} C_{ij} X_{ij}$$

subject to at every node $\boxed{\text{i}}$:

SumFlowOut – SumFlowIn	= Balance	– for every node i
$\sum_j X_{ij} - \sum_k X_{ki}$	$= B_{\boxed{\text{i}}}$	– for every node i

---

COMPLETE COVER locate minimum # of sites to cover every customer node

analysis decision variables and input data

X_j – advar	– whether (binary) we place facility at candidate site j
A_{ij} – input	– whether (binary matrix) site j can cover node i

derive binary matrix A_{ij} from a table of site to city distances
and a critical distance D_c which defines possible coverage

objective minimize number of facilities at candidate sites

minimize $\sum_j X_j$

subject to cover every customer node i at least once

$$\sum_j A_{ij} X_j \geq 1 \qquad \text{– for every customer node i}$$

MAXIMUM COVER locate **P** facilities to maximize covered demand
(without necessarily covering every demand node)
vice covering all nodes with minimum # of sites

analysis decision variables and input data

X_j – advar	– whether (binary) we place facility at site j
Z_i – advar	– whether (binary) node i is covered by some site j
A_{ij} – input	– whether (binary matrix) site j covers node i
H_i – input	– demand at node i
P – input	– number of facilities to place

objective maximize covered demand

$$\text{maximize} \quad \sum_i H_i Z_i$$

subject to

$$\sum_j A_{ij} X_j \geq Z_i \qquad \text{– for every customer node i}$$

$$\sum_j X_j \leq P$$

MEDIAN PLACEMENT locate **P** facilities to minimize weighted distance between customers & facilites

analysis decision variables and input data

H_i – input	– demand load at customer node i
D_{ij} – input	– distance from customer node i to candidate site j
P – input	– number of facilities to place
X_j – advar	– whether (binary) we locate a facility at candidate site j
Y_{ij} – advar	– whether (binary) facility at site j serves entire demand at node i

objective minimize demand weighted distance from customers to facilities

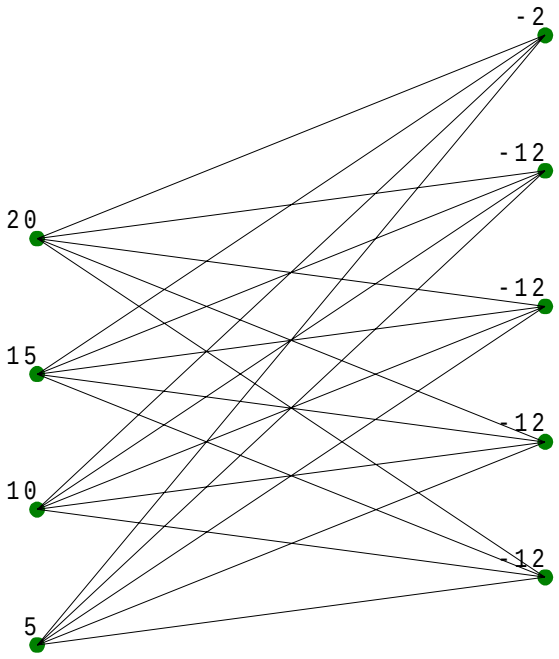
$$\text{minimize} \quad \sum_{ij} H_i D_{ij} Y_{ij}$$

subject to

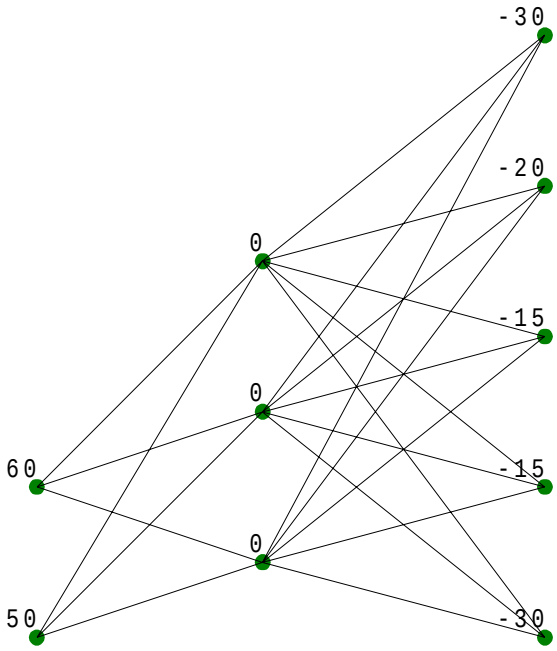
$$\sum_j X_j = P$$

$$\sum_j Y_{ij} = 1 \qquad \text{– for every customer node i}$$

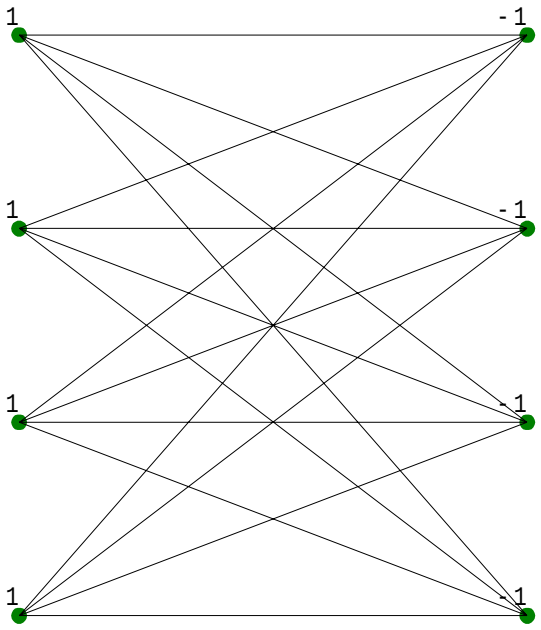
$$Y_{ij} \leq X_j \qquad \text{– for every ij pair}$$



Transport



Transship



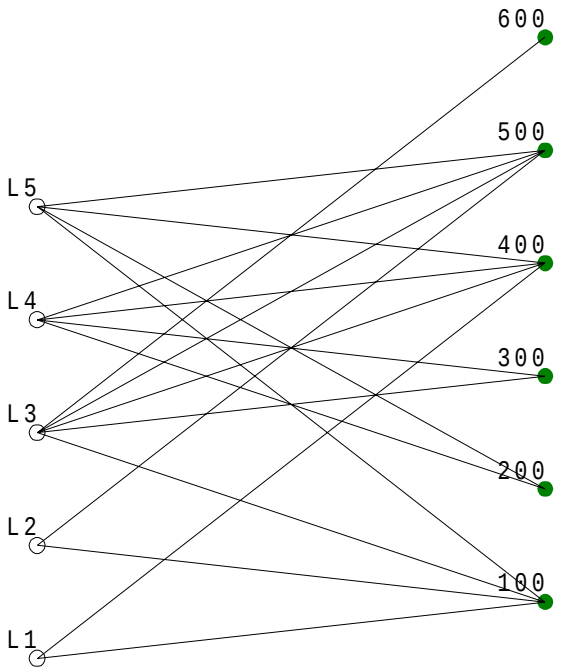
Assign

Network Flow Models

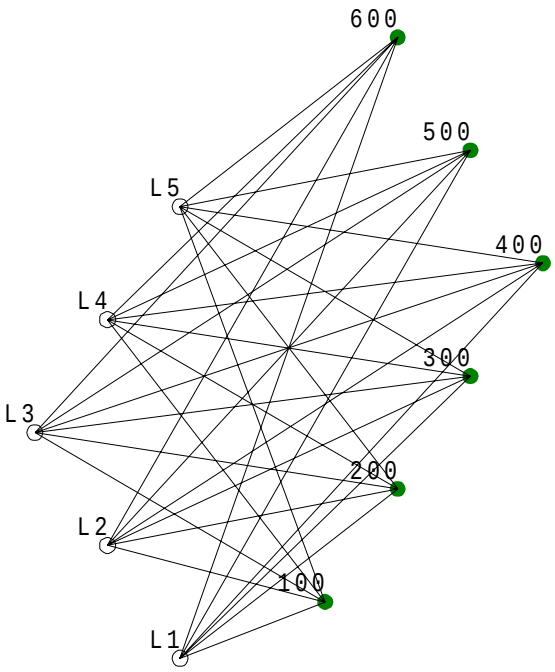
Assignment Model

Transportation Model

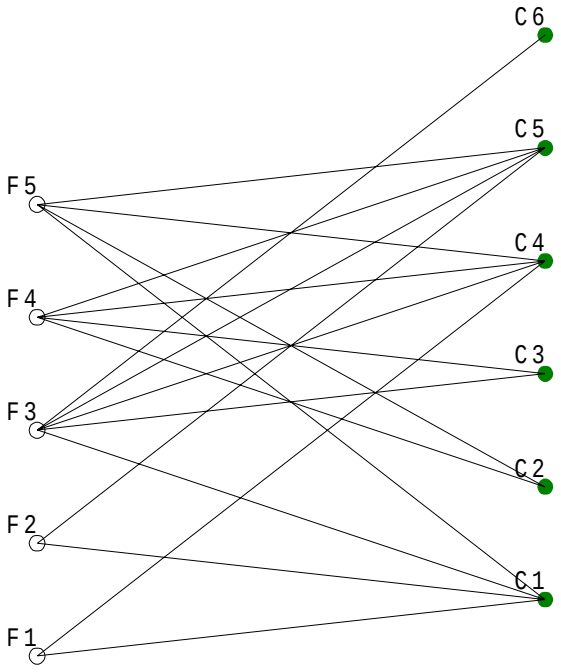
Transshipment Model



Max_Cover



Med_Place



Com_Cover

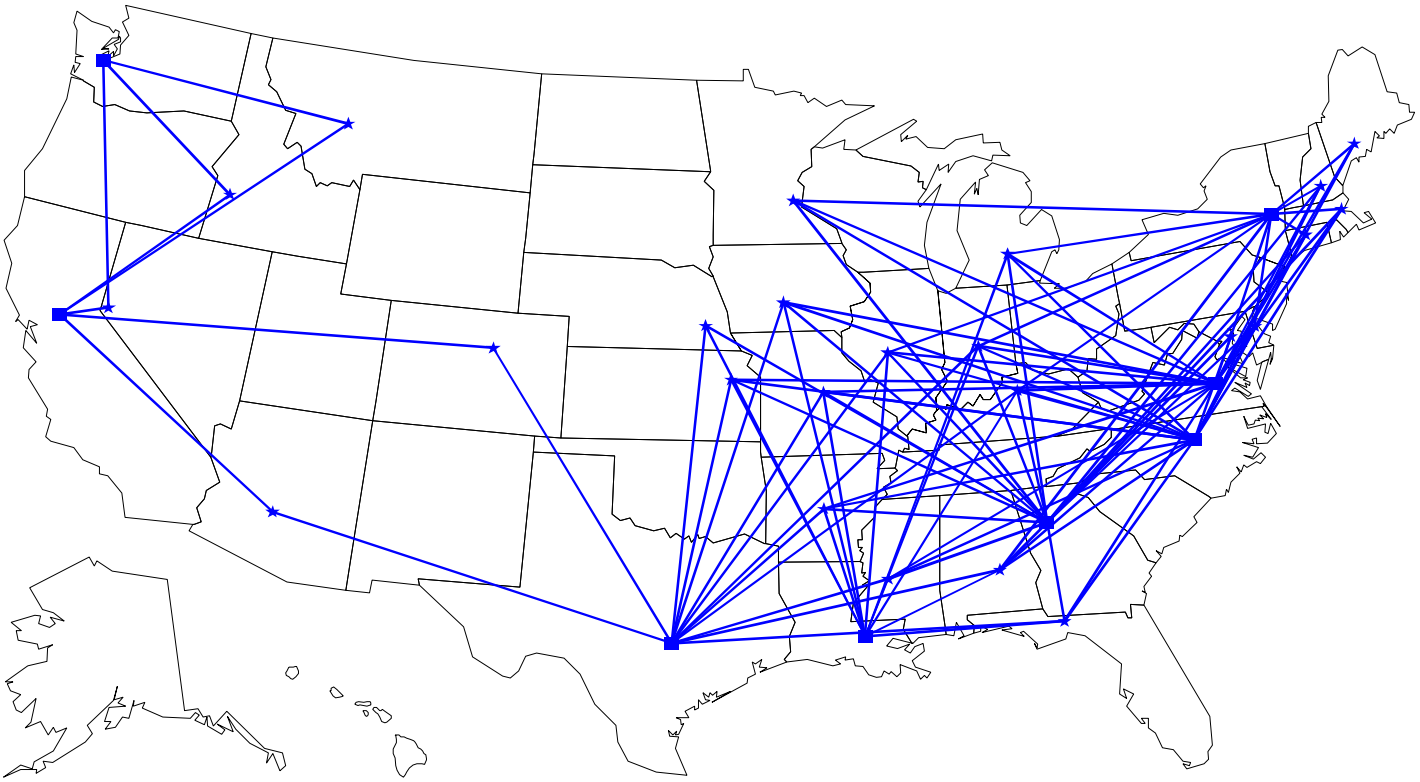
Covering Models

Complete Cover

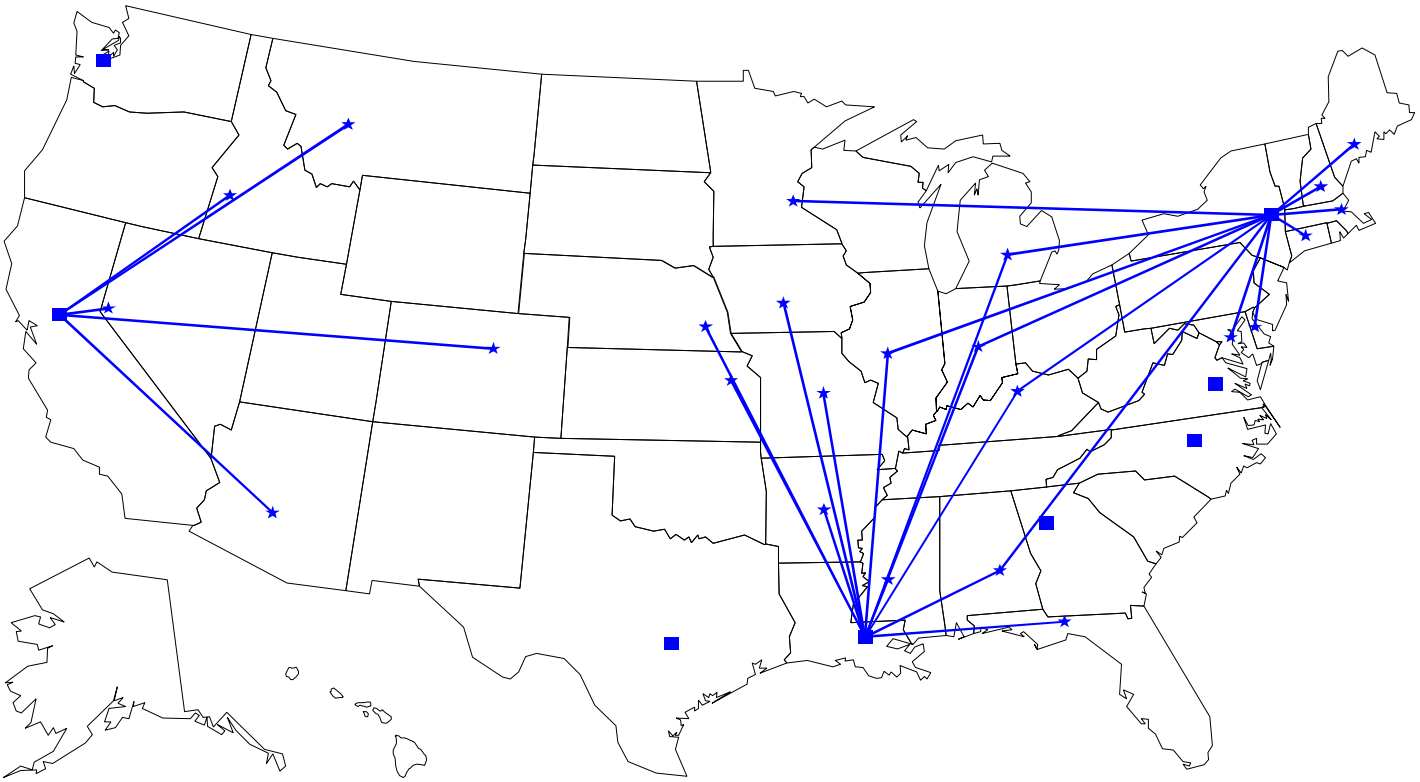
Maximum Cover

Median Placement

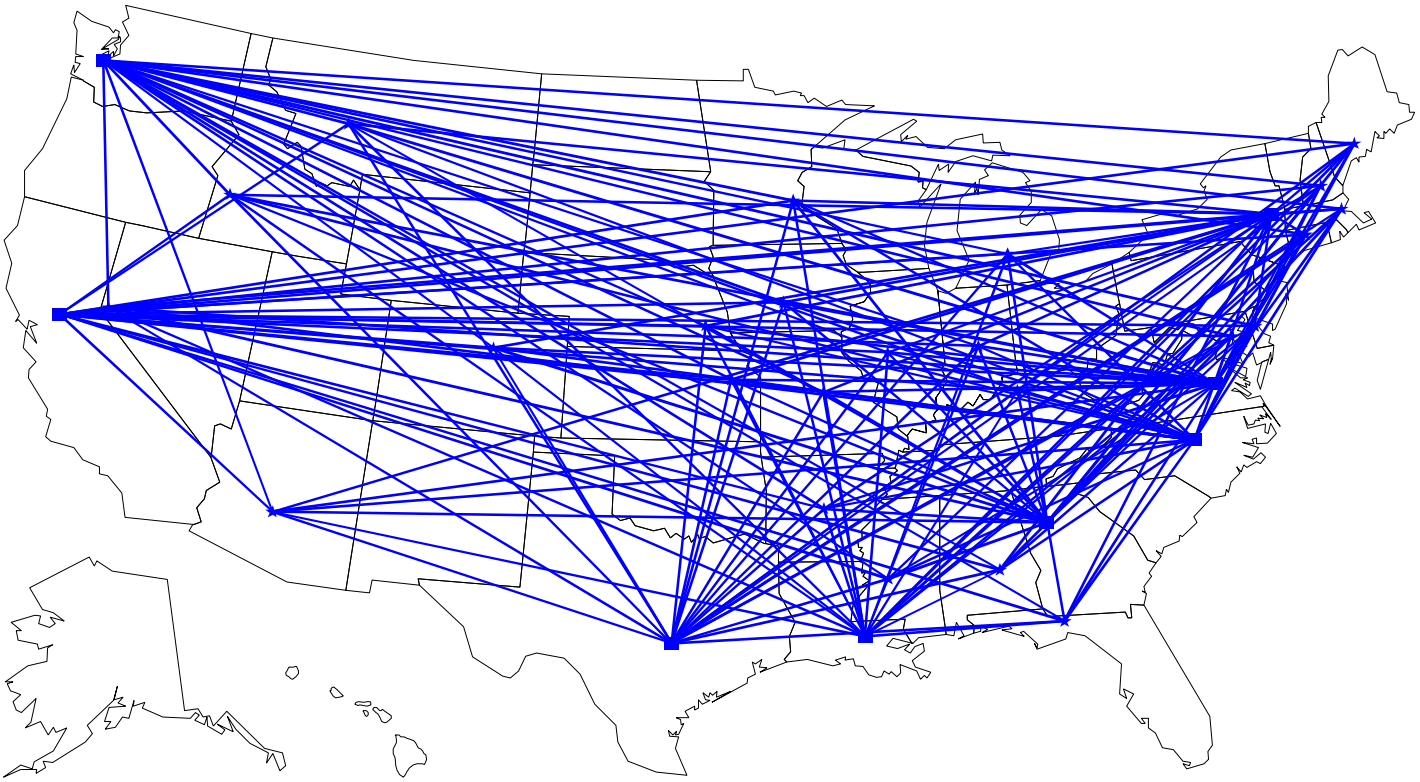
complete cover problem (crit dist = 999 mi)



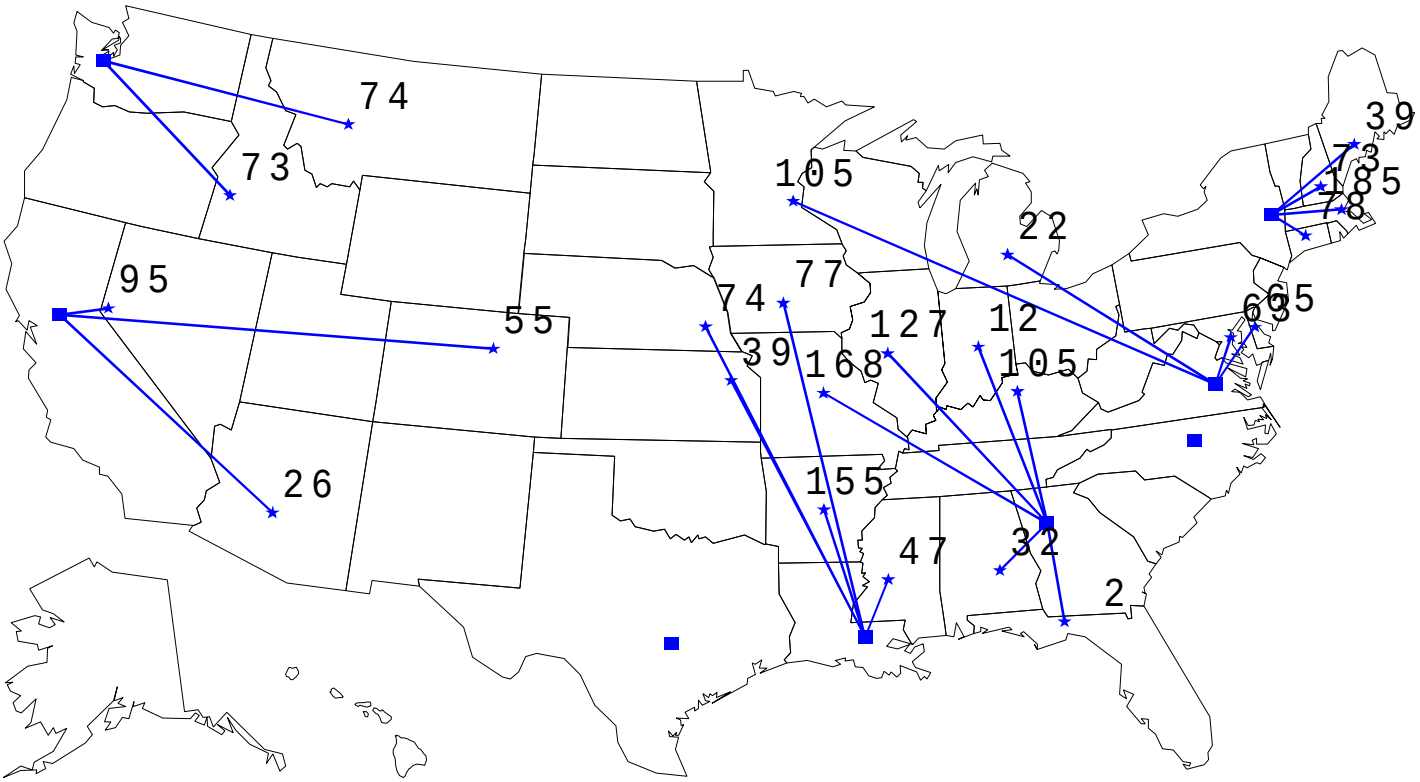
complete cover solution – five customers covered twice



ARNG median placement problem



ARNG median placement solution



PROC LP, INPUT TABLE, DENSE FORMAT

row	CA	GA	LA	NC	NY	TX	VA	WA	type	rhs
OBJ	1	1	1	1	1	1	1	1	MIN	.
BIN	1	1	1	1	1	1	1	1	BIN	.
AL	0	1	1	1	1	1	1	0	GE	1
AR	0	1	1	1	0	1	1	0	GE	1
AZ	1	0	0	0	0	1	0	0	GE	1
CO	1	0	0	0	0	1	0	0	GE	1
CT	0	1	0	1	1	0	1	0	GE	1
DE	0	1	0	1	1	0	1	0	GE	1
FL	0	1	1	1	0	1	1	0	GE	1
IA	0	1	1	1	0	1	1	0	GE	1
ID	1	0	0	0	0	0	0	1	GE	1
IL	0	1	1	1	1	1	1	0	GE	1
IN	0	1	1	1	1	1	1	0	GE	1
KS	0	1	1	1	0	1	1	0	GE	1
KY	0	1	1	1	1	1	1	0	GE	1
MA	0	1	0	1	1	0	1	0	GE	1
MD	0	1	0	1	1	0	1	0	GE	1
ME	0	0	0	1	1	0	1	0	GE	1
MI	0	1	1	1	1	0	1	0	GE	1
MN	0	1	0	1	1	0	1	0	GE	1
MO	0	1	1	1	0	1	1	0	GE	1
MS	0	1	1	1	0	1	1	0	GE	1
MT	1	0	0	0	0	0	0	1	GE	1
NE	0	1	1	0	0	1	0	0	GE	1
NH	0	1	0	1	1	0	1	0	GE	1
NV	1	0	0	0	0	0	0	1	GE	1

PROC LP, INPUT TABLE, SPARSE FORMAT

ROW	COL	TYPE	COEF	ROW	COL	TYPE	COEF	ROW	COL	TYPE	COEF	ROW	COL	TYPE	COEF
AL	GA	GE	1	DE	VA	GE	1	KS	VA	GE	1	MO	LA	GE	1
AL	LA	GE	1	DE	_RHS_	GE	1	KS	_RHS_	GE	1	MO	NC	GE	1
AL	NC	GE	1	FL	GA	GE	1	KY	GA	GE	1	MO	TX	GE	1
AL	NY	GE	1	FL	LA	GE	1	KY	LA	GE	1	MO	VA	GE	1
AL	TX	GE	1	FL	NC	GE	1	KY	NC	GE	1	MO	_RHS_	GE	1
AL	VA	GE	1	FL	TX	GE	1	KY	NY	GE	1	MS	GA	GE	1
AL	_RHS_	GE	1	FL	VA	GE	1	KY	TX	GE	1	MS	LA	GE	1
AR	GA	GE	1	FL	_RHS_	GE	1	KY	VA	GE	1	MS	NC	GE	1
AR	LA	GE	1	IA	GA	GE	1	KY	_RHS_	GE	1	MS	TX	GE	1
AR	NC	GE	1	IA	LA	GE	1	MA	GA	GE	1	MS	VA	GE	1
AR	TX	GE	1	IA	NC	GE	1	MA	NC	GE	1	MS	_RHS_	GE	1
AR	VA	GE	1	IA	TX	GE	1	MA	NY	GE	1	MT	CA	GE	1
AR	_RHS_	GE	1	IA	VA	GE	1	MA	VA	GE	1	MT	WA	GE	1
AZ	CA	GE	1	IA	_RHS_	GE	1	MA	_RHS_	GE	1	MT	_RHS_	GE	1
AZ	TX	GE	1	ID	CA	GE	1	MD	GA	GE	1	NE	GA	GE	1
AZ	_RHS_	GE	1	ID	WA	GE	1	MD	NC	GE	1	NE	LA	GE	1
BIN	CA	BINA	1	ID	_RHS_	GE	1	MD	NY	GE	1	NE	TX	GE	1
BIN	GA	BINA	1	IL	GA	GE	1	MD	VA	GE	1	NE	_RHS_	GE	1
BIN	LA	BINA	1	IL	LA	GE	1	MD	_RHS_	GE	1	NH	GA	GE	1
BIN	NC	BINA	1	IL	NC	GE	1	ME	NC	GE	1	NH	NC	GE	1
BIN	NY	BINA	1	IL	NY	GE	1	ME	NY	GE	1	NH	NY	GE	1
BIN	TX	BINA	1	IL	TX	GE	1	ME	VA	GE	1	NH	VA	GE	1
BIN	VA	BINA	1	IL	VA	GE	1	ME	_RHS_	GE	1	NH	_RHS_	GE	1
BIN	WA	BINA	1	IL	_RHS_	GE	1	MI	GA	GE	1	NV	CA	GE	1
CO	CA	GE	1	IN	GA	GE	1	MI	LA	GE	1	NV	WA	GE	1
CO	TX	GE	1	IN	LA	GE	1	MI	NC	GE	1	NV	_RHS_	GE	1
CO	_RHS_	GE	1	IN	NC	GE	1	MI	NY	GE	1	OBJ	CA	MIN	1
CT	GA	GE	1	IN	NY	GE	1	MI	VA	GE	1	OBJ	GA	MIN	1
CT	NC	GE	1	IN	TX	GE	1	MI	_RHS_	GE	1	OBJ	LA	MIN	1
CT	NY	GE	1	IN	VA	GE	1	MN	GA	GE	1	OBJ	NC	MIN	1
CT	VA	GE	1	IN	_RHS_	GE	1	MN	NC	GE	1	OBJ	NY	MIN	1
CT	_RHS_	GE	1	KS	GA	GE	1	MN	NY	GE	1	OBJ	TX	MIN	1
DE	GA	GE	1	KS	LA	GE	1	MN	VA	GE	1	OBJ	VA	MIN	1
DE	NC	GE	1	KS	NC	GE	1	MN	_RHS_	GE	1	OBJ	WA	MIN	1
DE	NY	GE	1	KS	TX	GE	1	MO	GA	GE	1				

A Confidence Interval Approach to Gene Chip Analysis

Jennifer L. Waller, Medical College of Georgia, Augusta, GA
Mark G. Anderson, Medical College of Georgia, Augusta, GA

ABSTRACT

Recently, gene chip technology and data generated from this technology have given statisticians a new realm in which to develop statistical methodology. In gene chip analysis, mRNA from a tissue sample for a disease of interest is isolated and placed on the chip. From expression data generated from the chip, we then determine what genes are highly expressed for the particular disease of interest. The challenge is to find approximately 100 genes from 12,000 that warrant further investigation. To further complicate matters, gene chips are very expensive (~$1000 each) and sample sizes tend to be very small, with $n \leq 5$ chips typically used.

Using SAS 8.0 ® under Windows 2000 © and programmed mostly within a DATA step, a confidence interval approach was developed that examines various thresholds for gene expression. mRNA from 8 normal and 8 apolipoprotine AI (apo AI, a gene known to play a pivotal role in HDL metabolism) knocked out mice were isolated and 16 gene chips were analyzed. Expression ratios (red/green) for each gene were determined within a gene chip (mouse) and 99% confidence intervals calculated for the mean expression ratio across the 8 mice within a treatment group (normal and apo AI). The lower limit of each confidence interval is compared to a series of expression ratio thresholds and those genes whose lower limit is greater than the threshold are flagged in both groups. The total number of genes flagged for each threshold is calculated for each group and plotted using SAS/GRAPH ®. Additionally, the genes flagged are compared between groups to determine how the genes are differentially expressed.

INTRODUCTION

cDNA microarrays, or gene chips, is a technology that enables a researcher to examine gene expression levels for thousands of genes simultaneously. The goal with most microarray experiments is to identify a number of genes that are differentially expressed in a diseased sample versus a referent sample. Gene identification is not the end of the investigation, however, but it is just the tip of the iceberg. From the identification of a subset of genes, a microbiologist, cellular biologist, physiologist or other basic science researcher can then further explore the functions of the subset of genes rather than having to sift through thousands. Thus the goal is to narrow down the possibilities, so that the more intensive and time consuming proteomics work can begin.

While opening up doors for clinical research, this microarray technology has also introduced a host of analytic opportunities for statisticians. This paper describes a relatively simple confidence interval approach to determining specific genes that are over or under expressed when there are two groups with replicate gene chips. This paper will give a brief overview of microarrays, describe the microarray data used, describe the confidence interval approach, and finally give further analytic issues which are currently being investigated.

OVERVIEW OF CDNA MICROARRAYS

Wildsmith and Elcock (2001), Hedenfalk et al. (2001), and Hamadeh and Afshari (2000) give good descriptions of microarrays and microarray processing. There are essentially two types of arrays, microarrays and oligonucleotide arrays. Microarrays are created by depositing a large number of genes onto a glass slide. cDNA clones are spotted in an array arrangement onto a glass slide or nylon membrane using a robotic spotting printer. mRNA from a reference sample and from a disease sample are reverse transcribed with different fluorescent dyes, green and red. Usually, green is used for the referent sample and red is used for the disease sample.

Following the reverse transcriptions with the dyes, the mRNA samples are hybridized to a cDNA microarray containing the robotically printed cDNA clones. The microarrays are then washed several times to remove unbound mRNA. Following the washes, the microarrays are scanned with a laser scanning microscope to obtain color images of the hybridization mRNA from the diseased and referent cells. Two images are then sent to a computer which generates the location and intensity of spots, one image for the green and one image for the red. These images are then overlaid. Genes which are over expressed in the diseased sample appear as red dots on the microarray image, those which have decreased expression appear as green dots, and those which are not differentially expressed in either the diseased sample or the referent sample appear as yellow dots. Because of the limitation of analyzing an image, each spot is assigned an expression value for red and an expression value for green. These two values are then used to create a red to green expression ratio and this ratio is used in analyses.

APOLIPOPROTIEN AI AND CONTROL GROUP MICROARRAY DATA

Callow et al. (2000) studied two lines of mice with very low HDL cholesterol levels compared to inbred control mice. Data for this paper consisted on one of the two studied mouse lines, theapolipoprotien AI (or apo AI) knock-out line. Of interest in this study was to determine genes with differential expression for low HDL. Tissue samples were taken from the livers of the mice.

The treatment group consisted of 8 mice with the apo AI gene knocked out. The control group consisted of 8 control C57B1/6 mice. Target cDNA was obtained from mRNA by reverse transcription and labeled using red fluorescent dye, Cy5 from each of the 16 mice. The referent sample cDNA was prepared by combining the cDNA from the 8 control mice and was labeled using green fluorescent dye, Cy3. Each microarray consisted of the cDNA for one mouse (i.e. the “diseased” sample representing the red dye) and the combined cDNA of the 8 control mice (i.e. the referent sample representing the green dye).

The cDNA was hybridized to a microarray that contained 5,548 cDNA probes, including 200 probes that were related to lipid metabolism. The microarrys were then imaged and red and green fluorescence intensities were generated for each cDNA probe.

THE IMPORTANCE OF REPLICATION

Due to the cost (approximately $1,000) and the vastness of the data that is produced by a single microarray, many researchers choose to perform only a single replicate or to pool data from several diseased samples and several referent samples and use these pooled data on a single chip. While cost effective, this is not an ideal approach. Lee et al. (2000) examined the variability between replicated microarrays and showed that a single microarray experiment has substantial variability from a variety of sources. In their paper, three replicates were used and one replicate gave different results than the other two indicating that “replication does not ensure duplication of results, a fact that cannot be quantified when replication is not used.” (Lee et al., 2000). Thus, when working with researchers on microarray data it is necessary to educate them about replication and how replication can help them get a handle on the different sources of variability.

CONFIDENCE INTERVAL APPROACH TO GENE IDENTIFICATION

The confidence interval approach taken to examine genes and determine which are differentially expressed in the diseased samples versus the control samples is a relatively simple one. The first step in construction of a confidence interval is to examine the skewness of the ratio statistic, R/G, calculated for each replicate within each gene. Transformations of the ratio statistic may be necessary before the confidence intervals are constructed. In this case, a family of transformations was examined and the transformation that produced the lowest mean skewness across genes was used. The family of transformations (Box and Tiao, 1992, pg. 530) used was

$$y^{(\lambda)} = \begin{cases} \dfrac{y^{\lambda}-1}{\lambda} & (\lambda \neq 0) \\ \log(y) & (\lambda = 0) \end{cases} \qquad y > 0$$

where y=R/G and $-2 \leq \lambda \leq 2$. After examining the skewness of the transformed ratio statistic, it was determined that the log(R/G) was the least skewed. Because many researchers are interested in at least 2-fold increases, $\log_2(R_r/G_r)$, r=the replicate, was used. SAS code for examining the skewness of different transformations is included with comments indicating code for various steps.

For each gene, a 100(1-α)% confidence interval for the mean ratio of R/G using the number of replicates within a group was calculated by the following:

$$\bar{x}_g \pm t_{(r-1,1-\alpha/2)} se(\bar{x}_g)$$

where g is the group, diseased or control, r is the number of replicates in each group, and $\bar{x}$ is calculated as the mean of $\log_2(R_r/G_r)$.

After calculating the confidence interval for each group, diseased or control, the lower and upper limits are compared to various expression ratio thresholds. If the lower limit is greater than the threshold, then the gene is considered to be over expressed for that particular group. If the upper limit is lower than the inverse of the threshold then the gene is considered to be under expressed for that particular group.

While knowing that a gene is differentially expressed within the diseased group or the control group of interest, it is also necessary to determine whether the gene was differentially expressed in both groups, only in the diseased group, only in the control group, or in neither group. This additional information is helpful is assessing which genes to concentrate further laboratory research on in the future. Newton et al. (2000) indicate that the ratio of the expression genes within a microarray vary greatly, i.e. some may have high expression for any sample and some may have very low expression. Thus knowing whether the gene is expressed in both, neither, or only in one group is necessary.

SAS code is given for the calculation of 99% confidence intervals and commonality of expression with comments indicating the steps taken.

RESULTS

The confidence interval method identified 7 genes which were under expressed in the diseased group of microarrays. Dudoit and Yang et al. (2000) used the same apo AI data in the investigation of their methodology. For the apo AI experiments they identified 8 genes which were differentially expressed in the apo AI knocked out mice as compared to the control mice. The confidence interval method identified 7 genes which were under expressed at the 2-fold level, six of which were the same genes identified by Dudoit and Yang. The different results between the CI approach and that taken by Dudoit and Yang is probably due to the different alpha levels used, here 0.01 and Dudoit and Yang 0.05.

The output of the number of genes identified at each threshold are given, the commonality of the genes expressed at each threshold in the apo AI mice and the control mice, and the plots of differentially (either over or under) expressed genes identified at the 2-fold level are shown following the SAS code. Output is shown only when genes were identified as over or under expressed.

FUTURE INVESTIGATIONS

Newton et al. (2001) indicate that when considering fold changes (e.g. 1.5-fold or 2-fold) in gene expression utilizing a ratio measure ignores the variation of the ratio across genes. In essences their argument is that a 1.5-fold change for one gene may be perfectly adequate to show that the specific gene is differentially expressed, but that a 3-fold change in another gene may be required to show differential expression. While the methodology he describes deals with a single microarray, he indicates that combining the information from replicates may aid in identifying the contribution of different sources of variation. Thus one possibility to investigate is the variability between genes in their expression and consider using different ratio threshold levels for each gene based on their individual distribution.

Second, Yang and Dudoit et al. (2000) and Chen et al. (1997) describe different normalization techniques for cDNA microarray data. These include within slide location normalizations (global, intensity dependent, and within print-tip-group), within slide scale normalizations, paired slide dye-swap normalizations, and multiple slide normalizations. Implementing these various normalizations within SAS before analysis is another area for further investigation.

Third, the number of confidence intervals which are generated is extremely large and the potential of misidentification of differentially expressed genes is large. Dudoit and Yang et al. (2000) investigated a two-sample t-statistic for determining differential expression, performed permutation tests on this test statistic, and calculated adjusted p-values using a variety of different methods. Implementing Dudoit and Yang et al.'s methodology in SAS is currently underway.

CONCLUSION

The relatively simple confidence interval method presented here provides similar results to others who have utilized the data from the apo AI experiments conducted by Callow et al. (2000). Improvements in normalization and implementation of permutation tests within SAS will provide additional tools for the analysis of microarray data.

Analysis of microarray data is of great interest and new methodologies and utilization of and improvements in old methodologies emerge every day. Additionally, many new software programs are currently on the market which aid a researcher in analyzing microarray data using both old and new methodologies. However, researchers must be aware of the pitfalls of just pointing and clicking without understanding their data and the methods they have used for analysis in these new programs. Rather than investing in new software, which can be quite costly, investing the time to learn the new package and verifying that the software is performing an analysis as it should, implementing current and new methodologies and investigating the capabilities in SAS is desirable. SAS is a powerful tool, we just need to educate the people we work with as to how powerful it is!

REFERENCES

Box, GEP, Tiao, GC. (1992) Bayesian Inference in Statistical Analysis. New York, NY: John Wiley and Sons.

Callow, MJ, Dudoit, S, Gong, EL, Speed TP, and Rubin EM. Microarray expression profiling identifies genes iwht altered expression in hdl deficient mice. Genome Research 10(12):2022-9, 2000.

Chen, Y, Dougherty, ER, Bittner, ML. Ratio-based decision and the quantitative analysis of cDNA microarray images. Journal of Biomedical Optics 2:364-374, 1997.

Dudoit, S, Yang, YH, Callow, MJ, Speed, TP. Statistical methods for identifying differentially expressed genes in replicated cDNA microarray experiments. (Technical report #578) Submitted, Journal of the American Statistical Association, 2000.

Hamadeh, H, and Afshari, CA. Gene chips and functional genomics. American Scientist 88:508-515.

Hedenfalk, I, Duggan, D, Chen, Y, Radmacher, M, Bittner, M, Simon, R, Meltzer, P, Busterson, B, Esteller, M, Kallioniemi, OP, Wilfond, B, Borg, A, Trent J. Gene expression profiles in hereditary breast cancer. The New England Journal of Medicine 344(8):539-548, 2001.

Lee, MLT, Kuo, FC, Whitmore, GA, Sklar, J. Importance of replication in microarray gene expression studies: Statistical methods and evidence from repetitive cDNA hybridizations. Proceedings of the National Academy of Science, 97(18):9834-9839, 2000.

Newton, MA, Kendziorski, CM, Richmond, CS, Blattner, FR, Tsui, KW. On differential variability of expression ratios: Improving statistical inference about gene expression changes from microarray data. Journal of Computational Biology 8(1): 37-52, 2001.

Wildsmith, SE, Elcock, FJ. Microarrays under the microscope. Molecular Pathology, 54(1):8-16, 2001.

Yang, YH, Dudoit, S, Luu, P, Speed, T. Normalization for cDNA microarray data. Technical Report #589, submitted, 2001.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Jennifer L. Waller
Medical College of Georgia
Office of Biostatistics and Bioinformatics (AE-3031)
Augusta, GA 30912-4900
Work Phone: (706) 721-3785
Fax: (706) 721-6294
Email: jennifer@stat.mcg.edu

TRANSFORMATION SAS PROGRAM

```
****************************************
** This program examines different      **
** transformations of R/G to determine  **
** which has the lowest mean skewness   **
** across genes.                        **
****************************************;
data apoa1;
  set in.apoa1;
  ****************************************
  ** Creates 8 observations per gene. Creates**
  ** ratio of R/G for each gene for each     **
  ** group, disease (t) and control (c).     **
  ****************************************;
  array cg {8} c1g c2g c3g c4g c5g c6g c7g c8g;
  array cr {8} c1r c2r c3r c4r c5r c6r c7r c8r;
  array tg {8} k1g k2g k3g k4g k5g k6g k7g k8g;
  array tr {8} k1r k2r k3r k4r k5r k6r k7r k8r;
  do i=1 to 8;
    gene=genenum;
        cgexp=cg{i};
        crexp=cr{i};
        tgexp=tg{i};
        trexp=tr{i};
        crg=cr{i}/cg{i};
        trg=tr{i}/tg{i};
        output;
  end;
  keep gene cgexp crexp tgexp trexp crg trg;
  ************************************
  ** Create transformations with lambda **
  ** in the interval -2 to 2, by 0.5.   **
  ************************************;
data new;
  set apoa1;
  array cln {4} cl_n05 cl_n1 cl_n15 cl_n2;
  array cl  {4} cl_05 cl_1 cl_15 cl_2;
  array tln {4} tl_n05 tl_n1 tl_n15 tl_n2;
  array tl  {4} tl_05 tl_1 tl_15 tl_2;
  do i=1 to 4;
    cln{i}=((crg)**(-(i/2))-1)/(-i/2);
    cl{i}=((crg)**((i/2))-1)/(i/2);
    tln{i}=((trg)**(-(i/2))-1)/(-i/2);
    tl{i}=((trg)**((i/2))-1)/(i/2);
  end;
  cl_0=log(crg);
  tl_0=log(trg);

proc sort data=new; by gene;
****************************************
** Calcualte skewness measure for the ratio**
** and transformations of R/G in each group**
** for each gene.                       **
****************************************;
proc means data=new skew noprint;
  var trg tl_n2 tl_n15 tl_n1 tl_n05 tl_0 tl_05
     tl_1 tl_15 tl_2 crg cl_n2 cl_n15 cl_n1
     cl_n05 cl_0 cl_05 cl_1 cl_15 cl_2;
  output out=vs skew=strg stl_n2 stl_n15
      stl_n1 stl_n05 stl_0 stl_05 stl_1
      stl_15 stl_2 scrg scl_n2 scl_n15
      scl_n1 scl_n05 scl_0 scl_05 scl_1
     scl_15 scl_2;
  by gene;
****************************************
** Calculate the mean skewness of R/G   **
** transformations across genes to determine **
** which gives a mean skewness nearest zero. **
****************************************;
proc means data=vs n mean std median;
  var strg stl_n2 stl_n15 stl_n1 stl_n05
      stl_0 stl_05 stl_1 stl_15 stl_2;
title 'Diseased Group Mean Skewness Across
       Genes';

proc means data=vs n mean std median;
  var scrg scl_n2 scl_n15 scl_n1 scl_n05
      scl_0 scl_05 scl_1 scl_15 scl_2;
title 'Control Group Mean Skewness Across
Genes';
run;
```

CONFIDENCE INTERVAL SAS PROGRAM

```
*********************************************
** This program calculates a CIfor each gene  **
** within each group, disease (t) or control  **
** (c ).  The lower and upper limits are      **
** compared to  different thresholds to       **
** determine the number of differentially     **
** expressed genes in each group. A           **
** commonality variable is created to show    **
** whether the gene expressed in both, only   **
** one, or neither the disease or control     **
** group. Finally graphs are produced which   **
** show the lower limit, mean, and upper limit**
** of the log2(R/G) expression for each group.**
*********************************************;
proc format;
  value oexp 1='Over Expressed'
             0='Not Expressed';
  value uexp 1='Under Expressed'
             0='Not Expressed';
  value common 1='Both Exp'
               2='Cntl Exp'
               3='Trt Exp'
               4='Neither Exp';

data apoa1;
****************************************
** Create ratios of log2(r/g) for each  **
** sample. Also create                  **
** log2(r/g)_trt/log2(r/g)_ctl.         **
****************************************;
  array cg c1g c2g c3g c4g c5g c6g c7g c8g;
  array cr c1r c2r c3r c4r c5r c6r c7r c8r;
  array tg k1g k2g k3g k4g k5g k6g k7g k8g;
  array tr k1r k2r k3r k4r k5r k6r k7r k8r;
  array cratio cratio1-cratio8;
  array tratio tratio1-tratio8;
  do over cg;
    cratio=log2(cr/cg);
    tratio=log2(tr/tg);
  end;
****************************************
** Calculate the mean, standard error   **
** and effective sample size across     **
** samples within the groups.           **
****************************************;
  /* Normal - Control Group */
  m_cratio=mean(of cratio1-cratio8);
  se_cratio=stderr(of cratio1-cratio8);
  n_cratio=n(of cratio1-cratio8);
  /* Disease - Treatment Group */
  m_tratio=mean(of tratio1-tratio8);
  se_tratio=stderr(of tratio1-tratio8);
  n_tratio=n(of tratio1-tratio8);
****************************************
** Set alpha, degrees of freedom, and   **
** reliability coefficient for each     **
** group.                               **
****************************************;
  alpha=0.995;
  cdf=n(of cratio1-cratio8)-1;
  tdf=n(of tratio1-tratio8)-1;
  ctcoeff=tinv(alpha,cdf);
  ttcoeff=tinv(alpha,tdf);
****************************************
** Calculate upper and lower confidence **
** intervals   for each group.          **
****************************************;
  cucl=m_cratio+ctcoeff*se_cratio;
  clcl=m_cratio-ctcoeff*se_cratio;
  tucl=m_tratio+ttcoeff*se_tratio;
  tlcl=m_tratio-ttcoeff*se_tratio;
****************************************
** Determine if lower confidence limit  **
** is greater than a threshold value.   **
** Determine if upper confidence limit  **
** is less than a threshold value.      **
** Determine the commonality of         **
** expression.                          **
****************************************;
  /* R greater than G */
  array cthresh {9} cthresh1 cthresh1_25
                    cthresh1_5 cthresh1_75
                    cthresh2 cthresh2_25
                    cthresh2_5 cthresh2_75
                    cthresh3;
  array tthresh {9} tthresh1 tthresh1_25
                    tthresh1_5 tthresh1_75
                    tthresh2 tthresh2_25
                    tthresh2_5 tthresh2_75
                    tthresh3;
  array commona {9} common1 common1_25
                    common1_5 common1_75
                    common2 common2_25
                    common2_5 common2_75
                    common3;
  do k=1 to 9;
    cthresh{k}=0;
    tthresh{k}=0;
    if clcl>((k+1)-((3*k)/4)) then cthresh{k}=1;
    if tlcl>((k+1)-((3*k)/4)) then tthresh{k}=1;
    if cthresh{k}=tthresh{k}=1 then
      commona{k}=1;
      else if cthresh{k}=1 and tthresh{k}=0 then
        commona{k}=2;
      else if cthresh{k}=0 and tthresh{k}=1 then
        commona{k}=3;
      else if cthresh{k}=tthresh{k}=0 then
        commona{k}=4;
  end;
  /* G greater than R */
  array cthresha {9} cthreshn1 cthreshn1_25
                     cthreshn1_5 cthreshn1_75
                     cthreshn2 cthreshn2_25
                     cthreshn2_5 cthreshn2_75
                     cthreshn3;
  array tthresha {9} tthreshn1 tthreshn1_25
                     tthreshn1_5 tthreshn1_75
                     tthreshn2 tthreshn2_25
                     tthreshn2_5 tthreshn2_75
                     tthreshn3;
  array commonaa {9} commonn1 commonn1_25
                     commonn1_5 commonn1_75
                     commonn2 commonn2_25
                     commonn2_5 commonn2_75
                     commonn3;
  do j=1 to 9;
    cthresha{j}=0;
    tthresha{j}=0;
    if cucl<-((j+1)-((3*j)/4)) then
      cthresha{j}=1;
    if tucl<-((j+1)-((3*j)/4)) then
      tthresha{j}=1;
    if cthresha{j}=tthresha{j}=1 then
      commonaa{j}=1;
      else if cthresha{j}=1 and tthresha{j}=0
        then commonaa{j}=2;
      else if cthresha{j}=0 and tthresha{j}=1
        then commonaa{j}=3;
      else if cthresha{j}=tthresha{j}=0 then
        commonaa{j}=4;
     end;
```

```
*******************************************
** Format threshold variables and common   **
** gene variables.                         **
*******************************************;
  format cthresh1 cthresh1_25 cthresh1_5
         cthresh1_75 cthresh2 cthresh2_25
         cthresh2_5 cthresh2_75 cthresh3
         tthresh1 tthresh1_25 tthresh1_5
         tthresh1_75 tthresh2 tthresh2_25
         tthresh2_5 tthresh2_75 tthresh3 oexp.
         cthreshn1 cthreshn1_25 cthreshn1_5
         cthreshn1_75 cthreshn2 cthreshn2_25
         cthreshn2_5 cthreshn2_75 cthreshn3
         tthreshn1 tthreshn1_25 tthreshn1_5
         tthreshn1_75 tthreshn2 tthreshn2_25
         tthreshn2_5 tthreshn2_75 tthreshn3
         uexp.
         common1 common1_25 common1_5 common1_75
         common2 common2_25 common2_5 common2_75
         common3 commonn1 commonn1_25 commonn1_5
         commonn1_75 commonn2 commonn2_25
         commonn2_5 commonn2_75 commonn3
         common.;
*******************************************
** Get frequency distribution of threshold   **
** variables for control and disease groups. **
*******************************************;
proc freq data=apoa1;
  tables cthresh1 cthresh1_25 cthresh1_5
         cthresh1_75 cthresh2 cthresh2_25
         cthresh2_5 cthresh2_75 cthresh3;
title 'Number of Genes with Lower CL';
title2 'Above the Threshold for Control Group';

proc freq data=apoa1;
  tables cthreshn1 cthreshn1_25
         cthreshn1_5 cthreshn1_75
         cthreshn2 cthreshn2_25
         cthreshn2_5 cthreshn2_75
         cthreshn3;
title 'Number of Genes with Upper CL Below';
title2 '1/Threshold for Control Group';

proc freq data=apoa1;
  tables tthresh1 tthresh1_25 tthresh1_5
         tthresh1_75 tthresh2 tthresh2_25
         tthresh2_5 tthresh2_75 tthresh3;
title 'Number of Genes with Lower CL';
title2 'Above the Threshold for Disease Group';

proc freq data=apoa1;
  tables tthreshn1 tthreshn1_25
         tthreshn1_5 tthreshn1_75
         tthreshn2 tthreshn2_25
         tthreshn2_5 tthreshn2_75
         tthreshn3;
title 'Number of Genes with Upper CL Below';
title2 '1/Threshold for Disease Group';

proc freq data=apoa1;
  tables common1 common1_25 common1_5
         common1_75 common2 common2_25
         common2_5 common2_75 common3
         commonn1 commonn1_25 commonn1_5
         commonn1_75 commonn2 commonn2_25
         commonn2_5 commonn2_75 commonn3;
title 'Number of Common Expressions Control vs
       Disease';

*****************************************
** Plot genes with lower limit greater  **
** than 2-fold expression ratio or upper**
** limit less than ½-fold expression.   **
*****************************************;
goptions reset=(axis, legend, pattern,
                symbol, title, footnote)
         norotate hpos=0 vpos=0 htext=
         ftext= ctext= target= gaccess=
         gsfmode= ;
goptions device=WIN  ctext=blue graphrc
         interpol=join;

symbol1 c=DEFAULT i=none ci=black v=dot;
symbol2 c=DEFAULT i=none ci=black v=x;
symbol3 c=DEFAULT i=none ci=black v=dot;

axis1 order=(-0 to 6 by 0.50) color=blue
      width=2.0 offset=(1 cm)
      label=('Log 2(LCL) Threshold');
axis2 color=blue width=2.0 offset=(1 cm)
      label=('Gene Number');
axis3 order=(-6 to 0 by 0.50) color=blue
      width=2.0 offset=(1 cm)
      label=('Log 2(UCL) Threshold');

proc gplot data=apoa1;
  plot genenum*clcl
       genenum*m_cratio
       genenum*cucl / overlay haxis=axis1
                      vaxis=axis2 frame;
  where cthresh1=1;
  title 'Control Expressions with LCL > 1
         - 2-fold';

  proc gplot data=apoa1;
  plot genenum*tlcl
       genenum*m_tratio
       genenum*tucl / overlay haxis=axis1
                      vaxis=axis2 frame;
  where tthresh1=1;
  title 'Disease Expressions with LCL > 1
         - 2 fold';

proc gplot data=apoa1;
  plot genenum*clcl
       genenum*m_cratio
       genenum*cucl / overlay haxis=axis3
                      vaxis=axis2 frame;
  where cthreshn1=1;
  title 'Control Expressions with UCL < -1
         - ½-fold';

proc gplot data=apoa1;
  plot genenum*tlcl
       genenum*m_tratio
       genenum*tucl / overlay haxis=axis3
                      vaxis=axis2 frame;
  where tthreshn1=1;
  title 'Treatment Expressions with UCL < -1
         - ½-fold';

run;
quit;
run;
```

Number of Genes with Upper CL Below
1/Threshold for Disease Group

The FREQ Procedure

tthreshn1	Frequency	Percent
Not Expressed	6377	99.89
Under Expressed	7	0.11

tthreshn1_25	Frequency	Percent
Not Expressed	6379	99.92
Under Expressed	5	0.08

tthreshn1_5	Frequency	Percent
Not Expressed	6381	99.95
Under Expressed	3	0.05

tthreshn1_75	Frequency	Percent
Not Expressed	6382	99.97
Under Expressed	2	0.03

tthreshn2	Frequency	Percent
Not Expressed	6382	99.97
Under Expressed	2	0.03

tthreshn2_25	Frequency	Percent
Not Expressed	6383	99.98
Under Expressed	1	0.02

tthreshn2_5	Frequency	Percent
Not Expressed	6383	99.98
Under Expressed	1	0.02

tthreshn2_75	Frequency	Percent
Not Expressed	6383	99.98
Under Expressed	1	0.02

tthreshn3	Frequency	Percent
Not Expressed	6383	99.98
Under Expressed	1	0.02

Number of Common Expressions Control vs Disease

commonn1	Frequency	Percent
Trt Exp	7	0.11
Neither Exp	6377	99.89

commonn1_25	Frequency	Percent
Trt Exp	5	0.08
Neither Exp	6379	99.92

commonn1_5	Frequency	Percent
Trt Exp	3	0.05
Neither Exp	6381	99.95

commonn1_75	Frequency	Percent
Trt Exp	2	0.03
Neither Exp	6382	99.97

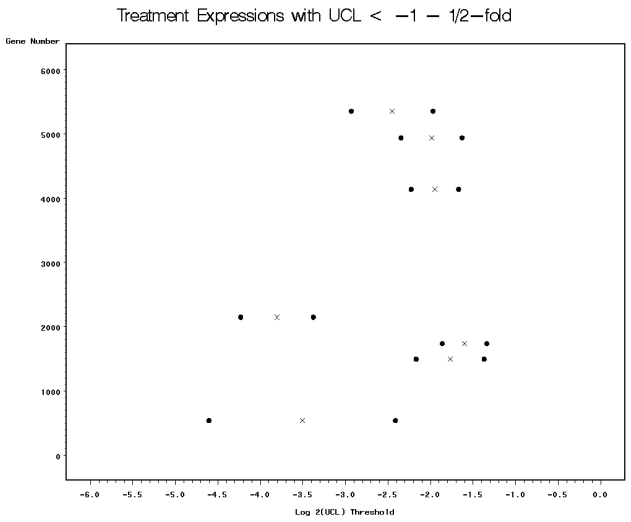
commonn2	Frequency	Percent
Trt Exp	2	0.03
Neither Exp	6382	99.97

commonn2_25	Frequency	Percent
Trt Exp	1	0.02
Neither Exp	6383	99.98

commonn2_5	Frequency	Percent
Trt Exp	1	0.02
Neither Exp	6383	99.98

commonn2_75	Frequency	Percent
Trt Exp	1	0.02
Neither Exp	6383	99.98

commonn3	Frequency	Percent
Trt Exp	1	0.02
Neither Exp	6383	99.98



The Output Delivery System for Data Analysis

Randy Tobias , SAS

Abstract: Beginning with Version 7 of SAS software, all procedures use the Output Delivery System (ODS) to produce their results. ODS gives you tremendous control over the appearance of your results. Using ODS, you can create SAS output data sets from every table in the output, rearrange columns, select or exclude individual pieces of output, and render output in HTML, Rich-Text, and other formats. In addition, you can control the appearance of your procedure output by changing column formats and headers and specifying colors and fonts. This workshop discusses the Output Delivery System and illustrates its use through a series of examples using SAS/STAT procedures. The audience should be familiar with SAS software.

Customizing Statistical Reports Using ODS and Proc Template

Joy Munk Smith , North Carolina State University
Sandy Donaghy, North Caroline State University

Abstract: The Output Delivery System (ODS) provides nearly limitless flexibility in formatting statistical output. All statistical output can be stored in SAS data sets and templates created with Proc Template can be used to create customized reports. This tutorial will cover using ODS to creating SAS data sets from statistical procedures, and using Proc Template to create templates for displaying statistical output. The creation, storage, and use of custom templates will be covered. This material will be covered using examples and will include an online demonstration.

GETTING STARTED WITH PROC LOGISTIC

Andrew H. Karp
Sierra Information Services, Inc.
Sonoma, California USA

Introduction

Logistic Regression is an increasingly popular analytic tool. Used to predict the probability that the 'event of interest' will occur as a linear function of one (or more) continuous and/or dichotomous independent variables, this technique is implemented in the SAS® System in PROC LOGISTIC. This paper gives an overview of how some common forms of logistic regression models can be implemented using PROC LOGISTIC as well as important changes and enhancements to the procedure in Releases 6.07 and above of the SAS® System, as well as new features available in Version 8.

Background

Logistic regression is commonly used to obtain predicted probabilities that a unit of the population under analysis will acquire the event of interest as a linear function of one or more:

- continuous-level variables
- dichotomous (binary) variables
- or, a combination of both continuous and binary independent variables.

Many concepts in logistic regression will be familiar to people who already have experience with simple and multiple regression models. In fact, much of the syntax in PROC LOGISTIC will be familiar to SAS System users already experienced with using PROCs REG and/or GLM.

In logistic regression, however, the dependent variable is *dichotomous* and is usually coded as:

- zero (event did not occur)
- one (event did occur)

for each particular subject in the data set upon which the analysis will be carried out.

The *logistic function* is used to estimate, as a function of unit changes in the independent variable, the probability that the event of interest will occur. This function is often called the *link function* in that it connects, or 'links' changes in values of the independent variables to increasing (or decreasing) probability of occurrence of the event being modeled by the dependent variable.

Implementation in the SAS System

Techniques for implementing logistic regression are found in PROC LOGISTIC in the STAT module of SAS System software, and is one of several procedures in this module which can be used for categorical data analysis. Other procedures for categorical data analysis in the STAT module include:

- FREQ
- GENMOD
- CATMOD
- PROBIT
- PHREG
- LIFETEST
- PROBIT

Data Preparation

As with other forms of data analysis, the results of a logistic regression analysis performed by PROC LOGISTIC can be seriously compromised if the analyst does not take care to prepare their data properly. Following important rules regarding construction and coding of the dependent variable are critical to the SAS System's generation of accurate results.

Dependent Variable

Your dependent variable should be:

- dichotomous
- coded zero for 'non-event'
- coded one for 'event'

Although *polytomous (or multinomial logistic) regression models* (those with three or more levels, or categories, of the dependent variable) can be implemented in the SAS System, discussion of these types of models, and how they are implemented in the SAS System, is beyond the scope of this paper and will not be considered here. SAS System implementation of these types of models is discussed in the following SAS Institute publications:

- *Logistic Regression Examples Using the SAS System (1995)*
- *SAS Technical Report R-109: Conjoint Analysis Examples (1993)*

Effect of Coding Dependent Variable on how PROC LOGISTIC Works

The zero/one coding scheme is the most commonly employed method by which events/non-events are classified for the purposes of conducting a logistic regression analysis. By default, however, PROC LOGISTIC will attempt to model (that is, predict the probability of) the lower of the two values of the dependent variable. which is usually not the desired result.

For example, if a researcher were attempting to determine the probability that a patient will die, the variable representing "outcome" (i.e., "dead" or "alive") might be coded zero for patients who survived and one for patients who died. Since zero "sorts lower" than one, PROC LOGISTIC will attempt to 'model' (that is, predict) the probability that the patient will be coded zero (lived) rather than the probability that the patient will be coded one (died). This is most likely the ***opposite*** of what the researcher desired.

Overriding SAS System Defaults

Users can override the default attempt by PROC LOGISTIC to predict the probability of the non-event using one of three approaches:

- re-coding the dependent variable (e.g., 0 = 'died', 1 = 'lived')
- creating and applying a FORMAT to the dependent variable where the *formatted value* of the 'event' group 'sorts higher' than the 'non-event' group (i.e., the external representation of 0 = 'Alive' and 1 = 'Dead')
- use the DESCENDING option in the PROC LOGISTIC statement.

This DESCENDING option, new in Release 6.07, is probably the easiest and most straightforward method by which to override the SAS System default, as it avoids potentially unnecessary work in a DATA Step before applying PROC LOGISTIC.

Implementing a Logistic Regression Analysis

The structure and syntax of many features in PROC LOGISTIC are similar to those used in PROCs REG and GLM, which facilitates comparison of how to perform a logistic regression analysis with linear models such as regression and analysis of variance.

The important difference, for our purposes, between what is being estimated by a logistic regression model and that estimated by a linear model is:

- *linear regression* attempts to predict the *value* of the dependent variable as a linear function of one (or more) independent variables
- *logistic regression* attempts to predict the *probability* that a unit under analysis will acquire the event of interest as a function of one or more independent variables.

Put another way, the logistic regression equation predicts the probability that the unit under analysis will, as a function of one or more independent variables, obtain the condition of interest which is (usually) coded as 1 in a zero/one coding scheme.

The general form of PROC LOGISTIC is:

```
PROC LOGISTIC DATA=dsn [DESCENDING] ;
MODEL depvar = indepvar(s)/options;
RUN;
```

Interpretation of SAS System-Generated Results

Tests of the Global Null Hypothesis

The default output generated by PROC LOGISTIC looks very similar to that generated by PROCs REG and/or GLM. This output includes several tests of overall model adequacy which test the global null hypothesis that *none* of the independent variables in the model are related to changes in probability of event occurrence. Of these, the **-2 LOG L** test is perhaps the most easy to interpret and is analogous

to the "Global F" test used in a linear regression analysis. The computation of and rationale for the **-2 LOG L** test, among others, is found in Hosmer and Lemeshow (1989). Other global tests, such as the **SCORE, Akaike Information Criterion,** and **Schwartz Bayesian Criterion** are also provided but are beyond the scope of this paper.

Tests of the Local Null Hypotheses
Tests of the 'statistical significance' of each independent variable are also provided. The **Wald Chi-Square** test (and its associated p-value) are printed along with the **parameter estimate** and **standardized parameter estimate.** As with linear regression analysis, the parameter estimate can be conceptualized as how much mathematical impact a unit changes in the value of the independent variable has on increasing or decreasing the probability that the dependent variable will achieve the value of one in the population from which the data are assumed to have been randomly sampled.

The Odds Ratio
Exponentiation of the parameter estimate(s) for the independent variable(s) in the model by the number *e* (about 2.17) yields the **odds ratio**, which is a more intuitive and easily understood way to capture the relationship between the independent and dependent variables. This quantity is automatically portrayed in PROC LOGISTIC starting in Release 6.07; users with earlier SAS System releases can easily compute this quantity by hand.

The odds ratio gives the increase or decrease in probability that a unit change in the independent variable has in the probability that the event of interest will occur. Two analytic scenarios will be presented here to further motivate this concept: a) categorical independent variable; and, b) continuous independent variable. Both examples are drawn from Hosmer and Lemeshow's (1989) study of patient survival after admission to a hospital intensive care unit (ICU).

a) categorical independent variable
A logistic regression model was implemented using 'admission type' as an independent variable. This variable was coded one if the patient was admitted to the hospital via emergency room and zero if the patient was admitted via another hospital 'service', such as surgery, cardiology, etc..

The resulting odds ratio for this model was 8.89, which suggests that a patient admitted via the emergency room is about 9 times more likely to die than a patient admitted from another service.

b) continuous level independent variable
Consider another analytic situation where the event of interest to be predicted is patient's survival after admission to a hospital intensive care unit (ICU), and the independent variable is age of patient in years. Application of PROC LOGISTIC to the Hosmer and Lemeshow data set yielded a parameter estimated for the variable AGE as 0.0275; exponentiation of that estimate gives an odds ratio of 1.028. In this example, a one unit (that is, one year) increase in a patient's age increases by 2.8 percent the chance they will die (i.e., acquire the event of interest). [Of course, while this result may be 'statistically significant', the clinical relevance to a health care provider of patient age, without regard to other prognostic factors (such as disease severity) may limit the practical usefulness of the results.]

Customized Odds Ratios
As with the previous example, a unit change in the values of the independent variable(s) may not be substantively relevant or useful to the analyst. In the ICU survival study, a five, ten, or twenty year change in patient age may be of more clinical relevance than a change of just one year. Customized odds ratios can be obtained by:

- hand, using a calculator
- placing the parameter estimates generated by PROC LOGISTIC into an output SAS data set using the OUTEST option and then working in the data step

using the **UNITS** option, which is available in Release 6.10 and above. For example:

UNITS AGE = 5 10 20 ;

Placed after the MODEL statement would generate customized odds ratios for five, ten and twenty year changes in patient age.

Confidence Intervals for Odds Ratios
As with regression analysis, the parameter estimates and associated odds ratios are *point estimates* of the true value of these quantities in the population from which the data under analysis are assumed to have been randomly sampled. Confidence intervals for the odds ratios can be obtained by:

- manual calculation

- use of the **RISKLIMITS** option, which was first available in Release 6.07

By default, the **RISKLIMITS** option produces 95% confidence intervals around the odds ratios for each independent variable in the model. Users can obtain customized confidence intervals by using the **ALPHA** option.

Multiple Logistic Regression Model

Researchers are frequently interested in examining either the joint effect of two more independent variables on the likelihood of event outcome. In other situations the effect of a single independent variable is analyzed controlling for (that is, holding constant the effect of) other independent variables. In these situations a *multiple logistic regression model* is required, and is implemented by placing the names of the independent variables of interest to the right of the equals sign in the MODEL statement.

Multiple logistic regression model results generated by PROC LOGISTIC are interpreted in much the same way as are results obtained from a multiple logistic regression model: the parameter estimates (and resulting odds ratios) are the unique effect (if any) on the probability of event occurrence as if each independent variable were entered in to the model last. This is analogous to "Type III" sum of squares analysis provided by PROCs REG and/or GLM.

Automated Selection of 'Optimal' Subsets of Independent Variables

PROC LOGISTIC implements three common methods to automate selection of a 'best subset' of independent variables:

- forward selection
- backward elimination
- stepwise selection

Implementation occurs when the user codes the **SELECTION=** option to the right of the slash sign in the model statement. The name of the desired selection method is placed following the equals sign.

Assessing Model Fit

PROC LOGISTIC provides several means of assessing how well the logistic regression model fits the data. These include:

- Hosmer and Lemeshow Chi-Square Goodness of Fit
- R-square 'like' statistics
- Classification Tables

Hosmer and Lemeshow Test

This approach provides a chi-square-based test which assesses how well the data under analysis perform under the null hypothesis that the model fits the data. This test, implemented by the SAS System in Release 6.07, is called by the **LACKFIT** option and is discussed at length by the authors in their text.

R-square 'Like' Statistics

These measures, implemented in SAS System Release 6.10, provide a generalization of the coefficient of determination to the logistic regression model. Their derivation is found both in the Hosmer and Lemeshow text and in *SAS/STAT Software: Changes and Enhancements, Release 6.10*. Two statistics are printed if the **RSQUARE** option is used: the 'adjusted R-square' statistic is appropriate for models containing one or more dichotomous independent variables.

Classification Tables

This approach provides a convenient way to assess the:

- sensitivity
- specificity
- false positive rate
- false negative rate
- proportion of cases correctly classified

by a particular logistic regression model. Classification tables are generated by use of the **CTABLE** option; additional use of the **PPROB** option avoids generation of unnecessary output.

Enhancements to PROC LOGISITIC in Version 8

Substantial enhancements to PROC LOGISTIC have been added in Version 8 of SAS/STAT Software, including:

- The CLASS Statement, which allows incorporation of polytomous categorical independent variables without having to code dummy variables in a Data Step prior to invoking PROC LOGISTIC. The CLASS Statement includes options permitted the user to specify a reference group and to implement different types of effect coding.
- Easy inclusion of interaction terms among independent variables using syntax similar to

that available in PROC GLM. Users can specify the "deepness" of the interactions PROC LOGISITC is to consider. For example, the following MODEL STATEMENT

```
MODEL RESPOND = VAR1|VAR2|VAR3|VAR4 @2;
```

instructs PROC LOGISTIC to consider only the two-way interactions among the independent variables.

The new Output Delivery System (ODS) can be used with PROC LOGISTIC to both enhance the visual quality of the output it generates and to create output SAS data sets containing parts of the output. The latter functionality is quite useful when an analyst wants to create a SAS data set containing "side by side" analyses of competing models.

Additional Functionalities in PROC LOGISTIC

PROC LOGISTIC provides a number of additional functionalities and tests not addressed in this paper. These include:

- detection of outliers and influential observations
- generation of values for a Receiver-Operator Characteristics (ROC) curve to an output data set for subsequent plotting by PROCs PLOT and/or GPLOT
- generation of false positive and false negative rates using Baye's Theorem.

Note: SAS is the trademark of SAS Institute, Cary, NC

References

Allison, Paul, *Logistic Regression Modeling Using the SAS System: Theory and Applications,* SAS Institute, 1998

Stokes, et. al., *Cateogorical Data Analysis Using the SAS System,* SAS Institute, 1996

Stokes, et al., *Categorical Data Analysis Using the SAS System,* Second Edition, 2000

Hosmer and Lemeshow ,*Applied Logistic Regression,* Wiley:, 1989

SAS Institute, Inc: *SAS/STAT Software, Volume 2: the LOGISTIC Procedure*

SAS Institute, Inc.: *SAS/STAT Technical Report P-229, SAS/STAT Software: Changes and Enhancements, Release 6.07*

SAS Institute, Inc.: *SAS/STAT Software: Changes and Enhancements Release 6.10*

SAS Institute, Inc.: *Logistic Regression Examples Using the SAS System* (1995)

SAS Institute, Inc: *SAS/STAT Software: Changes and Enhancements through Release 6.12* (1997)

Acknowledgments

The author would like to thank Miriam G. Cisternas, M.S., M.R.P., formerly of the Technology Assessment Group, San Francisco, and Judy Calem of the United States Environmental Protection Agency, Washington, DC for their comments on earlier versions of this paper. The author is also deeply indebted to Paul Allison, Ph.D., of the University of Pennsylvania, and Philip W. Wirtz, Ph.D., of The George Washington University, for their helpful comments and suggestions.

The author can be contacted at:

Sierra Information Services, Inc.
19229 Sonoma Highway
PMB 264
Sonoma, California 95476 USA
707 996 7380
SierraInfo@ AOL.COM
www.SierraInformation.com

Paper P710

Ideas on Variable Selection and Alternative Links in Procedure CATMOD

Kimberly Hughes DeJarnatt , John Brown University, Siloam Springs, AR
James E. Dunn, University of Arkansas, Fayetteville, AR

ABSTRACT

Two practical features are lacking in SAS procedure CATMOD: (1) for nominal categories of response, only use of baseline logit link functions admits to maximum likelihood estimation, and (2) no provision exists for automated "SELECTION=" model-building. This paper illustrates how both might be implemented, based in the first case on a class of generalized additive models previously presented to SUGI, and improvising on the basis of sweep operations in the second. Agresti-level data sets are used to illustrate the methodology. Even in those cases, we demonstrate that either model fit or power may improve by choosing alternatives to baseline logits. Considerably simplified models often result by zeroing parameters in addition to those suggested by the ANOVA table. The implementing SAS/IML code is available on request. Statisticians and other scientists faced with modeling multinomial response will find this paper of interest.
Key words: multinomial, IML, sweep, Wald statistics, logit.

INTRODUCTION

This is a story of what a next generation version of SAS procedure CATMOD might include. CATMOD's origins are in the work by Grizzle, Starmer and Koch (1969), the 'GSK model', having passed through the now defunct procedure FUNCAT in order to arrive at its current form. Fundamentally, CATMOD operates by fitting a multivariate generalized linear model (MGLM) to a link-transformed multinomial response. The baseline logit, which is appropriate for nominal responses, is the pre-programmed default link function, and adjacent category and cumulative logits are pre-programmed for ordinal responses. While additional link functions can be defined using a RESPONSE statement, only use of baseline logits admits to maximum likelihood estimation. All others default to use of generalized least squares (GLS), an artifact of the original GSK approach. In that GLS is not fully efficient at best, and non-applicable for independent multinomial samples without replication, an alternative to baseline logits seldom is considered as an option. Since Dunn (1985) pointed out the existence of alternatives to baseline logits, even for nominal responses, CATMOD of the future would make fully efficient estimation possible for these options. We demonstrate here how this might be accomplished, and present analyses of familiar data sets where improved model fits were attained.

Automated variable selection, e.g., the SELECTION = option in procedure REG, has a long history in multiple regression, and recently has been implemented for both logistic regression and proportional hazards models in procedures LOGISTIC and PHREG, respectively. Any active data analyst knows the value of these model-building algorithms. However, the problem is even more complex in the case of data sets suitable for CATMOD. Not only are the effects listed in the analysis of variance table subject to retention or deletion, but also these same effects within the individual link functions need be examined. Without limiting ourselves to baseline logits, we show that this is easily implemented using the SWEEP operator contained in the IML procedure, and again demonstrate its effectiveness using familiar data sets.

MATHEMATICAL FORMULATION

We suppose independent multinomial sampling from each of m, d-category multinomial populations, where at most a nominal relation exists among the multinomial categories. Associated with each sample are associated values of c explanatory variables, $\mathbf{x} = (x_1, \ldots, x_c)'$ which may be continuous, or indicators for levels of a categorical variable, or a mixture of both. The resulting data has the form

Population	Category 1	2 … d	Sample Size	$\mathbf{x}'$
1	n_{11}	$n_{12} \ldots n_{1,d}$	n_{1+}	$\mathbf{x}_1'$
2	n_{21}	$n_{22} \ldots n_{2,d}$	n_{2+}	$\mathbf{x}_2'$
	…….			
m	n_{m1}	$n_{m2} \ldots n_{m,d}$	n_{m+}	$\mathbf{x}_m'$

where $P[n_{i1},\ldots,n_{id}] = n_{i+}! \prod_{j=1}^{d} \pi_{ij}^{n_{ij}} / \prod_{j=1}^{d} n_{ij}!$, $0 < \pi_{ij} < 1$, and $\sum_{j=1}^{d} \pi_{ij} = 1$ for i = 1,…,m. Defining $\boldsymbol{\pi}_i = [\pi_{i1}, \ldots, \pi_{id}]'$, we postulate that $\boldsymbol{\pi}_i$ is related to $\mathbf{x}_i$ through a set of multivariate link functions, $\mathbf{f}(\boldsymbol{\pi}_i) = [f_1(\boldsymbol{\pi}_i), \ldots, f_u(\boldsymbol{\pi}_i)]'$, where for a MGLM,

$$\mathbf{f}(\boldsymbol{\pi}_i) = (\mathbf{I}_{(u)} \otimes \mathbf{x}_i')\boldsymbol{\beta} \text{ with } \boldsymbol{\beta} = (\boldsymbol{\beta}_1', \ldots, \boldsymbol{\beta}_u')' \qquad (1)$$

for i = 1,…,m.

Since necessarily u = d – 1 for invertibility (Dunn, 1985), we shall assume that this restriction on u always holds in the following development. A broad class of multivariate link functions (Dunn, 1985) applicable for nominal categories of response are the *generalized additive logits* (GAL), where for any monotone h, mapping (0,1) into the positive real axis, $f_{ij} = \ln[h(\pi_{ij})/h(\pi_{id})]$ for j = 1,…,d - 1. Its inverse is

$$\pi_{ij} = h^{-1}[h(\pi_{id})e^{f_{ij}}], \qquad (2)$$

where π_{id} solves $\sum_{j=1}^{d-1} h^{-1}[h(\pi_{id})e^{f_{ij}}] + \pi_{id} = 1$.

Baseline logits, or *additive logits* (AL) in Aitchison's terminology (1982), with $h(\pi) = \pi$ provide a familiar example, while *additive log-logits* (ALL) and *additive tangent-logits* (ATL) with $h(\pi) = -\ln(\pi)$ and $h(\pi) = \tan(3.14159\ldots\pi/2)$, respectively, provide immediate extensions. Dunn (1985) also noted that if ln in GAL were replaced by any continuous g: $R^+ \leftrightarrow R$ (and exp by g^{-1}), these links also would be invertible and the class described as *generalized additive* (GA) link functions. While the examples presented here involve only the GAL class, the general approach to maximum likelihood, as well as variable selection, are applicable to the broader class of GA links.

EQUATIONS WHICH ALWAYS APPLY FOR MAXIMUM LIKELIHOOD ESTIMATION

Starting with the likelihood function $L = C\prod_{i=1}^{m}\prod_{j=1}^{d}\pi_{ij}^{n_{ij}}$,

$$\ln(L) = \sum_{i=1}^{m}\left[\sum_{j=1}^{d-1} n_{ij}\ln(\pi_{ij}) + n_{id}\ln(1-\sum_{k=1}^{d-1}\pi_{ik})\right] + \ln(C),$$

from which

$$\frac{\partial \ln(L)}{\partial \boldsymbol{\beta}} = \sum_{i=1}^{m}\left[\sum_{j=1}^{d-1}\frac{n_{ij}}{\pi_{ij}}\frac{\partial\pi_{ij}}{\partial\boldsymbol{\beta}} - \frac{n_{id}}{1-\sum_{k=1}^{d-1}\pi_{ik}}\sum_{k=1}^{d-1}\frac{\partial\pi_{ik}}{\partial\boldsymbol{\beta}}\right]$$

$$= \sum_{i=1}^{m}\left[\sum_{j=1}^{d-1}\left\{\frac{n_{ij}}{\pi_{ij}} - \frac{n_{id}}{\pi_{id}}\right\}\frac{\partial\pi_{ij}}{\partial\boldsymbol{\beta}}\right] = \mathbf{w}(\boldsymbol{\beta}), \qquad (3)$$

defining the maximum likelihood estimator (MLE) to be $\hat{\boldsymbol{\beta}}$ satisfying $\mathbf{w}(\hat{\boldsymbol{\beta}}) = \mathbf{0}$. In order to implement Fisher's method of scoring, the matrix of second derivatives of log-likelihood is given by $\partial\,\mathbf{w}'/\partial\boldsymbol{\beta}$ =

$$\sum_{i=1}^{m}\left[\sum_{j=1}^{d-1}\left\{-\frac{n_{ij}}{\pi_{ij}^2}\frac{\partial\pi_{ij}}{\partial\boldsymbol{\beta}} - \frac{n_{id}}{\pi_{id}^2}\sum_{k=1}^{d-1}\frac{\partial\pi_{ik}}{\partial\boldsymbol{\beta}}\right\}(\partial\pi_{ij}/\partial\boldsymbol{\beta})' + \sum_{j=1}^{d-1}\left\{\frac{n_{ij}}{\pi_{ij}} - \frac{n_{id}}{\pi_{id}}\right\}\frac{\partial}{\partial\boldsymbol{\beta}}(\partial\pi_{ij}/\partial\boldsymbol{\beta})'\right].$$

Replacing n_{ij} by its expectation, $E(n_{ij}) = n_{i+}\ \pi_{ij}$ yields $E(\partial\,\mathbf{w}'/\partial\boldsymbol{\beta}) = -\mathbf{W} =$

$$-\sum_{i=1}^{m} n_{i+}\left[\sum_{j=1}^{d-1}\frac{1}{\pi_{ij}}\frac{\partial\pi_{ij}}{\partial\boldsymbol{\beta}}(\partial\pi_{ij}/\partial\boldsymbol{\beta})' + \frac{1}{\pi_{id}}\sum_{k=1}^{d-1}\frac{\partial\pi_{ik}}{\partial\boldsymbol{\beta}}\left(\sum_{j=1}^{d-1}\partial\pi_{ij}/\partial\boldsymbol{\beta}\right)'\right], \qquad (4)$$

a symmetric matrix, where $\mathbf{V} = \mathbf{W}^{-1}$ is the asymptotic covariance matrix of $\hat{\boldsymbol{\beta}}$. Solution is by means of a Newton-Raphson iteration, $\hat{\boldsymbol{\beta}}_{s+1} = \hat{\boldsymbol{\beta}}_s + \mathbf{W}(\hat{\boldsymbol{\beta}}_s)^{-1}\mathbf{w}(\hat{\boldsymbol{\beta}}_s)$, or more concisely, $\hat{\boldsymbol{\beta}}_{s+1}$ solves the linear equations $\mathbf{W}(\hat{\boldsymbol{\beta}}_s)\hat{\boldsymbol{\beta}}_{s+1} = \tilde{\mathbf{w}}(\hat{\boldsymbol{\beta}}_s)$, where the right hand side $\tilde{\mathbf{w}}(\hat{\boldsymbol{\beta}}_s) = [\mathbf{W}(\hat{\boldsymbol{\beta}}_s)\hat{\boldsymbol{\beta}}_s + \mathbf{w}(\hat{\boldsymbol{\beta}}_s)]$, is a vector of "working values" at iteration step s as traditionally defined, e.g., Finney (1971).

From equations (3) and (4) we see that $\hat{\boldsymbol{\beta}}$ depends on choice of link functions only through $\partial\pi_{ij}/\partial\boldsymbol{\beta}$. But even this has a general form for GAL and GA links since from equation (2), $\pi_{ij} = \pi_{ij}(f_{ij}, \pi_{id})$ for j = 1,…,d – 1. Thus,

$$\frac{\partial\pi_{ij}}{\partial\boldsymbol{\beta}} = \begin{bmatrix}\frac{\partial\pi_{ij}}{\partial\boldsymbol{\beta}_1}\\ \cdots \\ \frac{\partial\pi_{ij}}{\partial\boldsymbol{\beta}_{d-1}}\end{bmatrix} = \begin{bmatrix}\delta_{j1}\frac{\partial\pi_{i1}}{\partial f_{i1}}\frac{\partial f_{i1}}{\partial\boldsymbol{\beta}_1} + \frac{\partial\pi_{ij}}{\partial\pi_{id}}\frac{\partial\pi_{id}}{\partial\boldsymbol{\beta}_1}\\ \cdots \\ \delta_{j,d-1}\frac{\partial\pi_{i,d-1}}{\partial f_{i,d-1}}\frac{\partial f_{i,d-1}}{\partial\boldsymbol{\beta}_{d-1}} + \frac{\partial\pi_{ij}}{\partial\pi_{id}}\frac{\partial\pi_{id}}{\partial\boldsymbol{\beta}_{d-1}}\end{bmatrix}$$

$$= \begin{bmatrix}\delta_{j1}\frac{\partial\pi_{i1}}{\partial f_{i1}}\mathbf{x}_i - \frac{\partial\pi_{ij}}{\partial\pi_{id}}\sum_{k=1}^{d-1}\frac{\partial\pi_{ik}}{\partial\boldsymbol{\beta}_1}\\ \cdots \\ \delta_{j,d-1}\frac{\partial\pi_{i,d-1}}{\partial f_{i,d-1}}\mathbf{x}_i - \frac{\partial\pi_{ij}}{\partial\pi_{id}}\sum_{k=1}^{d-1}\frac{\partial\pi_{ik}}{\partial\boldsymbol{\beta}_{d-1}}\end{bmatrix}, \qquad (5)$$

where δ_{rs} is Kronecker's delta, equal to 1 if r = s and 0 otherwise. Equation (5) can be simplified. Summing over j = 1,…,d - 1, obtain

$$\sum_{j=1}^{d-1}\frac{\partial\pi_{ij}}{\partial\boldsymbol{\beta}} = \begin{bmatrix}\sum_{j=1}^{d-1}\frac{\partial\pi_{ij}}{\partial\boldsymbol{\beta}_1}\\ \cdots \\ \sum_{j=1}^{d-1}\frac{\partial\pi_{ij}}{\partial\boldsymbol{\beta}_{d-1}}\end{bmatrix} = \begin{bmatrix}\frac{\partial\pi_{i1}}{\partial f_{i1}}\mathbf{x}_i - \sum_{j=1}^{d-1}\frac{\partial\pi_{ij}}{\partial\pi_{id}}\sum_{k=1}^{d-1}\frac{\partial\pi_{ik}}{\partial\boldsymbol{\beta}_1}\\ \cdots \\ \frac{\partial\pi_{i,d-1}}{\partial f_{i,d-1}}\mathbf{x}_i - \sum_{j=1}^{d-1}\frac{\partial\pi_{ij}}{\partial\pi_{id}}\sum_{k=1}^{d-1}\frac{\partial\pi_{ik}}{\partial\boldsymbol{\beta}_{d-1}}\end{bmatrix}$$

which shares the common term $\sum_{j=1}^{d-1}\frac{\partial\pi_{ij}}{\partial\boldsymbol{\beta}}$ on both sides. Solving for this yields

$$\sum_{j=1}^{d-1}\frac{\partial\pi_{ij}}{\partial\boldsymbol{\beta}} = (1 + \sum_{k=1}^{d-1}\frac{\partial\pi_{ik}}{\partial\pi_{id}})^{-1}\begin{bmatrix}\frac{\partial\pi_{i1}}{\partial f_{i1}}\mathbf{x}_i\\ \cdots \\ \frac{\partial\pi_{i,d-1}}{\partial f_{i,d-1}}\mathbf{x}_i\end{bmatrix}, \qquad (6)$$

which when substituted in equation (5) yields

$$\frac{\partial\pi_{ij}}{\partial\boldsymbol{\beta}} = \begin{bmatrix}[\delta_{j1} - (1+\sum_{k=1}^{d-1}\frac{\partial\pi_{ik}}{\partial\pi_{id}})^{-1}\frac{\partial\pi_{ij}}{\partial\pi_{id}}]\frac{\partial\pi_{i1}}{\partial f_{i1}}\mathbf{x}_i\\ \cdots \\ [\delta_{j,d-1} - (1+\sum_{k=1}^{d-1}\frac{\partial\pi_{ik}}{\partial\pi_{id}})^{-1}\frac{\partial\pi_{ij}}{\partial\pi_{id}}]\frac{\partial\pi_{i,d-1}}{\partial f_{i,d-1}}\mathbf{x}_i\end{bmatrix}. \qquad (7)$$

Equations (6) and (7) are readily computable as intermediate steps in evaluating equations (3) and (4).

ANOVA TESTS USING SWEEP OPERATIONS

Since sweeping $[\mathbf{W}(\hat{\boldsymbol{\beta}}_s) \vdots \tilde{\mathbf{w}}(\hat{\boldsymbol{\beta}}_s)]$ with respect to all rows at each iterative step yields $[\mathbf{W}(\hat{\boldsymbol{\beta}}_s)^{-1} \vdots \hat{\boldsymbol{\beta}}_{s+1}]$, at the MLE solution, $\hat{\boldsymbol{\beta}}$, $\text{var}[\hat{\boldsymbol{\beta}}]$ is estimated by $\mathbf{W}(\hat{\boldsymbol{\beta}}_s)^{-1}$. Once convergence is attained, then a generalized Wald statistic defined by $X^2 = \hat{\boldsymbol{\beta}}'\mathbf{G}'[\mathbf{G}\,\mathbf{W}(\hat{\boldsymbol{\beta}}_s)^{-1}\mathbf{G}']^{-1}\mathbf{G}\hat{\boldsymbol{\beta}}$ for testing $H_0: \mathbf{G}\boldsymbol{\beta} = \mathbf{0}$, where G is any column permutation of $[\mathbf{I}_{(m)} \vdots \mathbf{0}]$, is obtained by sweeping

$$\begin{bmatrix}\mathbf{W}(\hat{\boldsymbol{\beta}}_s)^{-1} & \hat{\boldsymbol{\beta}}\\ \hat{\boldsymbol{\beta}}' & 0\end{bmatrix} \qquad (8)$$

with respect to m rows corresponding to subscripts of non-zero columns of **G**. The result of the sweep is $\begin{bmatrix}\bullet & \bullet\\ \bullet & -X^2\end{bmatrix}$, so that one only needs to change the sign of the bottom corner element to obtain the test statistic, $X^2 \sim \chi^2_{(m)}$ under H_0. Since matrix (8) is unaffected by the sweep, it may be swept again to test other hypotheses.

Consider an example given by Agresti (1996) in which alligators from five lakes and two size classes were classified with respect to their primary food choice. The data appears in Table 1.

Table 1. Alligators classified by primary food choice.

		Primary Food Choice				
Lake	Size (m)	Fish	Invert.	Reptile	Bird	Other
Hancock	≤ 2.3	23	4	2	2	8
	>2.3	7	0	1	3	5
Oklawaha	≤ 2.3	5	11	1	0	3
	>2.3	13	8	6	1	0
Trafford	≤ 2.3	5	11	2	1	5
	>2.3	8	7	6	3	5
George	≤ 2.3	16	19	1	2	3
	>2.3	17	1	0	1	3

Table 2 shows identical chi-squared statistics given by CATMOD using default AL links and those resulting from the sweep operations described here. Since the explanatory variables, Lake and Size Class, are categorical, zero-sum restrictions were imposed (as does CATMOD) before the sweep operations. Because the order of parameters corresponds to that of equation (1) (unlike CATMOD), rows 1, 6, 11, and 16 of matix (8) were swept to obtain SS for Intercept; rows 2, 3, 4, 7, 8, 9, 12, 13, 14, 17, 18, and 19 were swept to obtain SS for Lakes; and rows 5, 10, 15, and 20 were swept to obtain SS for Size Class. This was repeated using ALL links, with results also shown in Table 2.

Table 2. Analysis of variance of primary food choice data in Table 1 using additive logistic (AL) and additive log-logistic (ALL) links.

		AL		ALL	
Source	d.f.	X^2	p-value	X^2	p-value
Intercept	4	70.39	<0.0001	84.02	<0.0001
Lake	12	35.49	3.91E-4	48.30	2.78E-6
Size	4	18.76	8.76E-4	20.29	4.37E-4
Fit	12	17.08	0.1466	17.85	0.1204

Even though lack-of-fit X^2 was marginally smaller for AL links, use of ALL links resulted in an increase of 12.81 in Lake X^2. Since the degrees of freedom are identical in both cases, the ALL-based test for Lake effects has increased power over that based on AL links. Since our focus until now primarily had been on comparing lack-of-fit under alternative choices of links, the potential for increased power as a result of link choice remains unexplored territory.

In fact it is not difficult to find examples in the published literature where use of ALL links provides a better fit than use of AL links. Grizzle, *et al.* (1969) illustrated GSK methodology using numbers of depletions (deaths) incurred in litters of mice classified by litter size and exposure to either treatments A or B. The data appears in Table 3, and the resulting ANOVA, treating litter size as quantitative, is shown in Table 4. Use of ALL links rather than AL reduced lack-of-fit X^2 from 10.21 to 9.04. Note that a marked increase in the X^2 statistic for litter size also has occurred.

Table 3. Number of litters showing 0, 1 or 2+ depletions.

Litter Size	Treatment	Number of depletions		
		0	1	2+
7	A	58	11	5
	B	75	19	7
8	A	49	14	10
	B	58	17	8
9	A	33	18	15
	B	45	22	10
10	A	15	13	15
	B	39	22	18
11	A	4	12	17
	B	5	15	8

Table 4. Analysis of variance of number of mice depletions treating litter size as quantitative using additive logit (AL) and additive log-logit (ALL) links.

		AL		ALL	
Source	d.f.	X^2	p-value	X^2	p-value
Intercept	2	107.15	<0.0001	123.03	<0.0001
Size	2	78.41	<0.0001	95.81	<0.0001
Treatment	2	6.71	0.0350	5.68	0.0585
Fit	14	10.21	0.7464	9.04	0.8287

BACKWARD ELIMINATION USING SWEEP

Quite often, even after eliminating factors and interactions based on the ANOVA table, many of the 1 d.f. Wald statistics, i.e., X^2 = (estimate/standard error)$^2 \sim \chi^2_{(1)}$, are not significantly different from zero, thus suggesting that a simpler model may be attained. The relative magnitudes of these, from small to large, suggest a tentative order for elimination. Since conditional odds ratios ala Agresti (1996) are identically 1 for all those non-intecept parameters which are zeroed, this often leads to considerable economy of interpretation. In order to eliminate additional parameters, one simply does successive sweeps of matrix (8) with respect to rows corresponding to indices of increasing Wald statistics until lack of statistical significance of the entire set of zeroed parameters is no longer obtained. This is illustrated in Table 5, based on the primary food choice data in Table 1, and using both AL and ALL links. Profiles are in order: bird, fish, invertebrates, other, reptiles (using reptiles as the baseline); George, Hancock, Oklawaha, Trafford; and >2.3, $\leq$ 2.3. Choosing α = 0.05 as the nominal significance level, selection stopped for AL links with 10 parameters declared not significantly different from zero (p = 0.105), while for ALL links, selection stopped with 7 parameters declared not significantly different from zero (p = 0.071). This difference seems attributable to the fact that Wald statistics generally were larger using ALL rather than AL links, which corresponds to a similar trend already noted in the ANOVA statistics in Table 2. With the exception of parameter #12, all other parameters selected using ALL links were included in the set selected using AL links.

Table 5. Order of selection of parameters to be zeroed in models for alligator primary food choice, using additive logits (AL) and additive log-logits (ALL).

		AL		ALL	
Index	Parameters	X^2	Order	X^2	Order
1	β_1	0.388	4	0.864	3
2		2.149	10	3.053	
3		1.349	8	1.603	7
4		3.756		4.306	
5		0.120	2	0.057	1
6	β_2	29.03		52.32	
7		4.371		6.881	
8		0.555	5	3.488	
9		2.882		2.698	
10		0.367	3	0.232	2
11	β_3	9.109		14.38	
12		3.676		2.494	8
13		4.084		8.979	
14		0.006	1	1.476	6
15		9.008		13.39	
16	β_4	3.168		3.155	
17		1.574	9	0.942	4
18		1.166	7	2.731	
19		4.837		4.947	
20		1.099	6	1.450	5

As a final remark in this section, note that we have been able to produce Table 5 based on only a single model fit for each set of links. This is a distinct advantage of backward elimination since model-fitting for MGLM's tends to be computationally expensive in that the size of parameter sets associated with MGLM's tends to be explosive. We have not based our model-building decisions entirely on 1 d.f. Wald statistics, but rather do a single, multiple d.f. test to make a decision about whether or not to zero the entire tentative subset of parameters. It's a simple computational step using sweep operations, and leads to a single p-value at the end.

CONCLUSION

For categorical predictors, the preliminary step of reparameterizing to full rank, e.g., using zero-sum restrictions, has not been essential in any of the data sets we've examined, since all of the computational steps basically are identical regardless of whether a true or a g2-inverse is used. This increasingly has become an encumbrance for CATMOD users who have forgotten or never learned the concept of reparameterizing a model to full rank.

For the alligator primary food choice data, a general increase in the size of Wald statistics was observed, both individually and in the ANOVA table, as a result of replacing AL links with ALL links. Selection of 7 parameters to zero in the latter case, compared to 10 using AL links is associated with this increase. Clearly this is attributable to standard errors using ALL links which are half or less than those based on AL links. We have not yet explored whether this is a general occurrence, or simply specific to this data set.

For some non-trivial data sets encountered in our consulting work, we have been able to obtain major model simplifications using an *ad hoc* approach to parameter selection now carried on systematically by our backward elimination algorithm. In the case of the alligator primary food choice example, we would focus on interpreting 7 or perhaps 9 conditional odds ratios involving reptiles, rather than the initial undifferentiated 16, 9 or 7 of which we expect to be close to 1. We are aware, however, that one cannot always fit a "full model" to use as a starting point for backward elimination. We can, however, compute efficient score statistics without fitting the model, and this is an avenue which we are pursuing.

REFERENCES

Agresti, A. 1996. *An Introduction to Categorical Data Analysis*, John Wiley & Sons, New York.

Aitchison, J. 1982. The statistical analysis of compositional data. *J. of the Royal Statistical Society (B)* 44:139 – 177.

Dunn, J.E. 1985. The role of invertibility and symmetry of polytomous metameters for GSK and other related analyses. *Proc. of 10th Annual Conf. of SAS Users Group*, pp. 990 – 999.

Finney, D.J. 1971. *Probit Analysis*, 3rd edition, Cambridge University Press.

Grizzle, J.E., Starmer, C.F., and Koch, G.G. 1969. Analysis of categorical data by linear models. *Biometrics* 25:489 – 504.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Kimberly Hughes DeJarnatt
John Brown University
2000 W. University
Siloam Springs, AR 72762
(501) 524-7279
khughes@jbu.edu

Paper P711

Using the SAS® System to Study the Gender and Level Measurement Equivalence of a Multi-rater Survey

Jim Penny, Center for Creative Leadership, Greensboro, NC

ABSTRACT

This research used logistic regression to model item responses from a popular 360-for-development survey. The survey contained 57 items on 11 scales. The model used gender and rater group to identify items that exhibited differential item functioning (DIF). The rater groups were self, boss, peer, and direct report. The sample consisted of 752 survey families where a survey family consisted of a matched set of four surveys: one self, one boss, one peer, and one direct report. The sample of 3008 surveys contained 76% male and 24% female raters. The procedure to flag items exhibiting differential functioning used effect size computed from Wald chi-square statistics rather than statistical significance, resulting in fewer flagged items.

Three items exhibited rating anomalies due to the gender of the rater or ratee. Twelve items exhibited DIF attributable to rater group. In each instance, the apparent effect of the DIF was small. An examination of the maximum likelihood parameter estimates suggested the rater group DIF was the possible result of hierarchical complexity. The DIF due to gender conformed to expectations of gender-related stereotypical interpretations of item text. This research further suggested that DIF due to environmental complexity could be a naturally occurring phenomenon in some 360-assessment, and that the interpretation of some 360-feedback might need to include the potential for such DIF to exist.

INTRODUCTION

There has been a veritable explosion in the use of 360-assessment, a form of multi-rater assessment for managerial development in organizations. The process of 360-assessment involves providing managers with feedback from four sources: (1) the manager's boss, (2) the manager's subordinates or direct reports, (3) the peers or the customers of the target, and (4) the self. Although the notion of receiving multi-source feedback is not new, at least one premise of multi-rater methodology remains unresolved: Does the 360-process produce a similar measure from different rater groups just as a measuring tape produces a similar measure with different carpenters? Alternatively, does the 360-methodology provide an equivalent measure with each rater group?

In addition to the increase in the use of 360-assessment, there continues to exist the question of whether or not women receive fair assessments of performance in the workplace, or do a variety of psychological and sociological biases influence the results of 360-methodology when applied to women? That is, does 360-methodology provide an equivalent measure for both men and women, or do the gender biases that sometimes accompany performance appraisals function to influence the manner in which some raters interpret some items? Moreover, might there exist a potential interaction between rater group and gender of the ratee or between the gender of the rater and the gender of the ratee?

DIFFERENCES AMONG RATER GROUPS

Discussions of the differences between self and others' ratings sometimes arise during 360-feedback sessions (Van Velsor & Leslie, 1991), making the existence of measurement equivalence important to interpretations of 360-feedback. It is possible, if not expected, that a feedback recipient will receive low ratings in one area from one rater source while receiving high ratings in that same area from another rater group. A manager, for example, may be interpersonally skilled with bosses yet cold and aloof with direct reports. This manager, therefore, could receive high ratings on interpersonal skills by the boss while receiving low ratings on this dimension by direct reports. However, in the interpretation of the between group difference, there is the underlying assumption that the raters are responding to their perceptions based on their observations of behaviors exhibited by the manager, and that two raters with similar observations will respond similarly to a given item even though the raters may occupy positions of different levels.

CONTINGENCY THEORY

Contingency theories of leadership (Fielder 1978; Fielder & Chemers, 1982) suggest that disparate ratings can be an indication of an effective manager, and that gaps in the perspectives between groups of raters are often a naturally occurring phenomenon of management. Moreover, Yukl (1981, pp. 99-119) suggested that managers often change their behavior to fit particular situations, and, following this line of argument, managers who behave differently toward different groups of co-workers may receive disparate ratings from members of those groups. Hence, between group differences may be an acceptable outcome for some managers.

Concomitant with the interpretation of group differences in 360-feedback is the expectation that a different interpretation of the item by one group of raters does not contribute substantially to the observed difference, and that the observed difference is only the result of behavioral differences produced by the circumstances of contingency. However, it seems reasonable to anticipate that some items may tap into differences produced by organizational contingency to a greater degree than do some other items. The ratings produced by items influenced by contingency, then, become composite scores comprised not only of an estimate of the managers standing on the trait measured by the survey but also of the degree to which contingency influenced the ratings.

COMPLEXITY THEORY

Jacques (1996) and Jacques & Clement (1994) suggested that the degree of environmental complexity and ambiguity seen by a person within an organization generally increases with rank. That is, a supervisor of the manager is likely to see a more complex and a more difficult to comprehend environment than is the direct report of the manager. For example, 360-surveys sometimes contain items that measure the resourcefulness of the manager, and one might argue that the increase in complexity from one level to another could produce different interpretations of what resourcefulness means. Hence, it seems reasonable to anticipate that differences in environment may influence the ratings given on a 360-survey.

As with contingency theory, the interpretation of between-group differences in 360-ratings is posited on the expectation that an observed difference is solely a function of the behavioral differences witnessed by the raters. However, it seems reasonable to anticipate that some items may tap environmental complexity more so than other items. In that event, a rating difference produced by such items may represent a composite of not only the standing of the manager on the trait assessed by the items but of also the degree to which complexity influences the rater's interpretation of that item. One might argue, then, that a manager reviewing 360-feedback could choose to make behavioral changes due not only to the behavioral observations of the raters but also, at least in part, to ratings produced by the anomalous functioning of some items.

DIFFERENCES BETWEEN GENDERS

Although far from conclusive, a condition indicative of the

complex role gender plays in society, many studies have examined the influence of gender on ratings of managerial effectiveness over the past twenty-five years. Some studies have demonstrated statistically significant differences attributable to gender of the ratee (Bartol & Butterfield, 1976; Jacobson & Effertz, 1974; Rosen & Jerdee, 1974; Schmitt & Lapin, 1980). Other studies have failed to produce such differences (Pulakos & Wexley, 1983; Thompson & Thompson, 1983).

In circumstances where a woman functions in a role often associated with men, one might expect to find to find differences attributable to the interaction of "role gender" and gender of the person filling the role. For example, a woman working as a firefighter might find herself at risk to receive performance reviews that carry not only an assessment of her performance but that also carry the influence of the interaction of her gender with the "gender" of the job. Of course, it stands to reason that men filling roles often associated with women will experience similar bias in performance reviews. Bartol & Butterfield (1976) and Rosen & Jerdee (1974) identified statistically significant interactions between role gender and person gender; however, Jacobsen & Effertz (1974) and Mobley (1982) failed to identify such interactions.

The influence of gender is likely a composite of many factors, some of which may have small effects until they exist in concert with gender. Moreover, one could argue that raters are more likely to remember the gender of the manager long after forgetting particular exemplars of either good or bad behaviors. Such biases attributable to gender may influence ratings more than other factors and behaviors. For instance, Nieva & Gutek (1980) suggested that level of qualification, level of performance, degree of inference resulting from the ratings, and sex-role incongruence may each explain a portion of rating variability. Other explanatory factors also have arisen in the study of gender differences in managerial ratings. Cash, Gillen, & Burns (1977) suggested that some raters attribute a man's success to ability while attributing the success of a woman to effort and luck. Greenhaus & Paurasuraman (1993) confirmed those findings, though only for women in the highest performance levels. At moderate levels of performance, they found that raters were likely to use ability to explain a woman's success.

In addition, one might also suggest that stereotypical behaviors and biases may influence performance ratings. Noe (1988) and Powell (1988) gave evidence to suggest that negative stereotypes against minorities and women can have a substantial impact on ratings of performance and effectiveness. Moreover, Martell (1991) found that if there existed less time to make an assessment of managerial performance, the performance of men was likely to receive higher ratings than comparable performances by women. Maurer & Taylor (1994) rendered this finding even more poignant when they demonstrated that the perceived masculinity of the ratee could produce higher ratings. Lastly, Powell & Butterfield (1989) suggested that the definition of "good manager" still carried connotation of masculinity despite the growing population of female managers.

It seems reasonable to anticipate that some items will tap perceptual differences due to gender to a greater extant than will other items, and one might also suggest that particular items may tap particular gender-related biases and either increment or decrement differentially the resulting 360-ratings. In addition, one may ask if the differential functioning of the item is due to the gender of the rater, the gender of the ratee, the interaction of the two genders, all three or some other combination. Moreover, do either or both genders interact with the rater group?

RESEARCH QUERSTIONS

This research sought to establish the degree to which differential item functioning attributable to rater group and gender may influence the ratings of a 360-survey. That is, will a given manager receive similar ratings from the boss, a direct report, and a peer if those three other raters have had similar experiences with the manager? Are there components in item ratings attributable to the gender of the rater or to the gender of the ratee? Are there items that function differently for particular combinations of rater and ratee gender? Is there evidence to support the existence of an interaction between the gender of ratee and the rater group? Moreover, if such items exist, does an explanatory model exist using extant measurement and psychological theory? Finally, if such items exist and if such explanatory models exist, what, then, may be the subsequent implications for the interpretation of the 360-feedback.

METHODOLOGY

This research used logistic regression to detect DIF. Swaminathan & Rogers (1990) first presented this methodology and demonstrated its relationship to the Mantel-Haenszel procedure (Mantel & Haenszel, 1959; Holland & Thayer, 1988). Swaminathan & Rogers (1990) and Clauser & Mazor (1998) have shown that logistic regression is equal in power to the Mantel-Hanzsel procedure for the detection of uniform DIF. Moreover, these same authors with Penny & Johnson (1999) have shown that the Mantel-Haenszel procedure may lack sufficient statistical power to detect some instances of nonuniform DIF. However, Rogers (1989), Rogers & Swaminathan (1993), and Swaminathan & Rogers (1990) found that logistic regression procedures likely to have sufficient power to detect non-uniform DIF.

Much of the initial research in the use of logistic regression for the detection of DIF involved the examination of dichotomous items; that is, items with two possible responses, usually 0 and 1. However, logistic regression is easy to extend to polytomous data where the respondent chooses one of an ordered set of responses. Samejima (1969, 1979) presented the Graded Response Model that describes such item responses which are common on 360-surveys. For example, the Graded Response Model describes a Likert-type item using a 5-point scale of 1=Strongly Disagree to 5=Strongly Agree positing a response function for each point on the scale according to

$$P(x = k) = \frac{1}{1 + e^{(-a(\theta - b_{k-1}))}} - \frac{1}{1 + e^{(-a(\theta - b_k))}} = P^*(k) - P^*(k+1)$$

in which a is the discrimination parameter and b_{k-1} is the threshold parameter. $P^*(k)$ is the item response function that describes the probability that a response is in category k or higher. In this model, the discrimination parameter is constant for all categories of k, and the threshold parameter, b_{k-1}, is the point on the θ-axis where the probability exceeds 50 percent that the response is in the next category. Researchers sometimes call the threshold parameter the "location" parameter.

For the detection of DIF using logistic regression with polytomous data (Clauser & Mazor, 1998), I can adapt the framework of Samejima (1969, 1979) and write the equation

$$P(x = k) = \frac{1}{1 + e^{-z_{k-1}}} - \frac{1}{1 + e^{-z_k}},$$

where $P(x=k)$ is the probability of a response k to a particular item from a respondent of standing θ, and where k takes the values of the Likert-type response scale (Miller & Spray, 1993; Samejima, 1969, 1979; Swaminathan & Rogers, 1990). From this point forward, I will drop the subscript i on z to make the model easier to read.

MODELLING ITEM RESPONSES

Were the existence of DIF not an issue, I would write z as

$$z = \tau_0 + \tau_1 \theta,$$

where θ represents the standing of the ratee on the attribute that the survey measures. The symbols τ_0 and τ_1 represent the intercept and the slope parameters of the logistic regression model; these symbols also represent forms of the discrimination and location parameters of the Graded Response Model. This

logistic model represents the situation where the rater group membership and gender do not influence the item response, and where the only factor that does influence the response is the standing of the ratee on the attribute that the survey measures.

To expand the model to include components to represent effects due to gender and rater group membership, I would write

$$z = \tau_0 + \tau_1\theta + \tau_2 g + \tau_3 r ,$$

where *g* and *r* represent gender and rater group membership, respectively, and τ_2 and τ_3 represent the logistic regression parameters for those two classifications. This model describes the instance where only uniform DIF exists. I can define the values for *g* in the typical 360-survey as *{male, female}*. Similarly, I can define the values for *r* as *{self, direct report, peer, boss}*. Later, to contrast the functioning of an item across these values, I used dummy codes to represent each value.

I can expand this model to accommodate the potential existence of nonuniform DIF by the addition of two more terms to produce

$$z = \tau_0 + \tau_1\theta + \tau_2 g + \tau_3 r + \tau_4 g\theta + \tau_5 r\theta ,$$

where the two new terms indicate an interaction, respectively, between (a) gender and standing on the attribute that the survey measures, and (b) rater group membership and standing. The symbols τ_4 and τ_5 represent the logistic regression parameters for these two interaction terms, respectively.

To complete the model for this research, I included two additional terms. One term is to indicate the possible interaction between rater group membership and gender; the other term is to indicate the possible three-way interaction of rater group membership, gender, and standing on the attribute that the survey measures. The types of DIF represented by these two terms are uniform and nonuniform, respectively. I can write this model as

$$z = \tau_0 + \tau_1\theta + \tau_2 g + \tau_3 r + \tau_4 g\theta + \tau_5 r\theta + \tau_6 gr + \tau_7 gr\theta ,$$

where the symbols τ_6 and τ_7 represent the logistic regression parameters for these two additional interaction terms, respectively.

A CLOSER LOOK AT MODELLING GENDER

Although this model permits the examination of the rater group and rater gender effects, it does not permit the comparison of the gender of the rater to the gender of the ratee. That is, is there evidence to suggest that some items exhibit DIF that is attributable to men rating women, women rating women, and so forth? To examine such an interaction, the model needs to include terms for the two genders in addition to the cross product of those genders. It is possible to add those terms to this model, but to do so leads to a conceptual problem: For the self-rater, do the self-ratings of a male represent a man rating a man or the self rating the self? In a manner of thinking, the self-ratings do represent gender-on-gender ratings, and there may be the occasional self-rater who can step outside of the self and produce gender-on-gender ratings, but it seems far more reasonable that the gender of the self-rater is not as important to the self-ratings as is self-awareness.

Hence, to examine the interaction of rater gender and ratee gender in addition to rater group and gender, I used the model

$$z = \tau_0 + \tau_1\theta + \tau_2 r + \tau_3 g_s + \tau_4 g_r + \tau_5 r g_s + \tau_6 r g_r + \tau_7 g_s g_r + \tau_8 r g_s g_r + \tau_9 r\theta + \tau_{10} g_s\theta + \tau_{11} g_r\theta + \tau_{12} r g_s\theta + \tau_{13} r g_r\theta + \tau_{14} g_s g_r\theta + \tau_{15} r g_s g_r\theta$$

where θ represents the covariate which is a proxy for standing on the trait measured by the survey, *r* represents the rater group, g_s represents the gender of the self-rater, and g_r represents the gender of the other rater. The additional terms represent the various interactions that may be important to understanding the functioning of the items. The values of the two gender variables are *male* and *female*. However, the values of the rater group variable no longer include *self*, but only the values *direct report*, *peer*, and *boss*.

STATISTICAL SIGNIFICANCE AND EFFECT SIZE

I used the SAS® System to evaluate this model for the data I collected. The SAS System produced Wald Chi Square statistics to test the null hypotheses that that the parameter estimates of τ_0 through τ_{15} were statistically significantly different from *0*. It was my anticipation that τ_0 and τ_1 would routinely achieve statistical significance. In addition, it was my anticipation that the parameter estimates of τ_2 through τ_{15} would not routinely achieve statistical significance.

However, I knew that, with number of terms in the model I had chosen, I would be making many statistical tests of significance, and that the experiment-wise Type I error rate could be high. To compensate for the accumulated Type I error rate that could naturally occur in this research and to avoid the complex power analysis (Hsieh, 1989; Whittemore, 1981) of logistic regression that would suggest an appropriate number of subjects to evaluate the model, I decided to use effect size instead of statistical significance. I chose the technique presented in Penny & Johnson (1999) and converted the Wald chi-square statistic to an effect size, *w*, described in Cohen (1988, ch. 7). The formula that relates the effect size to the sample size is

$$X^2 = nw^2$$

where X^2 is the chi-square statistic, *n* is the sample size, and *w* is the effect size. Cohen (1988, ch. 7) used the arbitrary values of .1, .3, and .5 to indicate small, medium, and large effects, respectively. Although these values are arbitrary, Penny & Johnson (1999) found that those values appeared to connote well derived from the Mantel-Haenszel chi-square statistic used to identify DIF, and I used the three values to define four effect ranges to categorize the DIF I discovered with the logistic regression model. These ranges were nil-to-small, small-to-medium, medium-to-large, and large-to-extreme.

CLASSIFICATION OF TYPE OF DIF

After I classified the items by the type of DIF, either uniform or nonuniform, and the source of DIF, I classified them further according to the apparent explanation of the differential functioning. An item influenced by complexity theory should produce maximum likelihood parameter estimates suggestive of a continuum from direct report to boss. For example, a boss may interpret an item that assesses the resourcefulness of a manager differently than would a direct report because the boss, by virtue of working in a more complex and, perhaps, more ambiguous environment, may have a different idea of what actions, and quantities of actions, constitute resourcefulness in a manager. If a manager, boss, peer and direct report have similar assessments of the manager's performance, and that item functions to augment the ratings of the direct report over those of the boss while not altering the ratings from the self and peers, then the item is exhibiting differential item functioning of a type that produces evidence of a continuum.

In this research, I denoted this type of differential functioning as "DR(+) to S/P to B(-)." To save space, I often dropped the "S/P" part of the notation to produce "DR(+) to B(-)." Hence, an item that functioned the produce differentially lower ratings from direct reports would receive the notation "DR(-) to B(+)."

An item influenced by contingency theory should produce maximum likelihood parameter estimates that suggest a contrast of one rater group to all the others. For example, an item that assesses the propensity of a manager to learn the work performed by direct reports may function differently for direct reports than it does for any other rater group. If the item functions to differentially decrement the ratings from the direct report raters, I can describe the functioning using the notation "DR(-) vs. all others." Of course, another item could function to isolate any of the other rater groups, and the ratings from the isolated group could be incremented or decremented resulting in notation such as "P(+) vs. all others" or "B(-) vs. all others."

I classified items influenced by gender into groups suggestive of the gender stereotype tapped by the item by reviewing the text of

the item after reviewing the regression results. After the final categorization, I presented my findings to a review panel of 360-process and 360-content experts who proceeded to challenge my classifications.

DATA

I used the 1996-1999 Prospector® database from the Center for Creative Leadership (CCL) for this research. Prospector is a 360-assessment-for-development feedback instrument developed to give managers and executives insight into their strengths and development needs. The instrument consists of 57 items designed to assess eleven domains related to managerial effectiveness. Table 1 presents the names and descriptions of the 11 scales. Each item used a 7-point, Likert-type response format. The textual anchors for the response scale were 1="very strongly disagree" to 7="very strongly agree" with 4="neutral" providing the anchor for the middle of the scale.

I selected a random sample of 752 survey families for use in this research. Each survey family included matched ratings from a manager (the self or ratee), a peer, a boss, and a direct report resulting in 3008 total surveys. Tables 2 and 3 give the breakdown of the gender, race, and rater group information for this sample. Because each family contained exactly one survey from each rater group, there were 752 each of self, peer, boss, and direct report surveys.

RESULTS

THE INFLUENCE OF STANDING ON RESPONSES

The main effect of standing on the attribute measured by the survey had by far the greatest influence on the item ratings of all the terms in the logistic model. The average effect due to the covariate was in the large-to-extreme category with only a very few items exhibiting a main effect due to covariate in the medium-to-large category. The effect of the covariate was never smaller than medium.

DIF ATTRIBUTABLE TO RATER GENDER

One item exhibited uniform DIF attributable to the gender of the rater. The effect was .12, placing the item in the lower end of the small-to-medium category. The product of the differential functioning was to elevate differentially the ratings given by male raters over those given by female raters. The text of the item was "Is willing to make substantial personal sacrifices for the sake of the business." The item occurred on the scale called "Committed to making a difference."

DIF ATTRIBUTABLE TO RATEE GENDER

Two items exhibited uniform DIF attributable to the gender of the ratee. The first item occurred on the scale called "Open to criticism." The text of the item was "Is not threatened by criticism." The second item occurred on the scale called "Seeks broad business knowledge." This item also exhibited DIF with respect to the rater type, and was the only item to exhibit DIF of two types. The text of this item was "Understands the financial side of the business." The effect associated with both items was .12, which placed the items in the lower end of the small-to-medium category. Both items functioned to differentially increment the ratings given to men.

DIF ATTRIBUTABLE TO RATER SOURCE

Twelve items on the survey exhibited uniform DIF attributable to the rater group. Table 4 presents these items along with the effect size and the impact of the DIF on the ratings. In each case, these items functioned to produce evidence of continuity from direct report to boss. Half of the items functioned to differentially increment the ratings given by bosses while the other half functioned to increment the ratings given by direct reports. Within particular scales, the direction of this functioning was consistently the same.

DISCUSSION

I used effect size instead of statistical significance to flag anomalous items in this study. I did this to avoid the complex power analysis that would suggest an appropriate sample size for use with logistic regression and these data. By using effect size, I failed to flag several items that I would have flagged were I using the typical p-value of .05, though the use of a Bonferroni correction may have ameliorated that Type I error rate. This result suggest to me that the sample size was sufficiently large for the analysis of the logistic model, and that the incidence of false positives is lower than it might otherwise be using statistical significance.

In most studies of differential item functioning, the existence of DIF is not something that is good, and I tend to concur with those who suggest that DIF is a quantity to remove from an assessment. The occurrence of DIF may threaten the validity of an assessment, and reduce the usefulness of the survey (Lord, 1980; Penny & Johnson, 1999). Psychological measures that assess the learning, the performance, or perhaps even the potential, of a candidate for a higher appointment are not the places where one generally wants to find measures influenced, even in part, by demographic quantities such as gender, race, nationality, or native language.

However, this study, suggests that the existence of differential item functioning may not be a completely bad thing, and further suggests that DIF produced by environmental complexity might be a naturally occurring phenomenon in the workplace. Hence, one could argue that the interpretation of 360-feedback reports should take into account the possibility of anomalous item functioning produced by such environmental and experiential differences. Moreover, this anomalous functioning may function to ameliorate, or exacerbate, some observed rating differences, and the degree of either may depend, at least in part, on the standing of the ratee on the trait measured by the survey.

The DIF related to the gender of the ratee and the raters seems another matter. The development of a useful and competitive 360-assessment is a long, involved, and arduous process where items receive repeated critical review from many groups of people. That this research uncovered (a) no DIF related to the cross of rater and ratee gender, (b) no DIF related to the cross of rater group and ratee gender, and (c) very little DIF attributable to the gender of ratees or raters is an important and heartening finding. However, that some items still tap, if ever so lightly, particular instances of stereotypical gender associations reminds us of the long road it has been towards gender equality and of the longer road that remains before us.

REFERENCES

Bartol, K. M., & Butterfield, D. A. (1976). Sex effects in evaluating leaders. *Journal of Applied Psychology, 61*, 446-454.

Cash, T. F., Gillen, B., & Burns, D. S. (1977). Sexism and "beautyism" in personnel consultant decision making. *Journal of Applied Psychology, 62*, 301-310.

Clauser, B. E., & Mazor, K. M. (1998). Using statistical procedures to identify differentially functioning test items. *Educational Measurement: Issues and Practices, 2*, 31-44.

Cohen, J. (1988). *Statistical power analysis for the behavioral sciences*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Fiedler, F.E. (1978) The contingency model and the dynamics of the leadership process, In *Advances in experimental social psychology*, ed. L. Berowitz. New York: Academic Press.

Fielder, F.E. & Chemers, M.M. (1982). *Improving leadership effectiveness: The leader match concept*. 2nd Ed. New York: Wiley.

Greenhaus, J. H., Parasuraman, S. (1993). Job performance attributions and career advancement prospects: An examination of gender and race effects. *Organizational Behavior and Human Decision Processes, 55*, 273-297.

Holland, P. W., & Thayer, D. T. (1988). Differential item performance and the Mantel-Haenszel procedure. In H. Wainer & H. I.

Braun (Eds.), *Test validity* (pp. 129-145). Hillsdale, NJ: Erlbaum.

Hsieh, F. Y. (1989). Sample size tables for logistic regression. *Statistics in Medicine, 8*, 795-802.

Jacobson, M. B., & Effertz, J. (1974). Sex roles and leadership perceptions on the leaders and the led. *Organizational Behavior and Human Performance, 12*, 383-397.

Jacques, E. (1996). *Requisite organization: A total system for effective managerial organization and managerial leadership for the 21st century*. Cambridge, MA: Cason Hall & Co.

Jacques, E., & Clement, S. D. (1994). *Executive leadership: A practical guide to managing complexity*. Cambridge, MA: Basil Blackwell.

Lord, F. (1980). *Application of item response theory to practical testing problems*. Hillsdale, NJ: Erlbaum.

Mantel, N., & Haesnzel, W. (1959). Statistical aspects of the analysis of data from retrospective studies of disease. *Journal of the National Cancer Institute, 22*, 719-748.

Martell, R. F. (1991). Sex bias at work: The effects of attention and memory demands on performance ratings of men and women. *Journal of Applied Social Psychology, 21*, 1939-1960.

Maurer, T. J., & Taylor, M. A. (1994). Is sex by itself enough? An explanation of gender bias issues in performance appraisals. *Organizational Behavior and Human Decision Processes, 60*, 231-251.

Miller, T. R., & Spray, J. A. (1993). Logistic discriminant function analysis for DIF identification of polytomously scored items. *Journal of Educational Measurement, 30*, 107-122.

Mobley, W. H. (1982). Supervisor and employee race and sex effects on performance appraisals: A field study of adverse impact and generalization. *Academy of Management Journal, 25*, 598-606.

Nieva, V. F., & Gutek, B. A. (1980). *Women and work: A psychological perspective*. New York: Praeger.

Noe, R. A. (1988). Women and mentoring: A review and research agenda. *Academy of Management Review, 13*, 65-78.

Penny, J., & Johnson, R. L. (1999). How group differences in matching criterion distribution and IRT item difficulty can influence the magnitude of the Mantel-Haenszel chi-square DIF index. *The Journal of Experimental Education, 67*, 343-366.

Powell, G. N. (1988). *Women and men in management*. Newbury Park, CA: Sage.

Powell, G. N., & Butterfield, D. A. (1989). The "good manager:" Did androgyny fare better in the 1980's? *Group and Organizational Studies, 14*, 216-233.

Pulakos, E. D., & Wexley, K. N. (1983). The relationship among perceptual similarity, sex, and performance ratings in manager-subordinate dyads. *Academy of management Journal, 26*, 129-139.

Rogers, H. J. (1989). A logistic regression procedure for detecting item bias. *Dissertation Abstracts International, 50*, 3928A. (University Microfilms No. 90-11,788).

Rogers, H. J., & Swaminathan, H. (1993). A comparison of logistic regression and Mantel-Haenszel procedures for detecting differential item functioning. *Applied Psychological Measurement, 17*, 105-116.

Rosen, B., & Jerdee, T. H. (1974). The influence of sex-role stereotypes on evaluations of male and female supervisory behavior. *Journal of Applied Psychology, 57*, 44-48.

Samejima, F. (1969). Estimation of latent ability using a response pattern of graded scores, *Psychometrika Monograph, 17*.

Samejima, F. (1979). *A new family of models for the multiple choice item*. Office of Naval Research Report 79-4. Knoxville, TN: University of Tennessee.

Schmitt, N., & Lappin, M. (1980). Race and sex as determinants of the means and variance of performance ratings. *Journal of Applied Psychology, 65*, 428-435.

Swaminathan, H., & Rogers, H. J. (1990). *Detecting differential item functioning using logistic regression procedures, 27*, 361-370.

Thompson, D. E., & Thompson, T. A. (1983). Task-based performance appraisal for blue-collar jobs: Evaluation of race and sex effects. *Journal of Applied Psychology, 70*, 747-753.

Van Velsor, E., & Leslie, J. B. (1991). *Feedback to managers: Vol 1. A guide to evaluating multi-rater feedback instruments*. Greensboro, NC: Center for Creative Leadership.

Whittemore, A. (1981). Sample size for logistic regression with small response probability. *Journal of the American Statistical Association, 42*, 415-427.

Yukl, G. A., (1981). Leadership in Organizations. Englewood Cliffs, New Jersey: Prentice Hall.

Table 1: Scales on the Prospector survey

Scale	Items
1. Seeks opportunities to learn	5
2. Seeks and uses feedback	5
3. Learns from mistakes	5
4. Open to criticism	3
5. Committed to making a difference	4
6. Insightful: Sees things from new angles	4
7. Has the courage to take risks	4
8. Brings out the best in people	5
9. Acts with integrity	4
10. Seeks broad business knowledge	4
11. Adapts to cultural differences	5

Table 2: Breakdown of gender and race by rater group

Rater Group	Gender		Race		
	Male	Female	White	Black	Other
Direct Report	495	257	638	62	52
Self	576	176	645	56	51
Peer	569	183	658	49	45
Boss	657	95	695	30	27

Table 3: Breakdown of gender by race

Race	Gender	
	Male	Female
White	2047	592
Black	136	61
Other	114	58

Note: N=3008

Table 4: Items that exhibited DIF attributable to rater type

Scale and Item	Text of item	Effect Size	Influence of DIF on responses
1.1	Has grown over time.	.15	B(+) to DR(-)
1.4	Has developed significant new skills over time.	.10	B(+) to DR(-)
2.4	Responds effectively when given feedback.	.13	B(+) to DR(-)
2.5	Has changed as a result of feedback.	.22	B(+) to DR(-)
6.1	Is good at identifying the most important part of a complex problem or issue.	.12	B(-) to DR(+)
6.4	Is good at asking insightful questions.	.11	B(-) to DR(+)
9.1	Can be depended on to tell the truth regardless of the circumstances.	.18	B(+) to DR(-)
9.3	Is seen by others as an honest person.	.12	B(+) to DR(-)
10.1	Has a solid understanding of our products and services.	.13	B(-) to DR(+)
10.2	Knows how the various parts of the organization fit together.	.18	B(-) to DR(+)
10.3	Knows the business.	.17	B(-) to DR(+)
10.4	Understands the financial side of the business.	.22	B(-) to DR(+)

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Jim Penny
Center for Creative Leadership
One Leadership Place
Greensboro, NC 27438-06300

Work Phone: 336-286-4442
Fax: 336-286-4434
Email: pennyj@leaders.ccl.org
Web: www.ccl.org

Paper P712

Using the SAS® System to Demonstrate the equivalence of On-line and On-paper Survey Administration across Levels of Raters

Jim Penny, Center for Creative Leadership, Greensboro, NC

ABSTRACT

This research used logistic regression to model item responses from a popular 360-for-development survey. The model used method of survey delivery and rater group to identify items that exhibited differential item functioning (DIF). The methods of survey delivery were pencil-and-paper and online by web page. The rater groups were self, boss, peer, and direct report. The sample consisted of 374 survey families where a survey family consisted of a matched set of four surveys: one self, one boss, one peer, and one direct report. Half of the survey families were from a pencil-and-paper administration; half were from an online administration. The sample contained 1496 total surveys. The procedure to flag items exhibiting differential functioning used effect size computed from Wald chi-square statistics rather than statistical significance, resulting in fewer flagged items.

The results indicated little evidence to suggest that rating differences exist due to the method of survey delivery. However, approximately 10% of the survey items exhibited differential item functioning attributable to rater group, though the effect size for each item was small. The examination of the maximum likelihood parameter estimates suggested that some of this differential functioning could be the result of hierarchical complexity. The anomalous functioning of other items was attributable to contingency theory. This research further suggested that such forms of DIF might be naturally occurring phenomena in some 360-assessment, and that the interpretation of 360-feedback may need to include the potential for this DIF to exist.

INTRODUCTION

The past decade has seen a steady increase in the use of 360-assessments for managerial development in organizations, and the use of these assessments appears to continue without abatement. One appeal of the 360-process is that it involves providing managers with feedback from four or more sources. The four primary sources include (1) the manager's boss, (2) the manager's subordinates or direct reports, (3) the peers or the customers of the target, and (4) the self. Although the concept of gathering multi-source information is not new to the field of professional development, at least one premise of multi-rater methodology remains unresolved: Does the 360-methodology provide an equivalent measure with each rater group? That is, does measurement equivalence exist between the rater groups? Without rater equivalence, the comparison of ratings, often by between-group differences, may be problematic.

In addition to the increase in the use of 360-assessment, there has been an unrelenting move from the pencil-and-paper administration of 360-surveys to electronic administration using the Internet and the World Wide Web. However, one can question the influence of delivery mode on the results of a 360-assessment asking, does 360-methodology provide an equivalent measure with each method of delivery, or is one manner of survey delivery likely to produce better, or worse, ratings than the other? Moreover, does there exist a potential interaction between rater group and delivery method? Is there a combination of survey delivery and rater group that is likely to influence ratings, and produce an aberrancy that might threaten rating equivalence? These research questions involving the measurement equivalence of 360-methodology provided the impetus for this research.

DIFFERENCES AMONG RATER GROUPS

Ratings differences among rater groups are central to the 360-assessment process and often lead to much discussion during feedback sessions (Van Velsor & Leslie, 1991). Indeed, the study of such differences informs in part the developmental planning of the individual, and, if such differences did not exist, there would be little motivation to incur the expense of 360-assessment. For example, a manager may be interpersonally skilled with peers yet cold and aloof with direct reports. Such a manager could receive high ratings on interpersonal skills from peers while receiving low ratings on this dimension from direct reports, and learning of these differences in these perceptions provides a central motivation for the 360-assessment process.

Indeed, such rating differences as these often provide rich behavioral feedback for the target manager, enabling that manager to make better choices when planning developmental activities. However, the accurate interpretation of such a rating difference requires that one can assume that each set of ratings uses the same metric. If, for whatever reason, one group of raters interprets the text of an item or a set of items differently than another group, then the resulting differences in the ratings may be the result of not only the observations of the raters but also of the interpretative difference elicited by the item.

CONTINGENCY THEORY

Contingency theories of leadership (Fiedler 1978; Fiedler & Chemers, 1982) suggest that what constitutes an appropriate response may depend on the situation in which the response is to occur, and that two substantially different responses may be appropriate for similar stimuli when the contexts of the stimuli differ. Moreover, Hershey & Blanchard (1969, 1982) and Yukl & van Fleet (1982) suggest that managers often need to change their behavior to fit particular situations, and, following this line of reasoning, managers who behave differently toward different groups of co-workers may receive disparate ratings from members of those groups. Hence, differences between the ratings received from different rater groups may be an acceptable outcome for some managers, though the accurate interpretation of such differences may require some study on the part of the feedback provider. In addition, disparate ratings arising from differential environmental contexts may provide an indication of an effective manager, and, as such, these gaps in the perspectives between groups of raters may exist as a naturally occurring and perfectly acceptable phenomenon of management.

However, if an item taps a particular situation more strongly than other items, the observed difference in ratings produced by the situational behaviors of a manager may become confounded with the propensity of that item to accentuate a situational contingency. In so doing, the observed difference in ratings is not simply the result of observed behavioral differences, but is the combined result of both a behavioral difference and the interaction of the item with the raters. As such, the interpretation of the observed difference becomes problematic because one does not know how much of the rating difference is due to a difference in observed behaviors and how much is due to the anomalous functioning of the item.

COMPLEXITY THEORY

Jacques (1996) and Jacques & Clement (1994) suggested that the degree of environmental complexity and ambiguity seen by a person within an organization generally increases with rank with direct reports seeing a simpler and more easily understood environment than bosses see. This continuity of complexity and ambiguity can produce a reality that creates substantially different experiences for bosses and direct reports and that may require

thinking by bosses that may be difficult for bosses to explain to direct reports. For example, environmental complexity could lead bosses and direct reports to give a manager substantially different ratings in areas such as understanding corporate strategy if the direct reports feel that the manager has a firm grasp of corporate strategy while the boss knows there are areas of strategy that the manager has yet to see. Hence, it seems reasonable to anticipate that environmental differences may produce rating differences given on a 360-survey; however, those rating differences are likely to be a naturally occurring phenomenon in many corporate environments.

However, it also seems reasonable to anticipate that some items on a 360-survey could be more likely than other items to tap environmental complexity. In doing so, a given item could produce ratings from direct reports that are differentially higher than the ratings given by the boss, even though the observations of the both the boss and the direct reports were likely to produce otherwise similar ratings. In such an event, the rating difference between boss and direct report raters is not only the result of behavioral differences observed by the raters but is also the result of an interaction between the item and the environment of the rater.

METHODOLOGY

This research involved the use of logistic regression to detect DIF. I made this choice over other methods such as item response theory (IRT) and factor analytic techniques for several reasons. First, I did not anticipate having a sufficient sample to compute the IRT parameter estimates for each category of rating. As well, I did not anticipate achieving sufficient multivariate normality for use with the factor analytic methods. However, my strongest reason for the choice of logistic regression was the ability to posit a model that I could evaluate in a single analysis and then interpret using much of the explanatory framework developed from linear regression. In addition, the model I built, and will explain anon, was analogous in form and function to the underlying polytomous logistic model of item response theory.

For the detection of DIF using logistic regression with polytomous data (Clauser & Mazor, 1998), I can adapt the framework of Samejima (1969, 1979) and write the equation

$$P(x = k) = \frac{1}{1 + e^{-z_{k-1}}} - \frac{1}{1 + e^{-z_k}},$$

where *P(x=k)* is the probability of a response *k* to a particular item from a respondent of standing θ, and where *k* takes the values of the Likert-type response scale (Miller & Spray, 1993; Samejima, 1969, 1979; Swaminathan & Rogers, 1990). Were the existence of DIF not an issue, I would write *z* as

$$z = \tau_0 + \tau_1\theta,$$

where θ represents the standing of the ratee on the attribute that the survey measures. I have dropped the subscript *k* to make the model easier to read. The symbols τ_0 and τ_1 represent the intercept and the slope parameters of the logistic regression model; these symbols also represent forms of the discrimination and location parameters of the Graded Response Model (Samejima (1969, 1979). This model represents the situation where the rater group membership and the survey delivery method do not influence the item response, and where the only factor that does influence the response to the item is the standing of the ratee on the attribute that the survey measures.

To expand the model to include components to represent effects due to delivery method and rater group membership, I would write

$$z = \tau_0 + \tau_1\theta + \tau_2 g + \tau_3 d,$$

where *g* and *d* represent rater group membership and delivery method, respectively, and τ_2 and τ_3 represent the logistic regression parameters for those two classifications. I can define the values for *d* as *{0,1}* where *0* would indicate survey administration by web browser and *1* would indicate administration by pencil-and-paper. Similarly, I can define the values for *g* as *{1, 2, 3, 4}* indicating *{self, direct report, peer, boss}* respectively. This model describes the instance where only uniform DIF exists.

I can expand this model to accommodate the potential existence of nonuniform DIF by the addition of two more terms to produce

$$z = \tau_0 + \tau_1\theta + \tau_2 g + \tau_3 d + \tau_4 g\theta + \tau_5 d\theta,$$

where the two new terms indicate an interaction, respectively, between (a) rater group membership and standing on the attribute that the survey measures, and (b) survey delivery method and standing. The symbols τ_4 and τ_5 represent the logistic regression parameters for these two interaction terms, respectively.

To complete the model for this research, I included two additional terms. One term is to indicate the possible interaction between rater group membership and survey delivery method; the other term is to indicate the possible three-way interaction of rater group membership, survey delivery method, and standing on the attribute that the survey measures. The types of DIF represented by these two terms are uniform and nonuniform, respectively. I can write this model as

$$z = \tau_0 + \tau_1\theta + \tau_2 g + \tau_3 d + \tau_4 g\theta + \tau_5 d\theta + \tau_6 gd + \tau_7 gd\theta,$$

where the symbols τ_6 and τ_7 represent the logistic regression parameters for these two additional interaction terms, respectively. It is this later model that I tested in this research.

I used the SAS® System to evaluate this model for the data that I collected. The SAS System produced Wald Chi Square statistics to test the null hypotheses that that the parameter estimates of τ_1 through τ_7 were statistically significantly different from *0*. It was my anticipation that τ_1 and τ_2 would routinely achieve statistical significance. It was also my anticipation that the parameter estimates of τ_3 through τ_7 would not routinely achieve statistical significance.

However, I knew that, with number of terms in the model I had chosen, I would be making quite a few statistical tests of significance, and that the experiment-wise Type I error rate could be high. To compensate for the accumulated Type I error rate that could naturally incur in this research and to avoid the complex power analysis (Hsieh, 1989; Whittemore, 1981) of logistic regression that would suggest an appropriate number of subjects to evaluate the model, I decided to use effect size instead of statistical significance. I chose to follow the lead of Penny & Johnson (1999) and to convert the Wald chi-square statistic to an effect size, *w*, described in Cohen (1988, ch. 7). The formula that relates the effect size to the sample size is

$$X^2 = nw^2$$

where X^2 is the chi-square statistic, *n* is the sample size, and *w* is the effect size. Cohen (1988, ch. 7) used the arbitrary values of .1, .3, and .5 to indicate small, medium, and large effects, respectively, and, although these values are indeed arbitrary, Penny & Johnson (1999) found that those values appeared to connote well when applied to the Mantel-Haenszel chi-square statistic. I used these three values to define four effect ranges that I would use to categorize the DIF that I discovered with the logistic regression model. These ranges were nil-to-small, small-to-medium, medium-to-large, and large-to-extreme.

DATA

I used the 1999 Benchmarks® database from the Center for Creative Leadership (CCL) for this research. Benchmarks is a 360-degree assessment-for-development feedback instrument developed by CCL researchers who conducted extensive research exploring the experiences that promote managers' development (Lindsey, Homes, & McCall, 1987; McCall & Lombardo, 1983; McCall, Lombardo, & Morrison, 1988). CCL researchers initially developed Benchmarks as a pencil-and-paper survey; it became available over the web in 1999. The instrument consists of three sections designed to assess skills and perspectives related to managerial effectiveness. I examined only one section of scales in this research. Table 1 presents the

names and descriptions of the 16 scales. All 106 items used a 5-point, Likert-type response format. The textual anchors for the scale were 1="Not at all," 2="To a little extent," 3="To some extent." 4="To a great extent," and 5="To a very great extent."

I selected a random sample of 374 survey families from the 1999 Benchmarks® database of the Center for Creative Leadership for use in this research. Each survey family included matched ratings from a manager (the self or ratee), a peer, a boss, and a direct report resulting in 1496 total surveys. Half of the families (*N*=187) were pencil-and paper-surveys; the other half was online surveys. Because each family contained exactly one survey from each rater group, there were 187 each of self, peer, boss, and direct report surveys.

I made no selection using any other demographic variables, so the sample reflected the demographic composition of the parent population, and was comprised of approximately 61% male ratees. Moreover, the sample contained approximately 90% white and 8% black ratees. In addition, about 49% held bachelor's degrees, 31% held master's degrees, and about 15% held doctoral or professional degrees. These data represented a broad range of organizations such as governmental agencies, manufacturing companies, and educational institutions. About 91% of the ratees held middle to upper level positions as managers.

RESULTS

THE INFLUENCE OF STANDING ON RESPONSES

The main effect of standing on the attribute measured by the survey had by far the greatest influence on the item ratings of all the terms in the logistic model. The average effect was .48 with a standard deviation of .08; the range was from .25 to .61. In most instances, the effect was in the medium-to-large and large-to-extreme categories.

DIF ATTRITUBLE TO DELIVERY METHOD

These data provided no evidence of any substantial uniform or nonuniform DIF attributable to the manner in which the raters completed the survey. The mean effect of delivery mode was .02 with a standard deviation of .02 and a range from .00 to .07. In every instance, the effect was in the nil-to-small category.

DIF ATTRIBUTABLE TO RATER SOURCE

Ten items on the survey exhibited uniform DIF attributable to the rater group. Table 2 presents these items along with the effect size, the apparent explanatory theory, and the impact of the DIF on the ratings. These ten items produced effects that barely made the cut-off, .1, for a small effect. Complexity theory could explain the DIF found in six of the ten items. The differential functioning produced lower ratings from the direct reports in half of these six items.

Contingency theory explained the differential functioning found in the other four items exhibiting uniform DIF. The effect associated with these items was also small. Three of these items produced differential functioning that contrasted the ratings of direct reports with those from all other groups. With these three items, the differential functioning produced higher ratings from the direct reports. The other item that appeared to tap into contingency theory contrasted the peer ratings with those from all other groups. For this single item, the differential functioning produced reduced ratings from the peers.

Only one item on the survey exhibited nonuniform DIF. The effect associated with this item is .11, which is still a small value. This item appeared to tap into complexity theory, and the resultant DIF produced lower ratings from direct reports and higher ratings from bosses; however, the nature of nonuniform DIF suggests that the direction of the differential functioning may reverse for some range of the covariate.

DIF ATTRITUBLE TO DELIVERY METHOD BY RATER SOURCE INTERACTION

This research uncovered no items that exhibited DIF attributable to an interaction between rater source and delivery method.

DISCUSSION

In most studies of differential item functioning, the existence of DIF is not something that is good. Rather, the occurrence of DIF may threaten the validity of an assessment, and reduce the usefulness of the survey (Lord, 1980; Penny & Johnson, 1999). I tend to concur with those who suggest that DIF is a quantity to remove from an assessment. Psychological measures that assess the learning, the performance, or perhaps even the potential, of a candidate for a higher appointment are not the places where one generally wants to find measures influenced, even in part, by demographic quantities such as gender, race, nationality, or native language.

However, this study, suggests that the existence of differential item functioning may not be a completely bad thing, and further suggests that the type and degree of DIF found might be the result of two naturally occurring phenomena in the workplace. First, the differential environmental complexity that can exist in the workplace may produce unavoidable, if not expectable, differential functioning. Second, DIF could also be the natural result of the different behaviors that a manager can present when interacting with different groups of people. Hence, one could argue that the interpretation of 360-feedback reports should take into account the possibility of anomalous item functioning produced by such environmental and experiential differences. Moreover, this anomalous functioning may function to ameliorate, or exacerbate, some observed rating differences, and the degree of either may depend, at least in part, on the standing of the ratee on the trait measured by the survey.

For example, the first scale contained 17 items, 3 of which exhibited DIF. I computed scale scores for the complete scale, and compared those values with the scale scores computed using the 14 items that did not exhibit DIF. The mean difference (and standard deviation of the difference) of those scale scores was {.12(.19), .04(.17), .10(.18), .03(.16)} for direct report, self, peer, and boss raters respectively. These differences, though statistically significant, do not appear to be of practical significance, conforming to the prior finding of small effect sizes.

One could argue that a manager receiving 360-feedback could benefit from knowing the degree to which environmental complexity influences particular feedback. With such a breakdown, the manager could better understand how much of a given rating arose from the observations of the boss, and how much arose from the different environment from which the boss interpreted not only the behaviors of the manager but also the text of the item. It is conceivable, then, that from such feedback a manager could at once choose the developmental areas on which to work for improved performance while learning more about the world as seen by the boss, a world in which the manager may soon work. Unfortunately, measurement theory does not yet permit the reliable partitioning of ratings that would allow the manager to understand the rating component due to behavior and the component due to environment, and additional theoretical research will be necessary to produce the methodology that will permit a breakdown of ratings into components.

REFERENCES

Clauser, B. E., & Mazor, K. M. (1998). Using statistical procedures to identify differentially functioning test items. *Educational Measurement: Issues and Practices, 2*, 31-44.

Cohen, J. (1988). *Statistical power analysis for the behavioral sciences*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Fiedler, F.E. (1978). The contingency model and the

dynamics of the leadership process, In *Advances in experimental social psychology*, ed. L. Berowitz. New York: Academic Press.

Fiedler, F.E. & Chemers, M.M. (1982). *Improving leadership effectiveness: The leader match concept*. 2nd Ed. New York: Wiley.

Hershey, P., & Blanchard, K. H. (1969). Life cycle theory of leadership. *Training and Development Journal*, *23*, 26-34.

Hershey, P., & Blanchard, K. H. (1982). Management of Organizational Behavior, (4th ed.). Englewood Cliffs, NJ: Prentice Hall.

Hsieh, F. Y. (1989). Sample size tables for logistic regression. *Statistics in Medicine, 8*, 795-802.

Jacques, E. (1996). *Requisite organization: A total system for effective managerial organization and managerial leadership for the 21st century*. Cambridge, MA: Cason Hall & Co.

Jacques, E., & Clement, S. D. (1994). *Executive leadership: A practical guide to managing complexity*. Cambridge, MA: Basil Blackwell.

Lindsey, E., Homes, V., & McCall, M.W., Jr. (1987). *Key events in executives' lives* (Tech. Rep. No. 32). Greensboro, NC: Center for Creative Leadership.

Lord, F. (1980). *Application of item response theory to practical testing problems*. Hillsdale, NJ: Erlbaum.

McCall, M.W., Jr., & Lombardo, M.M. (1983, February). What makes a top executive? *Psychology Today*, 26-31.

McCall, M.W., Jr., Lombardo, M.M., & Morrison, A.M. (1988). *The lessons of experience: How successful executives develop on the job.* Lexington, MA: Lexington Books.

Miller, T. R., & Spray, J. A. (1993). Logistic discriminant function analysis for DIF identification of polytomously scored items. *Journal of Educational Measurement, 30*, 107-122.

Penny, J., & Johnson, R. L. (1999). How group differences in matching criterion distribution and IRT item difficulty can influence the magnitude of the Mantel-Haenszel chi-square DIF index. *The Journal of Experimental Education*, *67*, 343-366.

Samejima, F. (1969). Estimation of latent ability using a response pattern of graded scores, *Psychometrika Monograph, 17*.

Samejima, F. (1979). *A new family of models for the multiple choice item*. Office of Naval Research Report 79-4. Knoxville, TN: University of Tennessee.

Swaminathan, H., & Rogers, H. J. (1990). Detecting differential item functioning using logistic regression procedure. *Journal of Educational Measurement, 27*, 361-370.

Van Velsor, E., & Leslie, J. B. (1991). *Feedback to managers: Vol 1. A guide to evaluating multi-rater feedback instruments*. Greensboro, NC: Center for Creative Leadership.

Whittemore, A. (1981). Sample size for logistic regression with small response probability. *Journal of the American Statistical Association, 42*, 415-427.

Yukl, G. A., & van Fleet, D. (1982). Cross-situational, multi-method research on military leader effectiveness. *Organizational Behavior and Human Performance*, *30*, 87-108.

Table 1. Name and description of scales on the Benchmarks survey

Scale	Items	Description
1. Resourcefulness	17	Can think strategically, engage in flexible problem-solving behavior, and work effectively with higher management.
2. Doing Whatever It Takes	14	Has perseverance and focus in the face of obstacles.
3. Being a Quick Study	4	Quickly masters new technical and business knowledge.
4. Decisiveness	4	Prefers quick and approximate actions to slow and precise one in many management situations.
5. Leading Employees	13	Delegates to employees effectively, broadens their opportunities, and acts with fairness towards them.
6. Setting a Developmental Climate	5	Provides a challenging climate to encourage employees' development.
7. Confronting Problem Employees	4	Acts decisively and with fairness when dealing with problem employees.
8. Work Team Orientation	4	Accomplishes tasks though managing others.
9. Hiring Talented Staff	3	Hires talented people for his or her team.
10. Building and Mending Relationships	11	Knows how to build and maintain working relationships with coworkers and external parties.
11. Compassion and Sensitivity	4	Shows genuine interest in others and sensitivity to subordinates' needs.
12. Straightforwardness and Composure	6	Is honorable and steadfast.
13. Balance between Personal Life and Work	4	Balances work priorities with personal life so that neither is neglected.
14. Self-awareness	4	Has an accurate picture of strengths and weaknesses and is willing to improve.
15. Putting People at Ease	4	Displays warmth and a good sense of humor.
16. Acting with Flexibility	5	Can behave in ways that are often seen as opposites.

Table 2. Description of the items that exhibited DIF and the type of DIF observed

Item text with scale and item number	Type of DIF	Effect	Explanatory Theory	Impact of DIF
1.6 Understands higher management values, how Higher management operates, and how they see things.	Uniform	.10	Complexity	Dr (-) to Boss (+)
1.10 Does his/her homework before making a proposal to top management	Uniform	.10	Contingency	DR (+) vs. all others
1.13 Establishes effective management practices for directing employees he/she sees only twice a month.	Uniform	.11	Contingency	DR (+) vs. all others
8.4 Focuses more on managing other people to accomplish a task than on personally finishing everything the work group does.	Uniform	.11	Complexity	DR (+) to Boss(-)
9.3 Surrounds him/herself with the best people	Uniform	.11	Complexity	DR (+) to Boss (-)
10.6 When working with peers from other functions or units, gains their cooperation and support	Uniform	.11	Contingency	DR (+) vs all others
11.3 Is sensitive to signs of overwork in others.	Uniform	.12	Complexity	DR(-) to Boss (+)
12.2 (Does not) rely on style more than substance in dealing with top management.	Uniform	.11	Contingency	Peer (-) vs. all others
2.11 Accepts conflicts as inevitable and does not shy away from them.	Uniform	.13	Complexity	DR (+) to Boss (-)
4.1 Displays a real bias for action, calculated risks, and quick decisions.	Uniform	.11	Complexity	DR (-) to Boss (+)
4.4 Does not hesitate when making decisions.	Nonuniform	.11	Complexity	DR (-) to Boss (+)

Note: An effect between .10 and .30 is a small-to-medium effect. The sign beside the rater group name indicates the direction in which the DIF changed the ratings.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Jim Penny
Center for Creative Leadership
One Leadership Place
Greensboro, NC 27438-06300
Work Phone: 336-286-4442
Fax: 336-286-4434
Email: pennyj@leaders.ccl.org
Web: www.ccl.org

Paper #P713

Using SAS® To Control Multistream Binomial Processes

Peter Wludyka, University of North Florida, Jacksonville, FL & Sheri Jacobs of Vistakon, Inc.

ABSTRACT

Process control schemes for multistream binomial processes are described. A multistream process is one in which the streams are uncorrelated or very weakly correlated. When each of the *J* streams are identical (a homogeneous multistream process) one can use a group control chart (a *k*-sigma *p*-chart on which only the largest and smallest of the J sample proportions nonconforming are plotted). A test (runs rule) that signals when the same stream produces the largest proportion nonconforming *R* consecutive times is presented along with two SAS® programs useful for designing runs rule schemes, p-charts, and Chi-Squared control charts.

INTRODUCTION

A multistream process is best introduced by means of an example. Consider an injection molding process that makes a plastic toy. Tubes running from a reservoir carry liquid plastic into molds in an apparatus (a frame) which contains four molds. Suppose for simplicity that one tube is used to fill each toy. Each mold location can be considered to be a "stream" and this will be referred to as a 4-stream process. After cooling and separation from the mold a toy can be inspected and classified as defective (nonconforming) or nondefective (conforming). A process quality control scheme is a methodology for determine whether the process is producing "conforming units" in a stable and predictable manner; that is, one may decide whether the process is in-control or out-of-control. The control schemes described here require that periodically samples of product be inspected. One might be concerned with two ways in which the process could cease to be in-control.

1. A process change that affects all of the streams
2. A change that impacts on one or more streams

The first might correspond with to a poor grade of plastic being used; the second to a clog or other problem with an individual tube.

Note that identifying which type of out-of-control condition occurs will be useful for bringing the process back under control.

PROCESS MODELING AND CONTROL CHARTS

Control charts can be used to control processes (see Montgomery). In order to use control charts a statistical model for the process is required. Suppose that n frames are selected for inspection at the end of each epoch. Then each stream will be sampled *n* times also. If the streams are uncorrelated then the process is called a *j*-stream multistream process. A plausible process model then is that each stream is a binomial process with *n* trials and success probability *p*, where *p* is the proportion of items in the process (stream) that are nonconforming. When *p* is the same for all the streams the process is a homogeneous multistream process. (See Jacobs and Wludyka for details). The process is out-of-control if

1. The overall *p* changes
2. *p* changes for one or more streams

A frequently used tool for detecting changes in a binomial process is the *p*-chart (see Montgomery). The sample proportion nonconforming is plotted on the chart at each epoch and a signal occurs whenever the sample proportion plots outside the *k*-sigma control limits. Typically *k* = *3* since experience has shown this to be a useful control band width. For the multistream process one may

1. construct a single overall p-chart
2. Construct J individual p-charts (one for each stream).
3. Use a Chi-squared chart

In (2) it is unwise to set *k* = *3* since the false alarm rate will usually be much higher than most quality engineers would find reasonable. That is, *k* should be greater than three. See Jacobs and Wludyka for methods for determining the proper *k*. Note that is it sufficient to plot only the largest and smallest of the *J* sample (stream) proportions on a single chart for (2). This chart is called a group p-chart. The Chi-squared chart will not be described in this paper; however, the simulation program provided produces estimated ARLs (average run lengths) for this chart.

RUNS RULES

A runs rule is a simple tool for determining whether one (or more) of the streams have shifted (*p* has changed). At each epoch the stream for which the proportion nonconforming is largest is determined. This will be used to produce a one-sided runs rule. This has the advantage of being simple to use. When the same stream has the largest proportion nonconforming for *r* consecutive times that will be defined to be a run of length *r*. When *r* equals some constant *R* (chosen by the user) then an out-of-control signal occurs. *R* is chosen such that the in-control process will have a large average run length (ARL). The ARL is defined as the average number of samples until a signal occurs. When the process is out-of-control a short ARL is desired. Since ties can occur for the stream with the highest sample proportion nonconforming there are several ways in which a "run" can be defined. Four rules are:

1. The run is maintained on a tie only if the run stream is included in the tie
2. The run length is maintained on all ties
3. The run continues (is increased) on a tie if the run stream is in the tie
4. The run continues on a tie that includes the run stream and is maintained on a tie that does not include the run stream.

Each of these has a variant in which the run starts over whenever there is a j-way tie at zero, which is called a zero restart rule. These rules are denoted Rule 1a, …, Rule 4a.

A RUNS RULE ARL PROGRAM

Given the process proportion nonconforming, *p*, one can then choose values for *n* and *R* that produce ARLs that have suitable properties. Note that *p* is usually estimated from in-control process data. The program below produces a series of tables that aid the user the selecting a control scheme.

```
/*  ARLs for Runs Rules */
data rule2mat;
j=4; /* the number of streams */
do p0=0.023; /* proportion nonconforming */
do n=75; /* sample size per stream */
do idelta=0.0, .01, .02,.03,.04, .05,.08, .10;
 delta= idelta;  /* prpcess shift vales */
p1=p0+delta; p2=p0;
alltie =
probbnml(p1,n,0)*(probbnml(p2,n,0))**(j-1);
/* calculate t1  */
q1 = 0; q2=0;
t1 = 0; t2in1=0;t2out1=0; t2=0;
```

```
do i=1 to n;
        py2eqi = probbnml(p2,n,i)-
probbnml(p2,n,i-1);
        py2lti = probbnml(p2,n,i-1);
        py1lti = probbnml(p1,n,i-1);
        py1eqi = probbnml(p1,n,i)-
probbnml(p1,n,i-1);
        t1wstep = py2eqi**2*py2lti**(j-
3)*py1lti;
        t2in1stp = py2eqi*py2lti**(j-2)*py1eqi;
        t2ot1stp = py2eqi**2*py2lti**(j-
3)*py1lti;
        q1 = q1+ py1eqi*py2lti**(j-1);
        q2 = q2 + py2eqi*py2lti**(j-2)*py1lti;
        t = 1-(q1+(j-1)*q2);
        alltie = alltie + py1eqi*py2eqi**(j-1);
        do w = 2 to j-1;
                c1 =
gamma(j)/(gamma(w+1)*gamma(j-w));
                c2 = gamma(j-
1)/(gamma(w)*gamma(j-w));
                if w = j-1 then c3 = 0;
                else c3 = gamma(j-
1)/(gamma(w+1)*gamma(j-w-1));

        if w = 2 then do;
                t1 = t1+c1*t1wstep;
                t2in1 = t2in1 + c2*t2in1stp;
                t2out1 = t2out1 + c3*t2ot1stp;
                t2 = t2 + c2*t2in1stp +
c3*t2ot1stp;
                end;
        else do;
                t1wstep =
t1wstep*(py2eqi/py2lti);
                t1 = t1+c1*t1wstep;
                t2in1stp =
t2in1stp*(py2eqi/py2lti);
                t2in1 = t2in1 + c2*t2in1stp;
                t2 = t2 +  c2*t2in1stp;

                    t2ot1stp =
t2ot1stp*(py2eqi/py2lti);
                    t2out1 = t2out1 +
c3*t2ot1stp;
                    t2 = t2 + c3*t2ot1stp ;

        end;
         /*  output; */
        end;
end;
z0 = (1-p2)**((j-1)*n)*(1-p1)**n;
output;
end; end; end;
proc print;
        var j n p0 delta q1 q2 t1 t2 t z0;
run;

data probs;
        set rule2mat;
        keep j n q1 q2 t t1 t2 p0 delta z0;
/* arl routine for rule 1 follows */
proc iml;
        use probs;
        /* show datasets; show contents;*/
        labtab = J(1,3,0);
        ptab = J(8,8,0); ptaba = J(8,8,0);
        do step = 1 to 8; /* print step;*/
        read next;/* list;*/
        do instep= 2 to 8; r=instep; /* print r;
*/
        /* print j t1 t2 t q1 p0 ; */
        P=J(2*r-1,2*r-1,0);
        /* print P;*/
        if r > 2 then do;
                do row = 4 to 2*r-1 by 2;

p[row,1]=t1;p[row,2]=0;p[row,3]=(J-
1)*q2;p[row,row]=t-t1;if row < 2*r-2 then
p[row,row+2]=q1;

p[row+1,1]=t2;p[row+1,2]=q1;p[row+1,3]=(J-
2)*q2;p[row+1,row+1]=t-t2; if row < 2*r-2 then
p[row+1,row+3]=q2;
                end;end;
        p[1,1]=t;   p[1,2]=q1;     p[1,3]=(J-
1)*q2;
        p[2,1]=t1;  p[2,2]=t-t1;   p[2,3]=(J-
1)*q2;   if r > 2 then p[2,4]=q1;
        p[3,1]=t2;  p[3,2]=q1;     p[3,3]=t-
t2+(J-2)*q2;  if r > 2 then      p[3,5]=q2;
        /* print P;*/
        Imat=I(2*r-1); /* print Imat;*/
        one=J(2*r-1,1,1); /* print one;*/
        arl=inv(Imat-P)*one;
        ptab[step,1]=delta;ptab[step,r]=arl[1];
        /* print 'arl using rule 1';
        print arl; */
        pzero = p;
        do rstep = 2 to 2*r-1;
                pzero[rstep,1]=p[rstep,1]+z0;

pzero[rstep,rstep]=p[rstep,rstep]-z0;
        end;
        arlz = inv(Imat-pzero)*one;

ptaba[step,1]=delta;ptaba[step,r]=arlz[1];
        end;end;
        print 'arl table for rule 1';
        tag1 = {J n p0};

labtab[1,1]=j;labtab[1,2]=n;labtab[1,3]=p0;
        print labtab[colname=tag1];
        tag2= {delta r2ARL r3ARL r4ARL r5ARL
r6ARL r7ARL r8ARL};
        print ptab[colname=tag2 format = 10.2];
        print 'ARL Table for Rule 1a';
        print ptaba[colname=tag2 format = 10.2];

/* arl routine for rule 2 follows */
```

```
proc iml;
         use probs;
        /* show datasets; show contents;*/
        labtab = J(1,3,0);
        ptab = J(8,8,0); ptaba = J(8,8,0);
        do step = 1 to 8; /* print step;*/
        read next;/* list;*/
        do instep= 2 to 8; r=instep; /* print r;
*/
        /* print j t1 t2 t q1 p0 ; */
        P=J(2*r-1,2*r-1,0);
        /* print P;*/
        if r > 2 then do;
                do row = 4 to 2*r-1 by 2;

p[row,1]=0;p[row,2]=0;p[row,3]=(J-
1)*q2;p[row,row]=t;if row < 2*r-2 then
p[row,row+2]=q1;

p[row+1,1]=0;p[row+1,2]=q1;p[row+1,3]=(J-
2)*q2;p[row+1,row+1]=t; if row < 2*r-2 then
p[row+1,row+3]=q2;
                end;end;
        p[1,1]=t;  p[1,2]=q1;   p[1,3]=(J-1)*q2;
        p[2,1]=0;  p[2,2]=t;    p[2,3]=(J-1)*q2;
if r > 2 then p[2,4]=q1;
        p[3,1]=0;  p[3,2]=q1;   p[3,3]=t+(J-
2)*q2; if r > 2 then p[3,5]=q2;
        /* print P;*/
        Imat=I(2*r-1); /* print Imat;*/
        one=J(2*r-1,1,1); /* print one;*/
        arl=inv(Imat-P)*one;
        ptab[step,1]=delta;ptab[step,r]=arl[1];
        /* print 'arl using rule 2';
        print arl; */
        pzero = p;
        do rstep = 2 to 2*r-1;
                pzero[rstep,1]=p[rstep,1]+z0;

pzero[rstep,rstep]=p[rstep,rstep]-z0;
        end;
        arlz = inv(Imat-pzero)*one;

ptaba[step,1]=delta;ptaba[step,r]=arlz[1];
        end;end;
        print 'arl table for rule 2';
        tag1 = {J n p0};

labtab[1,1]=j;labtab[1,2]=n;labtab[1,3]=p0;
        print labtab[colname=tag1];
        tag2= {delta r2ARL r3ARL r4ARL r5ARL
r6ARL r7ARL r8ARL};
        print ptab[colname=tag2 format=10.2];
        print 'ARL Table for Rule 2a';
        print ptaba[colname=tag2 format = 10.2];
/* arl routine for rule 3 follows */
        proc iml;
        use probs;
        /* show datasets; show contents;*/
        labtab = J(1,3,0);
        ptab = J(8,8,0); ptaba = J(8,8,0);
        do step = 1 to 8; /* print step;*/
        read next;/* list;*/
        do instep= 2 to 8; r=instep; /* print r;
*/
        /* print j t1 t2 t q1 p0 ; */
        P=J(2*r-1,2*r-1,0);
        /* print P;*/
        if r > 2 then do;
          do row = 4 to 2*r-1 by 2;
          p[row,1]=t1;  p[row,2]=0;
p[row,3]=(J-1)*q2;   if row < 2*r-2 then
p[row,row+2]=q1+t-t1;

p[row+1,1]=t2;p[row+1,2]=q1;p[row+1,3]=(J-2)*q2;
if row < 2*r-2 then p[row+1,row+3]=q2+t-t2;
                end;end;
        p[1,1]=t;  p[1,2]=q1;   p[1,3]=(J-1)*q2;
        p[2,1]=t1; p[2,2]=0;    p[2,3]=(J-1)*q2;
if r > 2 then     p[2,4]=q1+t-t1;
        p[3,1]=t2; p[3,2]=q1;   p[3,3]=(J-2)*q2;
if r > 2 then                p[3,5]=q2+t-t2;
        /* print P;*/
        Imat=I(2*r-1); /* print Imat;*/
        one=J(2*r-1,1,1); /* print one;*/
        arl=inv(Imat-P)*one;
        ptab[step,1]=delta;ptab[step,r]=arl[1];
        /* print 'arl using rule 3';
        print arl; */
        pzero = p;
        do rstep = 2 to 2*r-1;
                pzero[rstep,1]=p[rstep,1]+z0;

pzero[rstep,rstep]=p[rstep,rstep]-z0;
        end;
        arlz = inv(Imat-pzero)*one;

ptaba[step,1]=delta;ptaba[step,r]=arlz[1];
        end;end;
        print 'arl table for rule 3';
        tag1 = {J n p0};

labtab[1,1]=j;labtab[1,2]=n;labtab[1,3]=p0;
        print labtab[colname=tag1];
        tag2= {delta r2ARL r3ARL r4ARL r5ARL
r6ARL r7ARL r8ARL};
        print ptab[colname=tag2 format=10.2];
        print 'ARL Table for Rule 3a';
        print ptaba[colname=tag2 format = 10.2];
/* arl routine for rule 4 follows */
proc iml;
       proc iml;
         use probs;
        /* show datasets; show contents;*/
        labtab = J(1,3,0);
        ptab = J(8,8,0); ptaba = J(8,8,0);
        do step = 1 to 8; /* print step;*/
        read next;/* list;*/
        do instep= 2 to 8; r=instep; /* print r;
*/
```

```
        /* print j t1 t2 t q1 p0 ; */
        P=J(2*r-1,2*r-1,0);
        /* print P;*/
        if r > 2 then do;
                do row = 4 to 2*r-1 by 2;

p[row,1]=0;p[row,2]=0;p[row,3]=(J-
1)*q2;p[row,row]=t1;if row < 2*r-2 then
p[row,row+2]=q1+t-t1;

p[row+1,1]=0;p[row+1,2]=q1;p[row+1,3]=(J-
2)*q2;p[row+1,row+1]=t2; if row < 2*r-2 then
p[row+1,row+3]=q2+t-t2;
                end;end;
        p[1,1]=t;p[1,2]=q1;p[1,3]=(J-1)*q2;
        p[2,1]=0;p[2,2]=t1;p[2,3]=(J-1)*q2;   if
r > 2 then p[2,4]=q1+t-t1;
        p[3,1]=0;p[3,2]=q1;p[3,3]=t2+(J-2)*q2;if
r > 2 then p[3,5]=q2+t-t2;
        /* print P;*/
        Imat=I(2*r-1); /* print Imat;*/
        one=J(2*r-1,1,1); /* print one;*/
        arl=inv(Imat-P)*one;
        ptab[step,1]=delta;ptab[step,r]=arl[1];
        /* print 'arl using rule 2';
        print arl; */
        pzero = p;
        do rstep = 2 to 2*r-1;
                pzero[rstep,1]=p[rstep,1]+z0;

pzero[rstep,rstep]=p[rstep,rstep]-z0;
        end;
        arlz = inv(Imat-pzero)*one;

ptaba[step,1]=delta;ptaba[step,r]=arlz[1];
        end;end;
        print 'arl table for rule 4';
        tag1 = {J n p0};

labtab[1,1]=j;labtab[1,2]=n;labtab[1,3]=p0;
        print labtab[colname=tag1];
        tag2= {delta r2ARL r3ARL r4ARL r5ARL
r6ARL r7ARL r8ARL};
        print ptab[colname=tag2 format=10.2];
        print 'ARL Table for Rule 4a';
        print ptaba[colname=tag2 format = 10.2];
```

Program output and interpretation

A series of tables are produced which can be used to pick *R* and *n*.

Table 1: Runs Rule ARL Output

arl table for rule 1

LABTAB	J	N	P0
	4	75	0.023

DELTA	R4ARL	R5ARL	R6ARL
0.00	195.01	926.44	4395.88
0.01	73.95	200.92	522.68
0.02	23.50	43.29	77.56
0.03	11.74	18.04	26.83
0.04	7.77	10.90	14.72
0.05	6.05	8.07	10.36
0.08	4.41	5.59	6.79
0.10	4.15	5.20	6.27

These are partial outputs; that is those for Rule1 (Table 1) and Rule 2 (Table 2) for selected values of *R*. Continuing with the TOY process, we have *J = 4* streams in which the in-control proportion nonconforming is estimated to be 0.023. If one takes samples of size n = 75, then the in-control ARL using Rule 1 is 926.44 with *R* = 5 and the in-control ARL for Rule 2 is 492.71 with *R* = 5 (note that in-control corresponds to delta = 0). If the process shifts upward by 0.03 for a single stream (so that the stream now produces 5.3% nonconforming) then using Rule 1 it will take on average 18 samples to detect the shift and 16.32 samples using Rule 2.

Table 2: Runs Rule ARLs for Rule 2

arl table for rule 2

LABTAB	J	N	P0
	4	75	0.023

DELTA	R4ARL	R5ARL	R6ARL
0.00	122.82	492.71	1972.28
0.01	55.46	137.12	322.56
0.02	20.37	35.57	60.14
0.03	10.91	16.32	23.57
0.04	7.49	10.36	13.78
0.05	5.93	7.86	10.01
0.08	4.40	5.57	6.76
0.10	4.14	5.20	6.26

A quality engineer might on this basis decide to use Rule 2 with *R* = 5; or, decide to examine other (larger) sample sizes to increase the power of the detection scheme. For example, increasing the sample size to 150 reduces the ARL for a shift of 0.03 to about 9 for each of the rules examined.

Given *J*, *p*, and *n* the program outputs ARLs for R =2, …,8 (and can be easily modified to larger values of R) for each of the eight rules so that an engineer can examine several scenarios before choosing a rule.

PROCESS SHIFTS

Runs rules are of no value for detecting an overall process shift. For that reason it is often wise to use the runs rule in conjunction with another tool, such as a process *p*-chart, in which the stream data is aggregated to form a single *p*-chart. It is also possible to use a Chi-Squared chart with a runs rule or a group *p*-chart (see Jacobs and Wludyka for a thorough explanation). Using a control chart along with a runs rule will decrease all ARLs, especially the in-control ARL., so care must be used when employing such a control scheme.
The advantage of a dual scheme is that both stream shifts and overall shifts are more easily detected.

If one plans to use a *p*-chart with a runs rule it is probably wise to choose a Runs Rule with an in-control ARL of 500 or more. Then using a *p*-chart with *k = 3* will probably likely produce a scheme with in-control ARL of about 225.

SIMULATION TOOLS FOR CONTROL SCHEMES

Monte Carlo methods can be used to estimate the ARLs associated with a complex control scheme.

Program for Runs Rule with a P-chart ARLs

```
/*********************************************
*********************
*        simulates ARLs: Runs rule 2 (maintain
run on ties) + p-chart
*        overal p-chart k = (user's choice)
*        group p-charts k1,k2 = equated
*        ChiSquare chart based on nominal ARL0
(user specified)
*        Default: case 1 to case 2
(ALPHA{1}=DELTAJ and DELTA + 0)
*        Modify: case 1 to case 1 ( ALPHA{1} = 0
and DELTA=DELTAJ)
*        signal(1) = Overall p-chart
*        signal(2) = pj
*        signal(3) = chi-square overall
*        signal(4) = chi-square for j-columns;
*        signal 10 - 20 for individual pj
*        signal(13) = chisq overall
*        signal(14) = chisq for j cols
*********************************************
*********************/
data simdat1;
        do jstrs=3;  /*users list for loop*/
        do n=50; /*user list for loop*/
 /* determine control limits  */
        p0=0.10;
        inum=1;jnum=jstrs;
        kovp = 3.0;
        ovp = p0;
        ovpucl = min(1,ovp + kovp*sqrt((ovp*(1-
ovp))/(inum*jnum*n)));
        ovplcl = max(0, ovp - kovp*sqrt((ovp*(1-
ovp))/(inum*jnum*n)));
        ovpnucl =int(n*inum*jnum*( min(1,ovp +
kovp*sqrt((ovp*(1-ovp))/(inum*jnum*n)))));
        ovpnlcl =int(n*jnum*inum*( max(0, ovp -
kovp*sqrt((ovp*(1-ovp))/(inum*jnum*n)))));
        /*  determine k1 and k2 by equating arl0
to p-chart */
uclpx=(ovp+kovp*sqrt((ovp*(1-
ovp))/(Jnum*n)))*(Jnum*n);
lclpx=(ovp-kovp*sqrt((ovp*(1-
ovp))/(Jnum*n)))*(Jnum*n);
if lclpx <=0 then powp0=1-
probbnml(ovp,Jnum*n,floor(uclpx));
else powp0 = 1-probbnml(ovp,Jnum*n,
floor(uclpx))+probbnml(ovp,Jnum*n,floor(lclpx));
        /*     */
pj0=p0;
k1=2.9;
do until (diff < 0 or k1 > 6);
k1=k1+.01;
uclpjx=(pj0+k1*sqrt((pj0*(1-pj0))/(n)))*(n);
lclpjx=(pj0-k1*sqrt((pj0*(1-pj0))/(n)))*(n);
if lclpjx <=0 then powpj0=1-
probbnml(pj0,n,floor(uclpjx));
else powpj0 = 1-probbnml(pj0,n,
floor(uclpjx))+probbnml(pj0,n,floor(lclpjx));
powpj0 = 1-(1-powpj0)**jnum;
diff = powpj0 - powp0;
end;
k2=k1-.01;
uclpjx2=(pj0+k2*sqrt((pj0*(1-pj0))/(n)))*(n);
lclpjx2=(pj0-k2*sqrt((pj0*(1-pj0))/(n)))*(n);
        /*
put 'powp0 ='powp0  '    powpj0= 'powpj0;
        */
        /*  end k1 and k2 */
        kpj = k1;
        pj0 = p0;
        pjucl = min(1,pj0 + kpj*sqrt((pj0*(1-
pj0))/(inum*n)));
        pjlcl = max(0,pj0 - kpj*sqrt((pj0*(1-
pj0))/(inum*n)));
        pjnucl =int( min(1,pj0 +
kpj*sqrt((pj0*(1-pj0))/(inum*n))));
        pjnlcl = int( max(0,pj0 -
kpj*sqrt((pj0*(1-pj0))/(inum*n))));
        kpj=k2;
        pjucl2 = min(1,pj0 + kpj*sqrt((pj0*(1-
pj0))/(inum*n)));
        pjlcl2 = max(0,pj0 - kpj*sqrt((pj0*(1-
pj0))/(inum*n)));
        pjnucl2 =int( min(1,pj0 +
kpj*sqrt((pj0*(1-pj0))/(inum*n))));
        pjnlcl2 = int( max(0,pj0 -
kpj*sqrt((pj0*(1-pj0))/(inum*n))));
        ncsarl = 800;
        csqcl = cinv(1-(1/ncsarl),inum*jnum);
        csqclq = cinv(1-(1/ncsarl),jnum);
        do deltaj  = 0.0,0.01,0.02,0.05,0.10,.2;
        array tsig(30);
    do init = 1 to 30;
```

```
        tsig(init)=0;
end;
seed1 = -1;
numreps = 1000;
totr=0; count=0;repct=0;
do reps = 1 to numreps;

        R=5;
        if jstrs = 3 then r = 6;
        if (jstrs = 3 and n=20 and p0=.01) then
r = 5;
        if jstrs = 4 then r=5;
        if jstrs = 5 then r=4;
                if (jstrs=5 and n=100 and p0 >
.05)  then r=5;
                if (jstrs=5 and n=200 and p0 >
.01) then r=5;

        if jstrs = 8 then r=4;
        if jstrs = 10 then r=4;
                if (jstrs = 10 and n = 20 ) then
r= 3;
                if (jstrs = 10 and n = 50 and p0
= .01) then r=3;
        if jstrs > 10 then r = 3;
        totr=totr+count;
        /*
        put 'count =' count totr;
        */
        count =0;
        rsig=0;
        rlength=1;
        prevmaxj=-1;

        do until ( rsig = 1 or count > 3000 );
        repct=repct+1;
        count = count+1;
        /*
        put rlength;
        */

        inum=1;
        jnum=JSTRS;
        /* p0=0.2; */
        delta = 0; /*  deltaj; */
        array alpha(20);
do ainit = 1 to 20;
        alpha{ainit}=0.0;
end;

        alpha{1}= deltaj;  /* 0 */
        alpha{2}=0.0;
        alpha{3}=0.0;

        array signal(30);
do ksig = 1 to 30;
        signal(ksig) = 0;
end;
        array p(1,20);
        array y(1,20);
        array pjhat(20);
        array yjsum(20);
do i=1 to inum;
        do j=1 to jnum;
        p(i,j) = p0 + delta + alpha(j);

end;end;
       /* n = 50; */

  /* generate random samples */
do i=1 to inum;
        do j=1 to jnum;
                pij=p(i,j);
               y(i,j) = ranbin(seed1,n, p(i,j));
end;end;
 /*  calculate phat,pjhat, chisq overall, chisq-
for-j-columns
        test for signals  */
      /*  calc run length */
        array ystr(20);
        maxj = 1;
        tiesw = 0;
        ystr(1) =y(1,1);
        do jstep = 2 to jnum;
                ystr(jstep) = y(1,jstep);
                if ystr(maxj) ge ystr(jstep)
then
                        if ystr(maxj) eq
ystr(jstep) then
                                tiesw =1;
                        else maxj = maxj;
                else do; maxj = jstep ; tiesw =
0; end;

        end;
        /*
put 'before'  ystr1 ystr2 ystr3 ystr4 ystr5 maxj
tiesw prevmaxj rlength count;
        */
if tiesw = 0 then
if maxj=prevmaxj then rlength = rlength +1;
else do;
        prevmaxj=maxj;
        rlength=1;
end;
        /*
put 'after '  ystr1 ystr2 ystr3 ystr4 ystr5 maxj
tiesw prevmaxj rlength count;

        */
if rlength = r then rsig =1;

        /* end calc run length  */
ysum = 0;
zsqsum = 0;
zsqsumq = 0;
do j = 1 to jnum;
        yjsum(j) = 0;

do i = 1 to inum;
```

```
        ysum = ysum+y(i,j);
        yjsum(j) = yjsum(j) + y(i,j);
        zsqsum = zsqsum +(y(i,j)-n*p0)**2;
end;
pjhat(j)=yjsum(j)/(n*inum);
if pjhat(j) gt pjucl or pjhat(j) lt pjlcl then
signal(2) = 1;
if pjhat(j) gt pjucl2 or pjhat(j) lt pjlcl2 then
signal(4) = 1;
if pjhat(j) gt pjucl or pjhat(j) lt pjlcl then
signal(j+9) = 1;
        else signal(j+9) = 0;
zsqsumq = zsqsumq + (yjsum(j) - n*inum*p0)**2;
end;
phat=ysum/(n*inum*jnum);
csqst = (zsqsum)/(p0*(1-p0)*n);
csqstq = (zsqsumq)/(p0*(1-p0)*n*inum);
if csqst gt csqcl then signal(3) = 1;
        else signal(3) = 0;
 /* if csqstq gt csqclq then signal(4) = 1;
        else signal(4) = 0;  */
if phat gt ovpucl or phat lt ovplcl then
signal(1) = 1;
        else signal(1) = 0;
do ksig2 = 1 to 20;
        tsig(ksig2)=tsig(ksig2)+signal(ksig2);
end;
if signal(1)=1 then rsig=1;
end;
end;

totr=totr+count;
arlRp=totr/numreps;
arlp=repct/tsig1;
arlpj=repct/tsig2;
arlpj2=repct/tsig4;
arlchi=repct/tsig3;
output;
end;
END;
end;
run;
  /*
proc print;
        by jstrs n deltaj;
   var repct arlRp arlp  arlpj arlchi tsig1
tsig2 tsig3 tsig4 tsig5 tsig6 tsig7 tsig7 tsig8
tsig9
        tsig10 tsig11 tsig12 tsig13 tsig14
tsig15
        r csqcl csqst csqclq csqstq
        p1 p2 p3 p4 p5 p6 p7 p8 p9 p10 p11 p12
p13 p14 p15 p16
        alpha1 alpha2 alpha3 alpha4;
        */
title 'CASE 1 TO CASE 2'; /* if delta > 0 then
title 'case 1 to case 2';*/
proc tabulate data=simdat1;
        BY JNUM;
        class p0 deltaj n;
        var arlp arlchi arlpj arlpj2 arlRp r k1
k2;
        table p0,deltaj,n*(arlp*f=6.1
arlchi*f=6.1 arlpj*f=6.1 arlpj2*f=6.1
arlRp*f=6.1 r*f=1.0 k1*f=4.2 k2*f=4.2)*(MEAN);
        KEYLABEL
                MEAN='ARL';
run;
```

Program output and interpretation

Returning to the TOY process (see Table 3, where ARLP is the average run length for a 3-sigma p-chart, ARLCHI is the ARL for a Chi-Squared chart, ARLPJ is the ARL for a group p-chart with K1 = 4.07 sigma limits, ARLPJ2 is the ARL for a group p-chart with K2=4.06 sigma limits) one sees that for a scheme in which a *k=3* sigma overall *p*-chart is used in conjunction with a runs rule with R = 5 the in-control ARL is about 149. This drop should be attributed to the fact that the in-control ARL for the p-chart is about 212. A user might consider increasing *k* with the overall p-chart.

Table 3: Simulation Output

P0 0.023

	N							
	75							
	ARLP	ARLCHI	ARLPJ	ARLPJ2	ARLRP	R ARL	K1	K2
	ARL	ARL	ARL	ARL	ARL		ARL	ARL
DELTAJ								
0	211.6	149.1	707.9	140.8	149.4	5	4.07	4.06
0.01	93.4	64.5	228.3	63.1	58.9	5	4.07	4.06
0.02	43.9	24.3	64.8	22.4	20.7	5	4.07	4.06
0.05	8.5	3.3	5.2	3.0	5.2	5	4.07	4.06
0.1	2.1	1.2	1.4	1.2	2.0	5	4.07	4.06
0.2	1.0	1.0	1.0	1.0	1.0	5	4.07	4.06

IMPLEMENTATION

Below is a SAS program that produces a 3-sigma *p*-chart for the TOY process. The streams are denoted by S1, …, S4, which contain the number of defective items in each sample. There is data for eight epochs and the proportion defective is known to be 0.023. Since each of the four streams has sample size 75, the sample size for the overall p-chart is 300.

```
goptions;
data toydat;
        input s1 s2 s3 s4  sample;
        defect = s1+s2+s3+s4;
        cards;
1 2 1 2    1
0 2 3 0    2
2 0 2 4    3
2 3 0 5    4
4 2 1 7    5
3 4 2 4    6
1 1 3 5    7
2 3 1 4    8
;
proc shewhart data=toydat graphics;
```

```
        pchart defect*sample / subgroupn = 300
p0 = 0.023;
run;
```

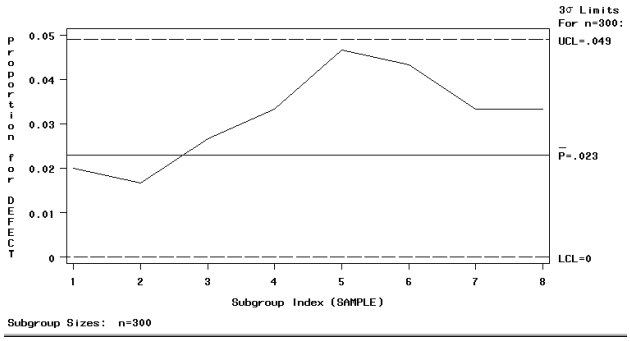


Figure 1: P-chart for TOY process

Using runs Rule 2 with $R = 5$, observe that at epoch 1 (sample 1) there is a tie for the max stream so there is no run-stream; at epoch 2 stream 3 has the largest proportion defective and the run length is one (that is, $r = 1$); at epoch 3 stream four has the largest proportion defective (there is a new max stream), and hence r = 1. Also, at epoch 4 stream 4 is the max and hence r = 2; at epoch 5 stream four is the max and hence $r = 3$; at epoch 6 there is a tie so r remains 3; at epoch 7 stream 4 is the max so r = 4 and at epoch 8 stream 4 is the max so r = 5. Since the run length is equal to R = 5, there is an out-of-control signal at epoch 8.

CONCLUSION

Several process control schemes useful for controlling multistream binomial processes have been discussed. A SAS® program for calculating ARLs for runs rules is provided. This program is a useful tool when designing a runs rule scheme. Next a SAS® monte carlo simulation program is provided that produces estimated ARLs that are useful for complex schemes, such as those using a runs rule in conjunction with an overall *p*-chart. Simulation study has shown this to be an effective process control scheme for multistream binomial processes..

DOWNLOADING SAS® PROGRAMS

Source code can be downloaded from the University of North Florida Center for Research and Consulting in Statistics web page (www.unf.edu/coas/math-stat/CRCS) as technical reports #080101 and #080201.

REFERENCES

Jacobs, S., and Wludyka, P., "Runs Rules and P-Charts for Multistream Binomial Processes," to appear (and CRCS technical report #080201).

Montgomery, D. C. (1997). *Introduction to Statistical Quality Control*, 3rd ed., John Wiley and Sons, New York.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors at:

Peter Wludyka
Associate Professor of Statistics
University of North Florida
Jacksonville, Florida
Work Phone: 904-620-1048
Fax: 904-620-2818
Email: pwludyka@unf.edu
Web: www.unf.edu/coas/math-stat/~pwludyka

Sheri Jacobs
Quality Engineer
Vistakon, Inc.
Jacksonville, Florida
Work Phone: 904-443-1607
Fax: 904-443-1607
Email: sjacobs@visius.jnj.com

2001: A SAS/STAT Odyssey

Maura Stokes , SAS

Abstract: Version 8 of SAS/STAT software introduced new tools in the areas of nonparametric regression, survey data analysis, and nonlinear mixed models. This talk reviews the new methodolgy included in Release 8.2, including the generalized additive model, exact logistic regression, and multiple imputation, and describes important additions to existing procedures such as ratio estimation in PROC SURVEYREG, generalized logistic regression in PROC LOGISTIC, and numerous enhancements in other procedures such as NPAR1WAY, FREQ, and PHREG. In addition, this talk discusses the new statistical enhancements to Version 9, which include additional work in survey data analysis, such as logistic regression for survey data, software for power and sample size computations, and numerous enhancements to existing procedures.

Paper P-715

A Simulation Study to Compare the Performance of Permutation Tests for Time by Group Interaction in an Unbalanced Repeated-Measures Design, Using Two Permutation Schemes

Mark S. Litaker and Bob Gutin, Medical College of Georgia, Augusta GA

ABSTRACT

The use of permutation tests for the analysis of data incorporating repeated measurements presents a choice of possible permutation schemes to generate the empirical null distribution of the test statistic. A commonly-recommended scheme keeps the vector of observations for each subject intact, with permutation based on group membership only. Another option is to permute the observations across the time points as well as relative to treatment group. These two permutation schemes are compared with respect to power and preservation of the nominal alpha level in testing for a time by group interaction using simulated data based on a randomized trial of effects of an exercise program. SAS® PROC IML is used to generate correlated observations under the null and alternative hypotheses which are then analyzed using PROC GLM. P-values are obtained for the usual F statistic and for each of the permutation tests. Proportions of significant interaction terms are calculated to obtain empirical power and alpha for each of the permutation schemes. The two schemes are then applied to the analysis of observed data from the study.

INTRODUCTION

In biological research it is often the case that observed data do not follow convenient distributions. Since many of the most-used methods of statistical analysis rely on the assumption that the data are sampled from a Normal distribution, use of these methods in many practical situations may give questionable results. While transformations may provide a remedy for some distributional problems, it is often the case that no suitable transformation exists. An alternative approach to analysis of such data is the use of permutation testing. Permutation tests are exact for all sample sizes regardless of the underlying distribution (Higgins, Noble; 1993). Assumptions regarding the form of the correlation structure of the parent distribution are not required. Two initial steps in the use of a permutation test are to identify a test statistic that differentiates between the null and alternative hypothesis situations, and to identify a permutation scheme that will generate the distribution of the test statistic under the null hypothesis (Good; 1995). Often there is a clear choice of permutation scheme based on the experimental design, but this is not always the case. In designs that incorporate repeated measurements on the same experimental units, one must decide whether to keep each subject's vector of observations intact or to permute across time points as well.

PERMUTATION TESTS

To perform a permutation test, the test statistic is calculated for the observed data. Then the data is permuted across groups, and the test statistic is calculated for each permutation. Typically a very large number of permutations is possible, so a random sample of permutations is selected instead of all possible arrangements. The test consists of calculating the area under the curve of this empirical null distribution of the test statistic that is at least as extreme as the test statistic that was calculated from the observed data. That is, the p-value is the fraction of the test statistics from the null distribution that are at least as extreme as that which resulted from the actual data. Typically, a random sample of 100 to 500 repetitions is adequate to approximate the permutation distribution (O'Sullivan, et al; 1984).

THE REPEATED MEASURES SITUATION

Consider an experimental design with two groups and repeated measurements made at three time points. The effect that is of interest is the difference between groups in change across the times of observation. An appropriate test statistic is one whose magnitude reflects the amount of deviation between the observed data and the null situation. For the current experimental design an obvious choice is the usual F statistic for group by time interaction from a split-plot or repeated measure analysis of variance (ANOVA). To generate the permutation distribution of the F statistic, observations are permuted across group assignments. The order of observations for each subject may be maintained, or the observed values for each subject may be permuted across observation times as well. In the repeated measures design, subjects are randomized to one or the other of the experimental groups, but the repeated observations occur at fixed times. That is, subjects are not randomized to times. It has been suggested (Higgins) that the permutation scheme should correspond to the randomization scheme used in the experiment. However, this is not consistent with the situation with the usual Normal-based F test. The null hypothesis for the overall F test is that both of the main effects and the interaction effect are zero, that is, that neither time nor group is a meaningful categorization, whether experimental units are randomized to group only or to group and subplot. Thus, it is of interest to compare the performance of the two permutation schemes.

EXAMPLE: A STUDY OF THE EFFECTS OF PHYSICAL TRAINING IN CHILDREN

Study subjects were 80 obese children 7 – 11 years of age , who were randomized to participate in a physical training program during either the first or second four-month period of the study. The primary interest was in estimating changes in body composition due to the physical training program. For this evaluation, percent body fat as measured by dual energy x-ray absorptiometry was selected as the dependent variable. The statistical analysis was by mixed-model ANOVA, which was performed using SAS PROC GLM. Subject was included as a random effect, and group and time were fixed effects in the model. The statistical test which is of interest is the F test for the group by time interaction term.

THE SIMULATED DATA SETS

In order to compare the performance of the two permutation schemes, simulated data was generated having approximately the structure that was observed in the actual study. The simulated data sets each consist of 80 subjects and 3 observations. Correlated observations were produced by generating for each subject a "subject effect" consisting of a Uniform(0,20) random variable. Observations at each of the time points were generated by adding a Normal random number to the subject effect with means set to generate the desired group by time effects. The standard deviation of the Normal term was set at 3.5, giving a standard deviation of approximately 6.75 for the sum of the Uniform and Normal components. This is similar to the within-cell standard deviation of the observed data. Since the subject effect is Uniform rather than Normal, the resulting

distributions will be symmetric, but will be overdispersed relative to the Normal distribution.

EVALUATING THE PERMUTATION TESTS

Each of the simulated data sets was analyzed using both of the permutation schemes. In scheme 1, each subject's vector of observations remains intact, and these 3-element vectors are permuted across the group assignments. In scheme 2, the vectors of observations are permuted across groups, and then the three observations for each subject are randomly permuted across times. In order to calculate p-values to four decimal places, 1000 random permutations were performed for each test. For each of the permutation tests and the Normal-based ANOVA on each simulated data set, p-values for group by time interaction were written to text files for performance comparisons. Validity of the tests was evaluated by simulating data with no group by time interaction. In the simulations of the null situation, mean percent body fat was increasing over the times of observation in both groups. If the test rejects the null hypothesis in 5% of trials when alpha = .05, when the null hypothesis is true, then it is a valid test. To evaluate the power of the tests, data was simulated with a group by time interaction. For group 1, the group which received the training program during months 0 – 4, mean percent fat values were 44 , 42, and 43 for months 0, 4, and 8, respectively. For group 2, the corresponding means were 44, 45, and 43. The power of the test is the proportion of trials that result in rejection of the null hypothesis. At the 95% confidence level, this is the proportion of tests which yield a p-value of .05 or less.

RESULTS

Preliminary results of the simulation study show little evidence of a difference in performance between the two permutation tests. In 253 simulations under the null hypothesis, scheme 1 demonstrated an empirical alpha level of .0553, while scheme 2 showed .0593. The observed alpha for the F test was also .0593. Under the alternative situation, observed power in 285 simulations was 76.5% for the test using permutation scheme 1, 77.2% using scheme 2, and 76.8% for the F test.

Analysis of the observed data from the physical training study using the Normal-based ANOVA and each of the permutation tests gave similar results. The p-values for group by time interaction were .0007 for the ANOVA, and .002 for the permutation tests. In this case, the two permutation schemes gave identical results.

CONCLUSION

Based on these results, there is not a clear performance basis on which to choose between the permutation schemes. Permutation scheme 1 is a simpler procedure than scheme 2, and requires less computing time. This may suggest that scheme 1 would be preferable in practice. Performance of both permutation tests was similar to that of the Normal-theory test for the observed and simulated data in this study. The assumption of Normality is not seriously violated for these data, despite the use of a mixed distribution in the simulations. Thus, this study may not provide an adequate tool to distinguish differences in performance between the tests that might exist for data with a more extreme departure from Normality.

REFERENCES

1. SAS/IML® Software: Usage and Reference, Version 6, First Edition. SAS Institute, Inc., Cary, NC; 1980.

2. Higgins JJ, Noble W. A Permutation Test for a Repeated Measures Design, Proceediings of the 1993 Kansas State University Conference on Applied Statistics in Agriculture; 1993

3. Good P. Permutation Tests: A Practical Guide to Resampling Methods for Testing Hypotheses. Springer, New York; 1995.

4. O'Sullivan F, Whitney P, Hinshelwood MM, Hauser ER. The Analysis of Repeated Measures Experiments in Endocrinology. J Animal Science 59 (4):1070-1079; 1984.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Author Name: Mark Litaker
Company: Medical College of Georgia
Address: Office of Biostatistics and Bioinformatics
City state ZIP: Augusta, GA 30912-4900
Work Phone: (706) 721-3785
Fax: (706) 721-6294
Email: mlitaker@mail.mcg.edu

APPENDIX.

```
options cleanup nosource nonotes;

*-------------------------------------------------------------;
* provide a name for the  dependent variable       ;
*-------------------------------------------------------------;
%let dep = fat;

*-----------------------------------------------------------------------;
* identify 3 measurements of the dependent variable     ;
*-----------------------------------------------------------------------;
%let dep1 = fat1;
%let dep2 = fat2;
%let dep3 = fat3;
*-----------------------------------------------------------------------------;
* macro to repeatedly simulate data and analyze by         ;
* permutation tests using different permutation schemes  ;
*-----------------------------------------------------------------------------;

* permutation scheme 1 *;

%macro permute1(reps1);
  %do i = 1 %to &reps1;
*-----------------------------------------------------------------------------;
* reassign 3 observations per subject randomly to groups  ;
* keeping each subjects vector of observations intact         ;
*-----------------------------------------------------------------------------;

proc iml;
use &dep;
read all var {subject &dep1 &dep2 &dep3} into &dep;
rand1=j(nrow(&dep),1,1);
do i = 1 to nrow(rand1);
  rand1(|i|)=ranuni(-1);
end;
order=&dep||rand1;
create order2 var {subject &dep1 &dep2 &dep3 rand1};
append from order;
run;

proc sort data=order2;
by rand1;
run;
proc iml;
use order2;
read all var {subject &dep1 &dep2 &dep3} into reorder;
use fat;
read all var {group} into group;
permute1=group||reorder;
create permute var {group subject &dep1 &dep2 &dep3};
append from permute1;
run;

 *---------------------------------------------;
 *  analysis of variance for the       ;
 *  permuted data                        ;
 *---------------------------------------------;
 *  rearrange data to have a single observation per record;
data permute2;
set permute;
do i = 0 to 8 by 4;
 subject = subject;
 group = group;
 month = i;
 if month eq 0 then &dep = &dep1;
 else if month eq 4 then &dep = &dep2;
 else if month eq 8 then &dep = &dep3;
 output;
end;
keep subject group month &dep;
run;
proc glm data=permute2 noprint outstat=stats;
class subject group month;
model &dep  = subject(group) group month group*month /ss3;
random subject(group);
run;
data stats;
set stats;
if  _source_ = 'GROUP*month';
file "&dep 1.txt" mod;
put f;
run;
   %end;
%mend permute1;


* permutation scheme 2;
* randomization by group and time;

%macro permute2(reps2);
  %do i = 1 %to &reps2;
*-----------------------------------------------------------------------;
* reassign 3 observations per subject randomly to groups  ;
* and randomly to times                                                 ;
*-----------------------------------------------------------------------;
proc iml;
use &dep;
read all var {&dep1 &dep2 &dep3} into &dep;
rand1=j(ncol(&dep),1,1);
do i = 1 to nrow(rand1);
  rand1(|i|)=ranuni(-1);
end;
rand2=rand1';
randcols=rand2//&dep;

create order2 var {col1 col2 col3};
append from randcols;
run;

proc transpose data=order2 out=order3;
run;
* permute columns;
proc sort data=order3;
by col1;
run;
data order3;
set order3;
drop _name_ col1;
run;
proc transpose data=order3 out=order2;
run;
data order2;
set order2;
drop _name_;
&dep1=col1;
&dep2=col2;
&dep3=col3;
drop col1 col2 col3;
run;
proc iml;
use &dep;
read all var {group} into group;
rand2=j(nrow(group),1,1);
do i = 1 to nrow(rand2);
  rand2(|i|)=ranuni(-1);
end;
group=group||rand2;
create group var {group rand};
append from group;                    run;
* permute rows;
proc sort data=group;
by rand;
run;

proc iml;
use group;
```

```
read all var {group};
use order2;
read all var {&dep1 &dep2 &dep3} into &dep;
use &dep;
read all var {subject} into subject;
permute=subject||group||&dep;
create permute var {subject group &dep1 &dep2 &dep3};
append from permute;
run;

 *---------------------------------- ---------------------------------------;
 *  analysis of variance for the  randomized data                ;
 *-------------------------------------------------------------------------;

 *  rearrange data to have a single observation per record;
data permute2;
set permute;
do i = 0 to 8 by 4;
 subject = subject;
 group = group;
 month = i;
 if month eq 0 then &dep = &dep1;
 else if month eq 4 then &dep = &dep2;
 else if month eq 8 then &dep = &dep3;
 output;
end;
keep subject group month &dep;
run;
proc glm data=permute2 noprint outstat=stats;
class subject group month;
model &dep  = subject(group) group month group*month /ss3;
random subject(group);
run;
data stats;
set stats;
if  _source_ = 'GROUP*month';
file "&dep 2.txt" mod;
put f;
run;
   %end;
%mend permute2;

*      ------------------------------------------------------------------------;
*      macro to simulate data and perform 3 tests                ;
*      ------------------------------------------------------------------------;
%macro simstudy(reps);
  %do j = 1 %to &reps;

*-------------------------------------------------------------------------;
* SIMULATE data with one record per subject,              ;
* three observations per subject in each record             ;
*-------------------------------------------------------------------------;

proc iml;
subject=j(80,1,1);
group1=j(40,1,1);
group2=j(40,1,2);
subeffect=j(80,1,1);
fixed1=j(80,1,1);
fixed2=j(80,1,1);
fixed3=j(80,1,1);
fat1=j(80,1,1);
fat2=j(80,1,1);
fat3=j(80,1,1);
group=group1//group2;

   *  generate  the alternative hypothesis situation  ;
   * simulate data with group*time effect;

do i = 1 to 40;
  subject(|i|) = i;
  subeffect(|i|)= 20*ranuni(-1);
  fixed1(|i|) = 34 + 3.5*rannor(-1);
  fixed2(|i|) =  32 + 3.5*rannor(-1);
  fixed3(|i|) = 33 + 3.5*rannor(-1);
end;
do i = 41 to 80;
  subject(|i|) = i;
  subeffect(|i|)= 20*ranuni(-1);
  fixed1(|i|) = 34 + 3.5*rannor(-1);
  fixed2(|i|) =  35 + 3.5*rannor(-1);
  fixed3(|i|) = 33 + 3.5*rannor(-1);
end;

  /*   *OR generate the null situation;
       * simulate data with time effect only;
      do i = 1 to 80;
  subject(|i|) = i;
  subeffect(|i|)= 20*ranuni(-1);
  fixed1(|i|) = 34 + 3.5*rannor(-1);
  fixed2(|i|) =  35 + 3.5*rannor(-1);
  fixed3(|i|) = 36 + 3.5*rannor(-1);
end;
  */
fat1=subeffect+fixed1;
fat2=subeffect+fixed2;
fat3=subeffect+fixed3;
fat=subject||group||fat1||fat2||fat3;
create fat var {subject group fat1 fat2 fat3};
append from fat;
run;


*-------------------------------------------------;
* analysis of observed data  (anova) ;
*-------------------------------------------------;

*  rearrange data to have a single observation per record;

data &dep2;
set &dep;
do i = 0 to 8 by 4;
 subject = subject;
 group = group;
 month = i;
 if month eq 0 then &dep = &dep1;
 else if month eq 4 then &dep = &dep2;
 else if month eq 8 then &dep = &dep3;
 output;
end;
keep subject group month &dep;
run;

* analysis of variance;

proc glm data=&dep2 noprint outstat=fstat;
class subject group month;
model &dep  = subject(group) group month group*month /ss3;
random subject(group);
run;
```

```
*  save the observed value of the F-statistic  ;

data fstat;
set fstat;
if  _source_ = 'GROUP*month';
file 'alt.txt' mod;
put _name_ _source_ f prob;
call symput('obs_f',f);    *value of F from the observed data;
run;

*  run permutation test using scheme 1;
%permute1(1000)
run;

data fstats;
infile "&dep 1.txt" missover;
input f;
if f eq . then delete;
p = 0;
if f ge &obs_f then p = 1;
label p='(p-value)';
run;
proc means data=fstats n mean noprint;
title1 'permutation 1:  permute by group only';
title2 'group*time p-value';
title3 "dependent variable: &dep";
output out=pval1 n=n1 mean=pval1;
var p;
run;
data pval1;
set pval1;
file 'alt.txt' mod;  * if simulating group*time effect;
* file 'null.txt' mod;  * if simulating null ;
put n1 pval1;
run;

* run permutation test using scheme 2;
%permute2(1000)
run;

data fstats;
infile "&dep 2.txt" missover;
input f;
if f=. then delete;
p = 0;
if f ge &obs_f then p = 1;
label p='(p-value)';
run;
proc means data=fstats n mean noprint;
title1 'permutation 2: permuting by group and time';
title2 'group*time p-value';
title3 "dependent variable: &dep";
output out=pval2 n=n2 mean=pval2;
var p;
data pval2;
set pval2;
file 'alt.txt' mod;
put n2 pval2;
run;

  * remove temporary files;
  data clear;
  space=.;
file "&dep 1.txt";
put space;
file "fat 2.txt";
put space;
                run;

%end;
proc datasets kill;
%mend simstudy;
run;
```

```
%let reps=200;   * set number of simulations;
%simstudy(&reps)
run;

* calculate and print empirical power and alpha values;

data output;
infile 'alt.txt';  * for alternative H;
* infile 'null.txt'; * for null H;
input _name_ $ _source_ $ obsf obsp / n1 pval1 / n2 pval2;
if obsp le .05 then obs_power=1;
if obsp gt .05 then obs_power=0;
if pval1 le .05 then p1_power=1;
if pval1 gt .05 then p1_power=0;
if pval2 le .05 then p2_power=1;
if pval2 gt .05 then p2_power=0;
run;

proc means data=output n mean maxdec=4;
title1 'power for F and two permutation tests';
title2 'alternative hypothesis';
*title2 'null hypothesis';
var obs_power p1_power p2_power;
run;
```

Heel Ultrasound as a Predictor of Appendicular Bone Mineral Density

Rebecca G. Frederick, Louisiana State University, Baton Rouge, LA
E. Barry Moser, Louisiana State University, Baton Rouge, LA
Ellen R. Brooks, Northwestern University Medical School, Chicago, IL
And Woman's Health Research Institute, Woman's Hospital, Baton Rouge, LA.

Abstract

Emerging medical technologies for use in the area of skeletal assessment of bone mineral density (BMD) are on the rise, while at the same time the aging population and the incidence of osteoporosis both increase. A new technology, quantitative peripheral heel ultrasound (Sahara Clinical Bone Sonometer, Hologic), measures variables related to sound transmission through the calcaneus. This study evaluates the utility of heel ultrasound in predicting appendicular BMD as determined by dual energy x-ray absorptiometry (Hologic QDR 4500) at the hip in Caucasian females, 65-81 years old. Computations were performed on an IBM 300PL personal computer with Microsoft® Windows NT® Workstation and SAS® software. Proc Reg was used for variable selection and model building, while Proc CanCorr provided canonical correlations of the data. There is an adequate correlation that exists for predicting the T-score (a diagnostic criterion of the World Health Organization) for the total femur (hip) and the femoral neck (hip) of women using the heel ultrasound T-score, broadband attenuation of sound waves (BUA), speed of sound (SOS), stiffness index (QUI), age and body mass index (BMI). This paper discusses the selection of predictor variables and the evaluation of the model.

Introduction

The measurement of bone mineral density (BMD = gm/cm^2) and bone mineral content (BMC = grams) using dual energy X-ray absorptiometry (DXA), is a widely accepted technology that affords low radiation exposure, quick scanning time, and quantitative data from which to rule out bone loss (Genant, et al, 1996). Two anatomical sites, the lumbar spine and proximal femur (hip) are usually measured using DXA.

Although the DXA has been accepted as the gold standard for evaluating area bone density, other methods have more recently been developed that may provide additional information on bone microarchitecture. This information is particularly significant with respect to ascertaining fracture risk. Quantitative peripheral heel ultrasound of the calcaneus (heel) uses imperceptible sound waves that are passed through the heel. This equipment is portable, whereas DXA is not, and may therefore make heel ultrasound a useful tool for mass (community-based) screening of the high-risk postmenopausal population. Further, it may be a more cost-effective means for identifying those individuals who should be referred for detailed osteoporosis evaluation (Sim, Stone, Johansen, and Evans, 2000; Stewart, 2000). Since the diagnostic criterion for osteoporosis is based upon DXA, further evaluation of the relationship between the variables measured with the two differing technologies is warranted.

Purpose

The purpose of this investigation was to assess the strength of the predictive relationship of heel ultrasound for variables from hip DXA scanning, particularly the T-score. Our specific focus was on the older Caucasian postmenopausal female (aged 65-81 years), since older women are at high risk for continued slow bone loss (Riggs, Khosla, and Melton, 1998).

Methods

The Institutional Review Board approved this study and the informed consent of all subjects was obtained. One hundred and sixty-seven women were enrolled in this study. Anthropometric data that included height, weight, and body mass index (BMI = wt in kg/ht in m^2) were measured. Age was also included as an explanatory variable, since it is a potentially confounding variable due to the decline that occurs in bone density with aging.

DXA scanning of the hip and heel ultrasound measurements were performed on the same day. Since the left hip was scanned by DXA, all duplicate heel ultrasound measurements were made using the left heel. The left lower leg (and thus heel) was positioned, using a goniometer, such that there was a 90 degree angle at the knee.

Variables that were measured using the differing technologies are:

Table 1: Quantitative Heel Ultrasound

- Broadband attenuation of sound waves (BUA)
- Speed of Sound (SOS)
- Stiffness index of bone (QUI)
- T-score (World Health Organization diagnostic criterion for determination of a patients' current BMD versus their expected peak BMD. This comparison is delineated into three categories: normal, low BMD, or osteoporosis. The T-score criterion is based upon DXA data)

Table 2: DXA of the Proximal Femur-Hip

- Total Femur BMD, BMC, and T-score (defined above)
- Femoral Neck BMD, BMC, and T-score (defined above)

Analysis

Regression Methods

The data was imported into SAS from an Excel spreadsheet. Proc Univariate was used to review the data and identify any potential errors and outliers in data entry. Proc Reg, with the stepwise option, was then used to determine which DXA variable (of the hip) was most related to the heel ultrasound

variables. These models used age and BMI. An example of a typical model was:

Exhibit 1:Typical Model of Data

Model $Y=\beta_0+\alpha_1 X_1+...+\alpha_k X_k+\beta_2 X_2+\beta_3 X_3+\varepsilon$

Where:

Y	: Hip Measurements
β_0	: Intercept
α_1 ,..., α_k	: Coefficients of Heel Measurements
X_1 ,...,X_k	: Heel Measurements
β_2	: Coefficients of Age
X_2	: Age of individual
β_3	: Coefficients of Body Mass Index
X_3	: Body Mass Index (BMI)
ε	: Error

Stepwise regression, at the default significance level of 0.15, was performed to identify potential models of explanatory variables. A typical model statement was:

Exhibit 2: Hip Measurements of SAS Regression Procedure

```
Proc Reg;
Model LT_TOTF2 = US_T_S US_QUI US_BUA US_SOS BMI AGE
                 / selection=stepwise;
Run;
```

Polynomial Interactions were also assessed, however, none were found to be statistically important for any of the explanatory variables using either heel measurement.

The results from the stepwise regressions did not identify the same model with respect to the heel measurements. The first explanatory measure, at alpha = 0.15, selected the total femur T-score, then the BMI, and finally, age. For the femoral neck T-score, using BMI and age as covariates, the procedure selected heel BUA. The total model R^2 for total femur T-score was 0.4959, and for the femoral neck T-score was 0.3670.

Table 3: Hip Procedure for Regression with the Stepwise Option

Variable Entered	*Partial* R^2	*Model* R^2	*F Value*	*Prob > F*
Total Femur T-Score				
T-score	0.3012	0.3012	69.39	<.0001
BMI	0.1832	0.4844	56.86	<.0001
Age	0.0115	0.4959	3.26	0.0590
Femoral Neck T-Score				
BUA	0.2514	0.2514	54.06	<.0001
BMI	0.0906	0.3420	22.04	<.0001
Age	0.0250	0.3670	6.27	0.0133

Regression Diagnostics

Partial regression plots are used to assess the major role that an explanatory variable, X_k, plays in the model given that all of other explanatory variables under consideration are already in the model. Both the response variable, Y, and the explanatory variable, X_k, are regressed against the other explanatory variables in the model and the residuals are obtained from each fit. Thus, a partial regression plot for the total femur T-score, Y, and the heel T-score, X_1, consists of a plot of the Y residuals, $\varepsilon(Y \mid X_2\ X_3)$ against the X_1 residuals, $\varepsilon(X_1 \mid X_2\ X_3)$ where X_2 and X_3 are BMI and Age, respectively. For the three-variable model there are three partial leverage plots of the residuals of the total femur T-score versus residuals from each of the predictors. These graphics appear in figure 1:

Figure 1: Partial Regression Plot 1 of T-Score

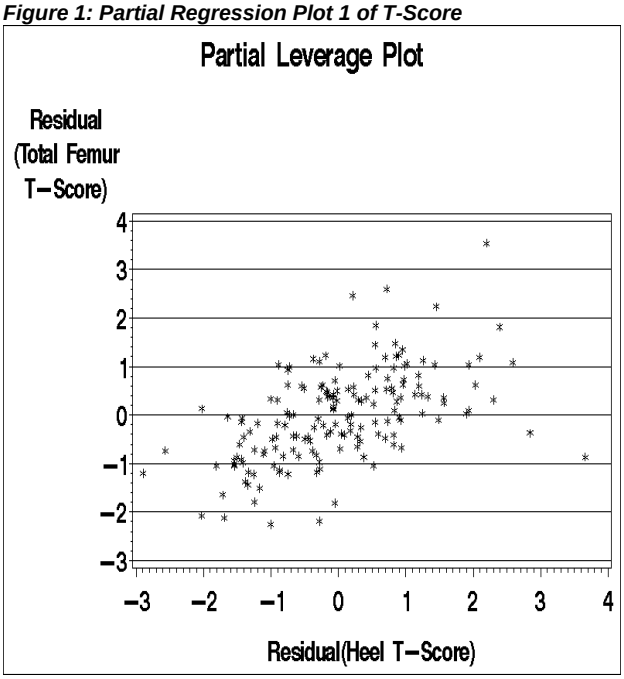


Figure 2: Partial Regression Plot 2 of Body Mass Index

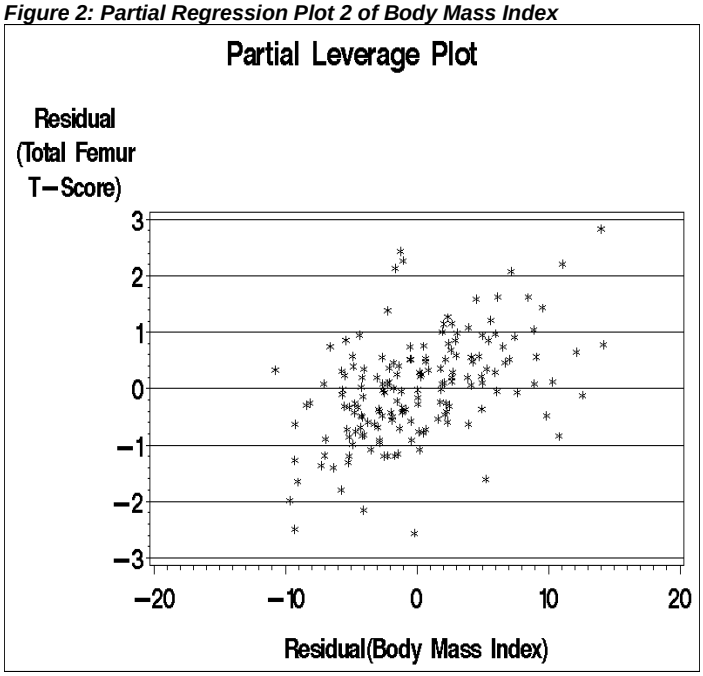
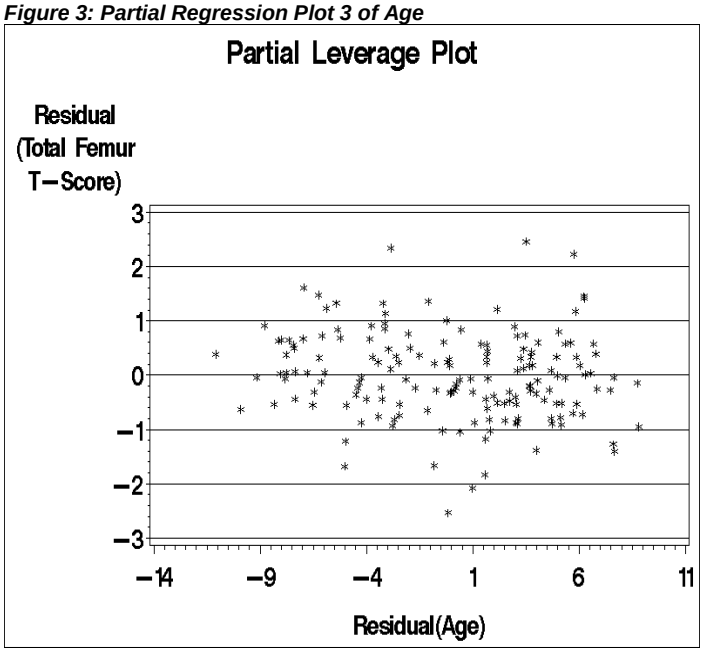


Figure 3: Partial Regression Plot 3 of Age



From the three partial regression plots, Figures 1-3, the T-score of the hip and the BMI support relationships with femur T-score. This occurred even when the other explanatory variables are in the model. Yet, age contributed little to no additional information for predicting femur T-score when the heel ultrasound T-score and BMI were in the model.

Canonical Correlation

Canonical correlation (CANCORR procedure) was used to perform canonical correlation, partial canonical correlation, and canonical redundancy analysis. It was used to analyze the relationship between two sets of variables (linear sets of both the hip and heel measurements). Canonical correlation is a type of correlation that is a variation on the concept of multiple regression and correlation analysis. The procedure was written as:

Exhibit 3: SAS Procedure of Canonical Correlations

```
PROC CANCORR   ALL
   Vprefix=Heel Vname="Measurement from the Heel"
   Wprefix=Hip Wname="Measurement from the Hip";
VAR   US_T_S US_SOS US_BUA US_QUI;
WITH LT_TOTF2 LT_FEM_3;
RUN;
```

Proc CANCORR has an option **ALL,** (*Exhibit 3)* that displays the correlations among the original heel and hip measurements. The correlations within the heel measurement were in the range of 0.9978 to 0.9184. The correlation within the hip measurements was 0.8878. The procedure also displays the correlations between the set of X (heel) variables, and the set of Y (hip) variables. The correlations between the heel and hip measurements were smaller than the within set of correlations as expected.

The largest correlation was between the heel BUA and the hip T-score (r=0.5391). The smallest correlation was between the heel SOS and the T-score for the femoral neck (r=0.4443). The correlations for the within and between measurements are shown below:

Table 4: Correlations Within the Heel Measurements

	T-score	SOS	BUA	QUI
T-score	1.0000	0.9893	0.9659	0.9977
SOS	0.9893	1.0000	0.9184	0.9876
BUA	0.9659	0.9184	1.0000	0.9638
QUI	0.9977	0.9876	0.9638	1.000

Table 5: Correlations Within Hip Measurements

	Total Femur T-Score	Femoral Neck T-Score
Total Femur T-Score	1.000	0.8878
Femoral Neck T-Score	0.8878	1.000

Table 6: Correlations Between Heel and Hip Measurements

	Total Femur T-Score	Femoral Neck T-Score
Heel T-score	0.5195	0.4717
SOS	0.4894	0.4443
BUA	0.5391	0.4893
QUI	0.5154	0.4673

Table 7: Canonical Correlations & Multivariate Statistics

	Canonical Correlation	Eigenvalue	Probability > F	Wilks' Lamba
1	0.556299	0.4490	<.0001	<.0001
2	0.026812	0.0007	0.9898	

As seen in **Table 7**, the first canonical correlation was 0.556 ($p<0.0001$), which indicates a definite linear association between hip and heel measurements, however the magnitude of the correlation was not very high, indicating that predictions would not be precise. Since the measurements were in the same units, the standardized coefficients were almost equivalent to the raw coefficients (**Tables 8 & 9**).

Table 8: Standardized Canonical Coefficients for Heel Measurements

	Heel 1	Heel 2
Heel T-score	13.2139	-37.8835
SOS	-8.7556	18.1880
BUA	-3.8424	10.5013
QUI	0.0934	9.9315

Table 9: Standardized Canonical Coefficients for Hip Measurements

	Hip 1	Hip 2
Total Femur T-score	0.8973	1.9790
Femoral Neck T-score	0.1141	-2.1700

Since canonical variable 1 was significant, it was interpreted. The standardized canonical heel measurement variable 1 was a weighted difference between T-score with SOS and BUA:

- a very strong positive value of T-score = 13.2139
- a strong negative value of SOS = -8.7556
- a negative value of BUA = -3.8424
- a very small positive value of QUI = 0.0934.

The first canonical variable of standardized hip measurements was largely dominated by the femoral neck T-score since we have:

- a positive value of total femur T-score = 0.897
- a small positive value of femoral neck T-score = 0.114

The canonical redundancy analysis did not indicate a good relationship between the opposite canonical variables and their own canonical variables (the hip and heel measurement). The standardized proportion of variance that explained the heel and hip measurement was 0.2553 and 0.2844, respectively. Even after they were added together, the cumulative proportions were only 0.2554 and 0.2845, respectively.

Table 10: Canonical Redundancy Analysis

	Standardized Variance of Heel Measurement		Standardized Variance of Hip Measurement	
	Proportion	Cumulative Proportion	Proportion	Cumulative Proportion
1	0.2553	0.2553	0.2844	0.2844
2	0.0000	0.2554	0.0001	0.2845

Conclusion

The strongest correlation relationship observed between the heel and hip, using differing technologies, was between the total femur T-score and heel BUA. Bone is known to attenuate high frequency (ultrasonic) sound waves, with those patients who are older and are osteoporotic, such that a lower BUA reading is obtained. Consistent with this, we were able to demonstrate that a direct correlation existed between the hip T-score and heel BUA measurement. Simply stated, the lower the hip T-score, the lower the heel BUA measurement.

Although a statistical relationship existed between variables from the heel and hip, using diverse technologies, a statistically robust predictive relationship was not observed. Anatomic site discordance in BMD, using DXA, is a well-known phenomenon that has made DXA interpretation for diagnosis somewhat confusing to the clinician (Mulder, Michaeli, Flaster, Siris, 2000; Faulkner, von Stetten, Miller, 1999). This problem was therefore already an issue (we assessed heel and hip) and confounded further by the use of different technologies to quantify the BMD.

Although two ultrasound variables (BUA and SOS) explained some of the variance in the hip data, BMI and age were also selected in the stepwise regression model for the total femur T-score. Both weight (reflected in the BMI ratio) and age were known to have an affect on BMD. When we controlled for age and BMI, BUA was the strongest predictor for the femoral neck T-score.

Quantitative peripheral ultrasound does not strongly explain and account for the variance in the total femur and femoral neck T-score, as quantified by DXA. Further studies are required to better understand the biological, corresponding statistical relationship, the clinical utility and interpretation of quantitative peripheral ultrasound of the heel and how those results relate to the gold standard, DXA.

References

Borland, Russell. Running Microsoft® Word for Windows® 95, Microsoft Press, A Division of Microsoft Corporation, 1995.

Faulkner, K.G., von Stetten, E., Miller, P. Discordance in patient classification using T-scores. Journal of Clinical Densitometry. 2:343-350, 1999.

Genant, H., Engelke, K., Fuerst, T., Gluer, C., Grampp, S., Harris, S.T., Jergas, M., Lang, T., Lu, Y., Majumdar, S., Mathur, A., Takada, M. Non invasive assessment of bone mineral and structure: state of the art. Journal of Bone and Mineral Research. 11:707-730, 1996.

Microsoft® Windows® 95 Resource Kit, Microsoft Press, A Division of Microsoft Corporation, 1995

Mulder, J.E., Michaeli, D., Flaster E.R., Siris, E. Comparison of bone mineral density of the phalanges, lumbar spine, hip, and forearm for the assessment of osteoporosis in postmenopausal women. Journal of Clinical Densitometry. 3:373-381, 2000.

Riggs, B.L., Khosla, S., Melton, L.J. A unitary model for involutional osteoporosis: estrogen deficiency causes both type I and type II osteoporosis in postmenopausal women and contributes to bone loss in aging men. Journal of Bone and Mineral Research. 13:763-773, 1998.

SAS Institute, Inc. SAS/INSIGHT® User's Guide, Version 6, Second Edition. Cary, NC: SAS Institute, Inc., 1993.

SAS Institute, Inc. SAS/GRAPH® Software: Usage, Version 6 Edition, First Edition. Cary, NC: SAS Institute, Inc., 1991.

SAS Institute, Inc. SAS/STAT™ Guide for Personal Computers, Version 6 Edition. Cary, NC: SAS Institute, Inc., 1987.

SAS Institute, Inc. SAS® Procedures Guide for Personal Computers, Version 6 Edition. Cary, NC: SAS Institute, Inc., 1985.

SAS Institute, Inc. SAS® Online Doc, Version 8 Edition. Cary, NC: SAS Institute, Inc., 1999.

Sim, M.F., Stone, M., Johansen, A., Evans, W. Cost effectiveness analysis of BMD referral for DXA using ultrasound as a selective pre-screen in a group of women with low trauma Colles' fractures. Technology and Health Care Journal. 8:277-284, 2000.

Stewart, R.A. Quantitative ultrasound or clinical risk-factors-which best identifies women at risk of osteoporosis? British Journal of Radiology. 73:165-171, 2000.

Virgile, Robert. Efficiency—Improving the Performance of Your SAS® Applications, Cary, NC SAS Institute, Inc., 1998.

Acknowledgments

Special thanks to the Osteoporosis Center at Woman's Hospital, under the direction of Joel Silverberg, M.D. and Laura Rodger, R.T. for their time and support of this study. In addition, a special thanks goes to Rehana Javed, M.D. for her dedication in making all of the heel ultrasound measurements.

The authors would like to thank Edith Flaster for her helpful comments on an earlier report from this investigation.

Contact Information

Your comments and questions are valued and encouraged. Contact the authors at:

Rebecca Frederick or E. Barry Moser
Louisiana State University
Dept. of Experimental Statistics
161 Agric. Admin. Bldg.
Baton Rouge, LA 70803
Work Phone: 225-578-8303
Fax Phone: 225-578-8344
Email: rfrederi@lsu.edu
bmoser@lsu.edu

Ellen R. Brooks, Ph.D.
Northwestern University Medical School-Division of Cardiology and Department of Preventive Medicine,
Chicago, IL 60611
Work Phone: 312-908-0020
Fax Phone: 312-908-0031
Email: e-brooks3@northwestern.edu

Survey Estimates and Variance Estimation Using the SURVEYMEANS Procedure

Hossein N. Yarandi, Ph.D.*, College of Nursing and Biostatistics Unit, University of Florida

Shawn M. Kneipp, Ph.D., ARNP, College of Nursing, University of Florida

(* - principal contact)

National surveys, conducted by the various governmental organizations, are usually characterized with a complex sample design. The characteristics of a complex sample design may include clustering, stratification, disproportionate sampling, and multiple stages of sample selection. Analyzing data from a complex sample survey cannot be done accurately using the standard SAS Procedures such as MEANS, UNIVARIATE, REG, or GLM. When using these procedures, it is assumed that the data are selected from a simple random sample.

A simple random sample requires that one have access to all the subjects in the population and the probability of selecting each subject in the population is the same for each and every subject. In addition, the chance of selecting a subject is independent even if other subjects are chosen. In contrast, when the data are collected from a survey with a complex sample design, the variance of the variables measured in the survey will be underestimated when using the standard statistical methods for nonsurvey data resulting in lower confidence intervals and higher Type I error by rejecting a true null hypothesis (Cohen 1997). A number of alternatives, called replication methods, have been developed for estimating variances and weighting procedures for the types of complex sample designs usually encountered in practice (Judkins and Wright 1990; Flyer, Morganstein, and Rust 1989).

In SAS Version 8 and beyond, three procedures are added to SAS/STAT software (SAS/STAT, 2000). These procedures are: SURVEYMEANS, SURVEYREG, and SURVEYSELECT. They are designed to specifically analyze complex survey data. The purpose of this article is to demonstrate the difference in variance estimation using the SURVEYMEANS procedures for a variable, such as AGE, from the Medical Expenditure Panel Survey. The data will be analyzed without the sampling weights, with sampling weights, and stratified sampling.

Medical Expenditure Panel Survey

The Medical Expenditure Panel Survey (MEPS) is cosponsored by the Agency for Healthcare Research and Quality (AHRQ) and the National Center for Health Statistics (NCHS). MEPS provides nationally representative estimates of health care use, its expenditure, sources of payment, and insurance coverage for the U.S. civilian non-institutionalized population (MEPS HC-003, 1996). MEPS also includes a national representative survey of nursing homes and their residents. MEPS is comprised of four component surveys: the household component, the medical provider component, the insurance component, and the nursing home component. In addition, it allows for

health services research intended to guide health care policy.

Designed to capture the changing dynamics of health care delivery and its insurance system, MEPS began in 1996. An important feature of the MEPS is that it uses a panel survey design in which minority and low-income households are oversampled. Medical expenditure data are collected at both the individual and household levels through a prescreening telephone interview, a mailed questionnaire, and a telephone followup interview for nonrespondents. The data reflect demographic characteristics, income, employment, health conditions, health status, access and use of medical care services, charges and payments, satisfaction with care, and health insurance coverage (MEPS HC-003, 1996).

Replication Methods for Analyzing Data from Complex Survey

Replication methods involve the selection of subsamples from the whole sample, then the estimated variances among the subsamples are used to estimate the variance of the full sample (Judkins and Wright 1990; and Wolter 1985). Another approach is the Taylor Expansion Method that can be used in estimating variances for the types of complex sample designs and weighting procedures usually encountered in practice. There are various replication approaches for creating subsamples from the full sample but the two widely used methods are Balanced Repeated Replication and the Jackknife Replication. Balanced repeated replication is generally used with multistage stratified sample designs while the Jackknife replication is preferred when there is no explicit stratification. The jackknife replication method can, however, be used when systematic sampling has been employed (Cohen, Burt, and Jones 1986; and Flyer 1987). Frequently, in complex survey data sets the full sample weights are provided. In addition, to employ a replication method, two additional replicate weights called the primary sampling unit (PSU) and the variance strata are usually required.

MEPS uses multistage sampling: At the first stage of sampling, a small number of primary sampling units (PSUs) are selected. Most of these PSUs consist of individual counties, but sometimes they include two or more adjacent counties. A random samples of PSUs are selected in the first stage of sampling. During the second and additional stages of sampling, only individuals living within these selected PSUs are selected.

To decrease the variability of parameter estimators based on data from the completed survey, sampling theory suggests that it is better to sample large-population PSUs with a higher probability than small-population PSUs (Korn and Graubard, 1999). Recall that, one of the design considerations of MEPS is to provide reliable estimates the minority groups. To help accomplish this, PSUs with larger proportions of these minorities were included in the sample with higher probabilities. Additionally, from a meteorological perspective, one can classify PSUs into a small number of groups (“strata”) and then sample a small number of PSUs from each stratum. This is known as stratified sampling, and is frequently used to decrease the variability of estimated quantities. Sample survey methodology allows one to sample units from the strata with differing probabilities provided by the survey designers. The data from each individual are associated with a sample weight, which is essentially the number of people in the target population that he or she represents. In calculating these weights, consideration is given to the differential probabilities with which individuals are sampled.

SAS Procedures

There are three SAS procedures designed to manage survey data. They include SURVEYMEANS Procedure, SURVEYREG Procedure, and SURVEYSELECT Procedure (SAS/STAT, 2000).

The SURVEYMEANS Procedure takes into account the sample design used to select the survey sample and provides descriptive statistics of survey population such as means, standard error, and confidence limits. The sample design can be a complex survey sample design with stratification, clustering, and unequal weighting. The Taylor expansion method is used to estimate the variance. It computes sampling errors of estimators by obtaining a linear approximation for the estimators (Wolter 1985). The estimate of the variance is the variation among PSU when there are clusters. When a stratified design is used, the procedure pools stratum variance estimates to compute the overall variance estimate.

The SURVEYREG can handle complex survey sample designs by performing regression analysis for survey data. The procedure fits linear models for survey data and computes regression coefficients and their variance-covariance matrix. The procedure also provides the predicted values for the sample survey data, computes the regression coefficient estimators by generalized least squares estimation using element-wise regression, and calculates the significance tests for the model effects and for any specified estimable linear functions of the model parameters. The procedure assumes that the regression coefficients are the same across strata and the PSUs.

The SURVEYSELECT procedure provides a variety of methods for selecting probability-based random samples. The procedure can select a simple random sample or a sample according to a complex multistage sample design that includes stratification, clustering, and unequal probabilities of sample selection.

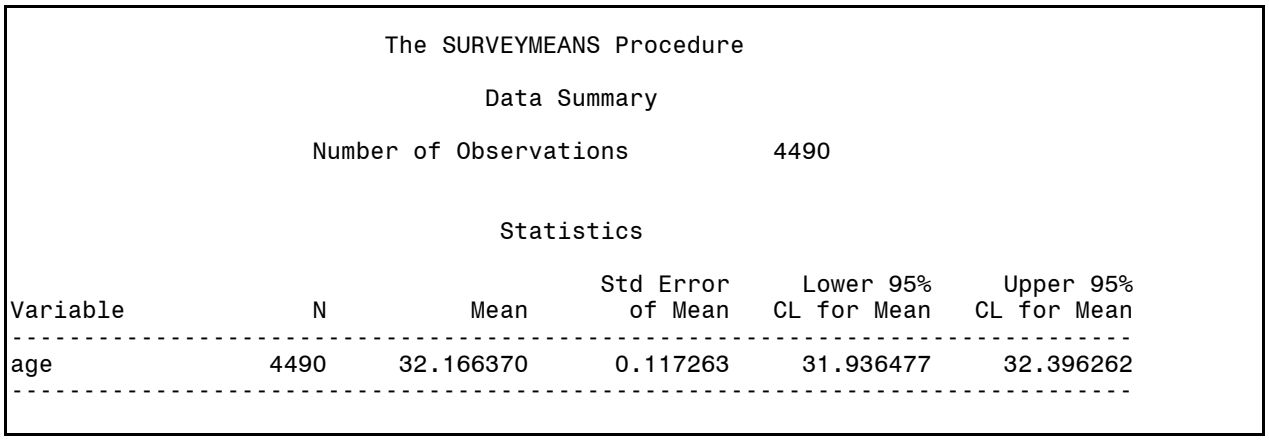
An Example

The variable AGE is chosen of a subsample of women with the age range of 18-46 years from the MEPS 1996 full-year data set. Procedure SURVEYMEANS is used to compute the mean, standard error, and 95% confidence limit for the variable age. Figure 1 shows the results of the analysis without using the sampling weights.

Quite often in complex surveys, respondents have unequal weights, which reflect unequal selection probabilities and adjustments for nonresponse and stratification. In such surveys, the appropriate sampling weights must be used to obtain valid estimates for the study population. Figure 2 illustrates the mean, standard error and 95% confidence interval for the variable AGE with unequal sampling weights. These statistics are different from the estimates shown in Figure 1 where the summary statistics are computed without the sampling weights. In Figure 2, the standard error is higher using the sampling weights. Finally, Figure 3 displays the analysis using the stratified information provided in the MEPS data set. Compared to the results in Figure 2, the estimates of the mean are the same. However, the standard error is larger when the number of strata are specified. When the design is stratified, the SURVEYMEANS procedure pools stratum variance estimates to compute the overall variance estimate (Figure 3) and the degrees of freedom equals the number of PSUs minus the number of strata in the sample design.

In summary, analyst using complex survey data, such as MEPS, to address their research questions must use

the procedures designed to deal with complex survey data. Otherwise, they risk the possibility of obtaining incorrect results. In particular, using standard SAS procedures for survey data that have a large variability results are biased estimates and will not accurately represent the survey population.



The SURVEYMEANS Procedure

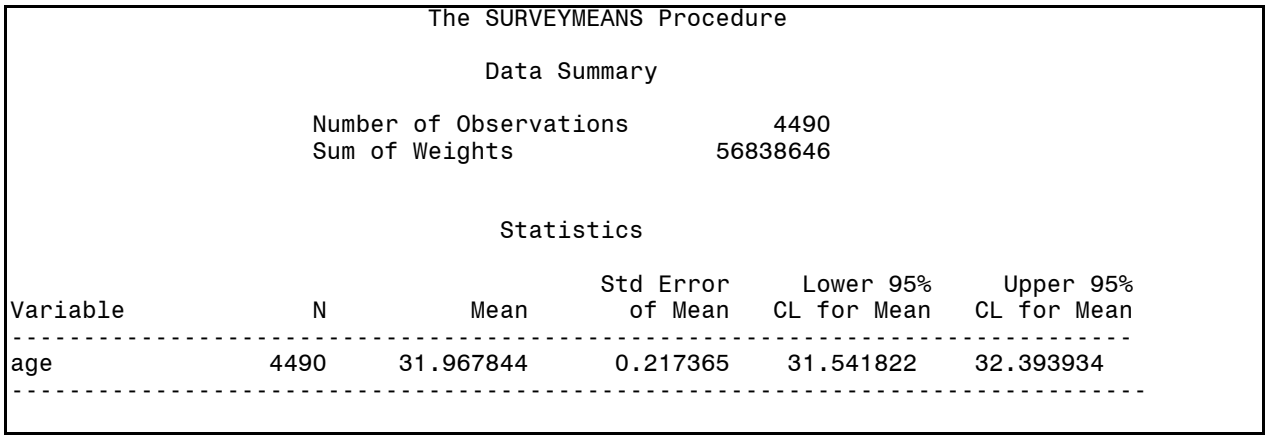
Data Summary

Number of Observations	4490

Statistics

Variable	N	Mean	Std Error of Mean	Lower 95% CL for Mean	Upper 95% CL for Mean
age	4490	32.166370	0.117263	31.936477	32.396262

Figure 1. Data Summary measures of variable age without the sampling weights



The SURVEYMEANS Procedure

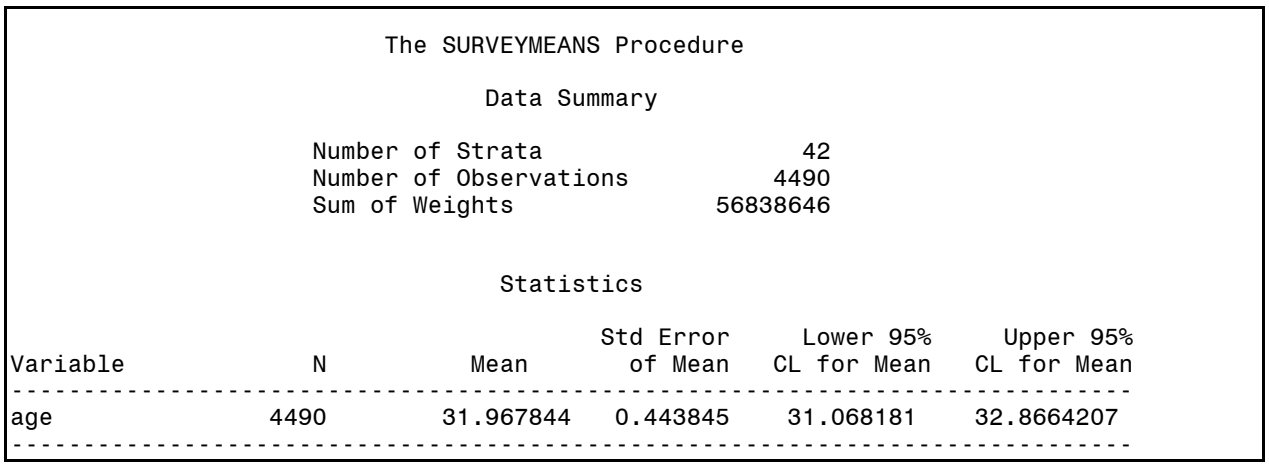
Data Summary

Number of Observations	4490
Sum of Weights	56838646

Statistics

Variable	N	Mean	Std Error of Mean	Lower 95% CL for Mean	Upper 95% CL for Mean
age	4490	31.967844	0.217365	31.541822	32.393934

Figure 2. Data Summary measures of variable age with the weights



The SURVEYMEANS Procedure

Data Summary

Number of Strata	42
Number of Observations	4490
Sum of Weights	56838646

Statistics

Variable	N	Mean	Std Error of Mean	Lower 95% CL for Mean	Upper 95% CL for Mean
age	4490	31.967844	0.443845	31.068181	32.8664207

Figure 3. Data Summary measures of variable age with the weights and the stratified information

Author Contact Information

Hossein N. Yarandi, Ph.D.*
Associate Professor
College of Nursing and Biostatistics Unit
University of Florida
Campus Box 100187
Gainesville, FL 32610-0187
Telephone: (353) 846-0658
Fax: (352) 846-1624
E-mail: yarandi@ufl.edu

Shawn M. Kneipp, Ph.D., ARNP
Assistant Professor
College of Nursing
University of Florida
Campus Box 100187
Gainesville, FL 32610-0187
Telephone: (353) 392-9207
Fax: (352) 846-1624
E-mail: skneipp@nursing.ufl.edu

* Correspondence should be directed.

References

Cohen, S. (1997). An evaluation of alternative PC-based software packages developed for the analysis of complex survey data. The American Statistician, 51, 293-299.

Cohen, S., Burt, V., and Jones, G. (1986). Efficiencies for variance estimation for complex survey data. The American Statistician, 40, 157-164.

Flyer, P. (1987). Finite population correction for replication estimates of variance. Proceedings of the Section on Survey Research Methods of the American Statistical Association, 732-736.

Flyer, P., Morganstein, D., and Rust, K. (1989). Complex survey variance estimation and contingency tables analysis using replication. Proceedings of the Section on Survey Research Methods of the American Statistical Association, 110-119.

Judkins, D. and Wright, D. (1990). National health interview survey: report on variance estimation. Rockville, MD: Westat.

Korn, E. & Graubard, B. (1999). Analysis of Health Surveys, New York: John Wiley & Sons.

MEPS HC-003 (1998). Population characteristics and utilization data for 1996, Agency for Health Care Policy and Research, Rockville, MD.

SAS Institute Inc. (2000). SAS/STAT User's Guide, Cary, NC: SAS Institute.

Wolter, K. (1985). Introduction to variance estimation, New York: Springer-Verlag.

Bootstrapping the Levene Test for Equality of Variances

Robert G. Stewart

East Tennessee State University

Since its introduction in 1960, Levene's test has remained prominent for testing the hypothesis of equal group variances. Indeed, many refinements have been proposed over the years to improve test robustness and power. Herein, the application of bootstrap methods to improve the robustness and power of Levene's test are discussed. A SAS macro (v6.12) for computing bootstrap versions of Levene's test is appended.

TUTORIALS

SECTION CHAIRS

Keith Cranford
Marquee Associates

Carla Mast
Transmedia Network, Inc

Joy Munk Smith
North Carolina State University

TUTORIALS

Paper P801

Conversion of SUDAAN® Output into Publication-Quality Tables—A Simplified Approach

Charlotte Cates Gard, Research Triangle Institute, Rockville, MD

ABSTRACT

We have developed a program that utilizes the capabilities of the SAS Output Delivery System and the SAS REPORT procedure to generate Rich Text Format tables from SUDAAN PROC CROSSTAB output. Because the SAS Output Delivery System allows style elements to be specified directly in PROC REPORT, the tables generated by our program are in "final form" and require little manipulation to meet formatting requirements. In addition, as most of the formatting is done "behind the scenes," the code is straightforward and can be easily modified for various table configurations.

INTRODUCTION

As the survey industry expands, more programmers are required to work with data from complex surveys. Traditionally, complex survey data have been analyzed using specialized software such as SUDAAN, WesVar, and Stata. Often, the output from these programs has required further manipulation to meet the specific formatting requirements of publication.

This paper addresses the conversion of SUDAAN output into publication-quality tables. Some previous solutions have relied on the transcription of data from printed output into pre-formatted tables. Other solutions have utilized sophisticated SAS® programs, combined with manipulations in a word processing program, to generate tables. Whatever the approach, producing more than a few such tables proved to be a time-consuming and tedious task.

THE DATA

The program presented here was used to analyze data from the 1999 National Youth Tobacco Survey. This survey, sponsored by the American Legacy Foundation, was designed to provide data on tobacco-related issues for a sample of students in grades six through twelve. More information on American Legacy Foundation research efforts can be obtained by visiting www.americanlegacy.org.

THE REQUIREMENTS

A number of formatting requirements were specified for the National Youth Tobacco Survey tables. The table below depicts many of these requirements.

Demographic Category	N	Never-Smoker, Not Open to Smoking	Never-Smoker, Open to Smoking
Age			
11 yrs	1784	72.9 [69.1 - 76.8]	14.3 [12.3 - 16.3]
12 yrs	2477	57.9 [54.6 - 61.2]	16.5 [14.8 - 18.2]
13 yrs	2712	44.8 [41.4 - 48.1]	16.5 [14.5 - 18.6]

Here, N represents the (unweighted) sample size, summed across all values of the response variable (i.e., across the columns). For each combination of a column variable and row variable, the (weighted) row percentage and its 95% confidence interval are displayed. Those familiar with SUDAAN will recall that the CROSSTAB procedure can be used to generate sample sizes and row percentages. Confidence intervals, however, are not available in PROC CROSSTAB and must be formed within SAS.

For this project, we were required to combine the output from multiple SUDAAN cross tabulations into a single table that could be read and manipulated in Microsoft Word. Most often, cross tabulations of a response variable with the demographic variables age, race, and gender were required. Header lines were desired, to separate the data from the various cross tabulations.

THE OLD WAY

The method initially used to generate tables for the National Youth Tobacco Survey relied on a combination of SAS programming and "by hand" manipulations in Microsoft Word. Although programmatically savvy, this approach proved to be quite labor intensive, particularly when a large number of tables were required.

The weakness of this approach lay in the hard coding of many table specifications. Changes to the specifications required that the programs be rewritten (or, at the very least, modified extensively). For instance, the number of levels of the response variable was utilized in key DATA step operations. As a result, separate programs had to be written for response variables with five levels, six levels, seven levels, etc. Because the programs relied on sophisticated programming techniques, they could be difficult for the novice programmer to understand and modify.

THE NEW WAY

With Release 8.1, SAS introduced into production its Output Delivery System Rich Text Format capability (this capability was available experimentally in Version 8 but was significantly improved with Release 8.1). With this, files created from SAS procedural output could be read and manipulated directly in Microsoft Word.

With this in mind, we sought to develop an approach that would utilize the new Rich Text Format capability to convert SUDAAN output into publication-quality tables quickly, accurately, and with modest programming effort.

PROC TABULATE VERSUS PROC REPORT

Originally, we intended to use the SAS TABULATE procedure to generate the required tables. However, we were unable to meet certain formatting requirements with this approach. For example, we could not create the confidence intervals in the desired format. PROC TABULATE would not allow us to display a character variable combining the (formatted) lower and upper confidence limits. We were, thus, forced to consider the lower and upper limits separately and, as a result, could not display them in the same cell, as required.

We also had problems displaying the lower and upper limits of the confidence interval below their corresponding point estimate. When we tried to do this, we ended up with a stack of cells for each row percentage—one containing the percentage itself, one containing the lower confidence limit, and one containing the upper confidence limit. In addition, with the confidence interval displayed below its corresponding point estimate, we lost the ability to display the (total) sample sizes, as with PROC TABULATE all statistics must be in the same dimension (either row, column, or page). Ultimately, the SAS REPORT procedure provided us a much better solution.

THE DETAILS

The program presented below uses SUDAAN PROC CROSSTAB to generate (weighted) cross tabulations of age, race, and gender with smoking stage. An output data set is created by SUDAAN, which is manipulated through a series of SAS DATA step

operations. SAS PROC REPORT operates on the modified data set to generate the required table.

SETUP

We begin our program with an OPTIONS statement. With the ORIENTATION = option, we specify the paper orientation to be used when printing to the Output Delivery System destination. Note that the OPTIONS statement can also be used to change the default paper size. This is often required when generating very large tables (tables with many columns) and can be achieved through use of the PAPERSIZE = option.

Next, formats are specified for the variables used in the cross tabulations. Formats for the demographic variables, QN1R, NEWRACE, and QN2R (which represent age, race, and gender, respectively) are as follows:

```
PROC FORMAT;

VALUE QN1RF 1 = "11 YRS"
            2 = "12 YRS"
            3 = "13 YRS"
            4 = "14 YRS"
            5 = "15 YRS"
            6 = "16 YRS"
            7 = "17 YRS"
            8 = "18-19 YRS";

VALUE NEWRACEF 1 = "WHITES"
               2 = "AFR-AMER"
               3 = "HISPANICS"
               4 = "ASIANS"
               5 = "OTHERS";

VALUE QN2RF 1 = "MALES"
            2 = "FEMALES";
```

The format for the response variable, STAGE, is also provided. Note the use of the "\" character in certain format values. This is called a split character and allows us to wrap these values for display in the final table.

```
VALUE STAGEF 1 = "NEVER-SMOKER,\NOT OPEN\TO
                  SMOKING"
             2 = "NEVER-SMOKER,\OPEN TO\SMOKING"
             3 = "EXPERIMENTER"
             4 = "FORMER SMOKER"
             5 = "NON-DAILY\CURRENT SMOKER"
             6 = "ESTABLISHED\SMOKER";
```

During setup, each of the above variables is created in the SUSC data set and the proper format is applied. (Note that LIBNAME statements, used to indicate storage locations for the input and output files, should also be specified during setup. They have been omitted here.)

MACRO VARIABLES

A number of macro variables are created, using the %LET statement, as below.

```
%LET COL_VAR = STAGE;
%LET LEVELS = 6;
```

COL_VAR specifies the response variable (here STAGE), and LEVELS specifies the number of levels of the response variable (here six). Although not shown here, macro variables are also used to specify the macro variables FILENAME and TITLE.

Note, with this particular approach, the %LET statement is the only place where the number of levels of the response variable is indicated. Otherwise, this program does not need to know the number of levels of the response variable. Therefore, if the response variable was redefined, resulting in a change in the number of levels, this is the only line of code that would have to be modified. Furthermore, if one wished to represent cross tabulations of age, race, and gender with another response variable (brand preference, for instance), he or she could do so simply by changing these %LET statements.

SUDAAN PROC CROSSTAB

We are now ready to run the SUDAAN CROSSTAB procedure. (Recall that, before using a SUDAAN procedure, we must sort the data set by the variables appearing on the NEST statement. This step is omitted here.)

```
PROC CROSSTAB DATA = YTS.SUSC FILETYPE = SAS
   DESIGN = WR;
NEST NSTRATUM NPSU / MISSUNIT;
WEIGHT FINALWT;
SUBGROUP QN1R NEWRACE QN2R &COL_VAR;
LEVELS 8 5 2 &LEVELS;
TABLES (QN1R NEWRACE QN2R) * &COL_VAR;
PRINT NSUM ROWPER SEROW;
OUTPUT NSUM ROWPER SEROW /
   FILENAME =OUTPUT.SUDOUT
   FILETYPE = SAS REPLACE;
RUN;
```

With the PRINT and OUTPUT statements, we request that the following statistics be output: NSUM (the unweighted sample size in each cell), ROWPER (the row percentage in each cell), and SEROW (the standard error of the row percentage in each cell). The PRINT statement requests that this information be printed to the screen, whereas, the OUTPUT statement requests that this information be written to a file (specifically to the file SUDOUT in the path specified by the LIBNAME statement for OUTPUT).

Note that PROC CROSSTAB generates a separate table for each cross tabulation requested. SUDAAN numbers the tables using the variable TABLENO. Here, TABLENO = 1 is the table for QN1R crossed with COL_VAR, TABLENO = 2 is the table for NEWRACE crossed with COL_VAR, and TABLENO = 3 is the table for QN2R crossed with COL_VAR.

A subset of the SUDOUT data set is provided below.

PROCNUM	TABLENO	QN1R	STAGE	NEWRACE	QN2R	NSUM	ROWPER	SEROW
1	1	0	0	-2	-2	14589	100	0
1	1	0	1	-2	-2	6024	40.29219	1.544229
1	1	0	2	-2	-2	1744	11.75898	0.560587
1	1	0	3	-2	-2	5278	36.50132	1.247278
1	1	0	4	-2	-2	173	1.314009	0.153495
1	1	0	5	-2	-2	388	2.860266	0.262266
1	1	0	6	-2	-2	982	7.273234	0.694255
1	1	1	0	-2	-2	1784	100	0
1	1	1	1	-2	-2	1310	72.93052	1.95798
1	1	1	2	-2	-2	248	14.30842	1.021948
1	1	1	3	-2	-2	223	12.5744	1.699181
1	1	1	4	-2	-2	1	0.051571	0.052627
1	1	1	5	-2	-2	1	0.070561	0.06979
1	1	1	6	-2	-2	1	0.064537	0.063832

DATA STEP MANIPULATIONS

A number of manipulations are required, to ready the SUDAAN output data set for SAS PROC REPORT.

```
DATA DATA1;
SET OUTPUT.SUDOUT;
IF &COL_VAR = 0 THEN DELETE;
GRP = TABLENO;
LCL = ROUND(ROWPER - (1.96 * SEROW), .1);
IF LCL < 0 THEN LCL = 0;
UCL = ROUND(ROWPER + (1.96 * SEROW), .1);
LENGTH CI $15 VALUE $51;
CI = "[" || COMPRESS(PUT(LCL, 5.1)) ||
   " - " || COMPRESS(PUT(UCL, 5.1)) || "]";
VALUE = PUT(ROUND(ROWPER, .1), 5.1) ||
   "                                " || CI;
RUN;
```

In the code above, we create a temporary data set, DATA1, from the SUDAAN output data set. We remove observations for which the macro variable COL_VAR is equal to zero. These observations represent the totals, summed across the columns. Although these totals are included in the final table, we will use PROC REPORT to recalculate them later.

Next, we create the variable GRP, which is equal to TABLENO. LCL and UCL, the (rounded) lower and upper confidence limits, are then calculated.

The character variable CI is formed by concatenating LCL and UCL (with leading and trailing brackets and an intervening dash). Similarly, the variable VALUE, is formed by concatenating the (rounded) value of ROWPER and the formatted confidence interval, CI.

```
DATA DATA2;
SET DATA1;
IF (QN1R = 0 or NEWRACE = 0 or QN2R = 0)
   THEN DO;
NSUM = .;
VALUE = "         ";
END;
RUN;
```

Here, we create a data set, DATA2, from DATA1. We identify those observations for which QN1R is equal to zero, NEWRACE is equal to zero, or QN2R is equal to zero. These represent the overall totals for QN1R, NEWRACE, and QN2R (summed across the rows). By setting NSUM and VALUE equal to missing (or blank) for these observations, we generate header rows (blank lines that appear before the age, race, and gender data in the final table).

```
DATA DATA3;
SET DATA1;
IF QN1R = 0 THEN DO;
GRP = 4;
NSUM = .;
VALUE = "         ";
OUTPUT;
END;
RUN;
```

Next, we create the data set DATA3, also from DATA1. We create a fourth group from those observations for which QN1R is equal to zero. This serves to add a blank line before the overall totals. (For convenience, we use observations for which QN1R is equal to zero. We might also have used observations for which NEWRACE is equal to zero or for which QN2R is equal to zero. The key is to have one observation for each (nonzero) value of COL_VAR.)

```
DATA DATA4;
SET DATA1;
IF QN1R = 0 THEN DO;
GRP = 5;
OUTPUT;
END;
RUN;
```

DATA4 is created from the data set DATA1 and contains the totals based on age. These totals will be used as overall totals in the final table.

```
DATA DATA5;
SET DATA2 DATA3 DATA4;
RUN;
```

Here, we merge the data sets we have created above. You will recall that DATA2 contains the age data, race data, and gender data, DATA3 contains a blank line (effectively), and DATA4 contains the overall (age) totals.

```
DATA DATA6;
SET DATA5;
IF GRP = 1 THEN TEMP = QN1R;
ELSE IF GRP = 2 THEN TEMP = NEWRACE;
ELSE IF GRP = 3 THEN TEMP = QN2R;
ELSE IF GRP = 4 THEN TEMP = QN1R;
ELSE IF GRP = 5 THEN TEMP = QN1R;
RUN;
```

With DATA6, we create the variable TEMP. Note that, for each GRP, TEMP is set equal to the variable from which the data were derived. (Recall that GRP = 1 is derived from TABLENO = 1, which is derived from the QN1R cross tabulation. GRP = 2 is derived from the NEWRACE cross tabulation, and GRP = 3 is derived from the QN2R cross tabulation. GRP = 4 and GRP = 5 are derived from QN1R.)

```
DATA DATA7;
SET DATA6;
BY GRP TEMP;
RETAIN LEVEL 0;
IF FIRST.TEMP THEN DO;
LEVEL = LEVEL + 1;
END;
RUN;
```

The GRP variable is used to distinguish between age data, race data, gender data, the blank line preceding the age totals, and the overall (age) totals. With this DATA step, LEVEL is incremented each time the value of TEMP changes. This effectively numbers the rows in the final table.

FORMATS

We now specify the format for the newly created variable LEVEL. To do this, we use the FORMAT procedure in SAS. Note that we have included here not only the values of each of the variables (QN1R, NEWRACE, and QN2R), but also headers for each of these variables.

```
PROC FORMAT;

VALUE LEVELF 1 = "AGE"
             2 = "11 YRS"
             3 = "12 YRS"
             4 = "13 YRS"
             5 = "14 YRS"
             6 = "15 YRS"
             7 = "16 YRS"
             8 = "17 YRS"
             9 = "18-19 YRS"
            10 = "RACE"
            11 = "WHITES"
            12 = "AFR-AMER"
            13 = "HISPANICS"
            14 = "ASIANS"
            15 = "OTHERS"
            16 = "GENDER"
            17 = "MALES"
            18 = "FEMALES"
            19 = "         "
            20 = "TOTAL";

VALUE MISSF  . = "         ";

RUN;
```

One disadvantage of this approach is that the rows of the table must be specified in the SAS program (and, thus, must be known in advance). This is necessary because we are combining cross tabulations for age, race, and gender in one table. If only one cross tabulation were required, this would not be necessary (and the number of DATA step manipulations would be greatly reduced).

STYLE DEFINITION

Next, we use the SAS TEMPLATE procedure to create a style definition. The SAS System provides a number of style definitions for use in formatting Output Delivery System output. These style definitions can be used as provided or can be modified to add new style elements or change existing style elements.

```
PROC TEMPLATE;
DEFINE STYLE STYLES.LEGACY;
PARENT = STYLES.RTF;
STYLE HEADER FROM HEADERSANDFOOTERS /
   BACKGROUND = WHITE;
STYLE SYSTEMTITLE FROM TITLEANDFOOTERS /
   FONT = ("TIMES", 10PT);
STYLE SYSTEMFOOTER FROM TITLEANDFOOTERS /
   FONT = ("TIMES", 10PT);
END;
RUN;
```

In the above code, we use PROC TEMPLATE to modify an existing style definition--STYLES.RTF. With the STYLE HEADER statement, we change the background color for the header from grey to white. With the STYLE SYSTEMTITLE and STYLE SYSTEMFOOTER statements, we specify the font type and size for the title and footnote.

The following code applies the style definition:

```
ODS LISTING CLOSE;
ODS RTF FILE = &FILENAME STYLE = LEGACY;
```

This code also tells the SAS System to close the Listing destination and to begin writing output to the Rich Text Format file specified by the macro variable FILENAME.

THE REPORT PROCEDURE

We are now ready to run PROC REPORT. PROC REPORT operates on DATA7, the final manipulated data set, to generate the formatted table.

```
PROC REPORT DATA = DATA7 SPLIT = "\" NOWD
   STYLE(COLUMN) = [FONT_SIZE = 8PT JUST = C];
COLUMNS GRP LEVEL NSUM &COL_VAR,VALUE N;
DEFINE GRP / GROUP NOPRINT;
DEFINE LEVEL / GROUP ORDER = INTERNAL
   FORMAT = LEVELF. " "
   STYLE = {CELLWIDTH = .75IN};
DEFINE NSUM / ANALYSIS FORMAT = MISSF. " "
   STYLE = {CELLWIDTH = .75IN};
DEFINE &COL_VAR / ACROSS ORDER = INTERNAL " "
   STYLE = {CELLWIDTH = 1IN};
DEFINE VALUE / DISPLAY " "
   STYLE = {CELLWIDTH = 1IN} FLOW;
DEFINE N / NOPRINT;
TITLE1 &TITLE;
FOOTNOTE "*DATA ARE FROM 1999 NATIONAL YOUTH
   TOBACCO SURVEY.";
RUN;
```

The NOWD option on the PROC statement runs PROC REPORT without the REPORT window and is required when using the Output Delivery System with PROC REPORT.

Recall that VALUE is a character variable combining each row percent with its corresponding (formatted) confidence interval. With &COL_VAR,VALUE in the COLUMNS statement, we are requesting that VALUE be displayed, for each level of COL_VAR, for each GRP and LEVEL combination.

The use of FLOW in the DEFINE statement for VALUE allows this particular column to flow to another line if it exceeds a specified width. This solves the problem of displaying the confidence interval below its corresponding point estimate as we have set our variable lengths and column widths to ensure that this flow takes place.

N, here, is a dummy variable. This program will not work without it. As you can see in the DEFINE statement for N, it is not actually printed.

Note that, in addition to the style elements specified in the style definition, we can also specify style elements directly in PROC REPORT. Here, we have specified the font size, the justification of text in the columns (centered), and the cell widths.

We must now tell the SAS System that we wish to stop writing the output to the Rich Text Format file. We do so with the following code:

```
ODS RTF CLOSE;
ODS LISTING;
```

This closes the Rich Text Format destination and reopens the Listing destination.

Appendix A displays the table that is generated by PROC REPORT.

"BY HAND" MANIPULATIONS

Although the table generated by our program satisfies the key formatting requirements, certain minor "by hand" manipulations must still be made. These include the following: the addition of a superscript character in the title and footnote, the addition of (one) gridline and (two) labels in the topmost header row, the application of italics to certain labels, and the application of shading to the header rows. In some cases, margins must also be adjusted.

Provided in Appendix B is the table generated by our program, modified slightly for publication.

CONCLUSION

With the introduction of the SAS Output Delivery System Rich Text Format capability, the opportunity exists to simplify the process of converting SUDAAN output into publication-quality tables. The program described above offers one approach for doing so. With it, we have realized significant reductions in the amount of time required to generate publication-quality tables for the National Youth Tobacco Survey.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Charlotte Gard
Research Triangle Institute
6110 Executive Boulevard, Suite 420
Rockville, MD 20852
Work Phone: (301) 770-8226
Fax: (301) 230-4646
E-mail: cgard@rti.org

APPENDIX A: Table A-2. Percentage Distribution of Smoking Stage*

		Never-Smoker, Not Open to Smoking	Never-Smoker, Open to Smoking	Experimenter	Former Smoker	Non-Daily Current Smoker	Established Smoker
Age							
11 yrs	1784	72.9 [69.1 - 76.8]	14.3 [12.3 - 16.3]	12.6 [9.2 - 15.9]	0.1 [0.0 - 0.2]	0.1 [0.0 - 0.2]	0.1 [0.0 - 0.2]
12 yrs	2477	57.9 [54.6 - 61.2]	16.5 [14.8 - 18.2]	24.6 [21.3 - 28.0]	0.1 [0.0 - 0.1]	0.3 [0.1 - 0.5]	0.6 [0.3 - 0.9]
13 yrs	2712	44.8 [41.4 - 48.1]	16.5 [14.5 - 18.6]	34.7 [31.6 - 37.9]	0.3 [0.1 - 0.5]	1.4 [0.8 - 2.1]	2.2 [1.5 - 3.0]
14 yrs	1895	33.0 [30.2 - 35.9]	14.2 [12.0 - 16.4]	43.4 [39.1 - 47.8]	1.0 [0.4 - 1.5]	2.8 [1.9 - 3.8]	5.6 [4.0 - 7.1]
15 yrs	1747	31.4 [27.6 - 35.3]	8.9 [7.4 - 10.4]	43.6 [40.2 - 46.9]	1.9 [1.1 - 2.7]	4.2 [3.0 - 5.4]	10.0 [7.8 - 12.1]
16 yrs	1761	24.8 [21.9 - 27.7]	6.7 [5.6 - 7.7]	47.9 [44.8 - 51.1]	3.0 [2.1 - 3.9]	4.8 [3.4 - 6.1]	12.8 [10.1 - 15.5]
17 yrs	1734	24.7 [21.6 - 27.7]	5.2 [4.0 - 6.5]	44.5 [41.0 - 47.9]	2.3 [1.6 - 3.0]	5.6 [4.1 - 7.1]	17.8 [14.0 - 21.5]
18-19 yrs	479	19.0 [14.2 - 23.8]	5.9 [3.4 - 8.3]	42.1 [37.4 - 46.9]	4.6 [2.5 - 6.6]	7.0 [5.1 - 9.0]	21.4 [15.6 - 27.2]
Race							
Whites	8345	40.6 [36.6 - 44.5]	11.7 [10.2 - 13.2]	32.9 [30.1 - 35.8]	1.6 [1.1 - 2.0]	3.7 [3.0 - 4.3]	9.5 [7.6 - 11.5]
Afr-Amer	2336	39.5 [35.5 - 43.4]	11.4 [9.9 - 12.9]	44.9 [41.2 - 48.6]	0.9 [0.4 - 1.4]	0.8 [0.4 - 1.2]	2.6 [1.6 - 3.5]
Hispanics	2623	39.0 [35.2 - 42.7]	12.4 [10.7 - 14.2]	42.0 [38.2 - 45.8]	0.9 [0.4 - 1.3]	1.9 [1.2 - 2.6]	3.8 [2.5 - 5.0]
Asians	538	54.4 [48.3 - 60.5]	11.6 [8.6 - 14.7]	25.6 [20.6 - 30.6]	1.3 [0.3 - 2.4]	1.3 [0.0 - 2.7]	5.8 [2.5 - 9.2]
Others	455	33.8 [27.4 - 40.2]	13.7 [8.7 - 18.8]	37.6 [31.8 - 43.3]	1.3 [0.1 - 2.4]	4.6 [1.4 - 7.8]	9.0 [4.9 - 13.2]
Gender							
Males	7265	39.0 [36.1 - 41.9]	11.7 [10.3 - 13.1]	36.5 [34.1 - 38.8]	1.7 [1.3 - 2.2]	3.3 [2.6 - 4.1]	7.7 [6.3 - 9.2]
Females	7267	41.5 [38.0 - 45.1]	11.8 [10.7 - 12.9]	36.6 [33.7 - 39.5]	0.9 [0.6 - 1.2]	2.4 [1.9 - 2.9]	6.8 [5.4 - 8.2]
Total	14589	40.3 [37.3 - 43.3]	11.8 [10.7 - 12.9]	36.5 [34.1 - 38.9]	1.3 [1.0 - 1.6]	2.9 [2.3 - 3.4]	7.3 [5.9 - 8.6]

*Data are from 1999 National Youth Tobacco Survey.

APPENDIX B: Table A-2. Percentage Distribution of Smoking Stage[1] (MODIFIED)

Demographic Category	N	Never-Smoker, Not Open to Smoking	Never-Smoker, Open to Smoking	Experimenter	Former Smoker	Non-Daily Current Smoker	Established Smoker
Age							
11 yrs	1784	72.9 [69.1 - 76.8]	14.3 [12.3 - 16.3]	12.6 [9.2 - 15.9]	0.1 [0.0 - 0.2]	0.1 [0.0 - 0.2]	0.1 [0.0 - 0.2]
12 yrs	2477	57.9 [54.6 - 61.2]	16.5 [14.8 - 18.2]	24.6 [21.3 - 28.0]	0.1 [0.0 - 0.1]	0.3 [0.1 - 0.5]	0.6 [0.3 - 0.9]
13 yrs	2712	44.8 [41.4 - 48.1]	16.5 [14.5 - 18.6]	34.7 [31.6 - 37.9]	0.3 [0.1 - 0.5]	1.4 [0.8 - 2.1]	2.2 [1.5 - 3.0]
14 yrs	1895	33.0 [30.2 - 35.9]	14.2 [12.0 - 16.4]	43.4 [39.1 - 47.8]	1.0 [0.4 - 1.5]	2.8 [1.9 - 3.8]	5.6 [4.0 - 7.1]
15 yrs	1747	31.4 [27.6 - 35.3]	8.9 [7.4 - 10.4]	43.6 [40.2 - 46.9]	1.9 [1.1 - 2.7]	4.2 [3.0 - 5.4]	10.0 [7.8 - 12.1]
16 yrs	1761	24.8 [21.9 - 27.7]	6.7 [5.6 - 7.7]	47.9 [44.8 - 51.1]	3.0 [2.1 - 3.9]	4.8 [3.4 - 6.1]	12.8 [10.1 - 15.5]
17 yrs	1734	24.7 [21.6 - 27.7]	5.2 [4.0 - 6.5]	44.5 [41.0 - 47.9]	2.3 [1.6 - 3.0]	5.6 [4.1 - 7.1]	17.8 [14.0 - 21.5]
18-19 yrs	479	19.0 [14.2 - 23.8]	5.9 [3.4 - 8.3]	42.1 [37.4 - 46.9]	4.6 [2.5 - 6.6]	7.0 [5.1 - 9.0]	21.4 [15.6 - 27.2]
Race							
Whites	8345	40.6 [36.6 - 44.5]	11.7 [10.2 - 13.2]	32.9 [30.1 - 35.8]	1.6 [1.1 - 2.0]	3.7 [3.0 - 4.3]	9.5 [7.6 - 11.5]
Afr-Amer	2336	39.5 [35.5 - 43.4]	11.4 [9.9 - 12.9]	44.9 [41.2 - 48.6]	0.9 [0.4 - 1.4]	0.8 [0.4 - 1.2]	2.6 [1.6 - 3.5]
Hispanics	2623	39.0 [35.2 - 42.7]	12.4 [10.7 - 14.2]	42.0 [38.2 - 45.8]	0.9 [0.4 - 1.3]	1.9 [1.2 - 2.6]	3.8 [2.5 - 5.0]
Asians	538	54.4 [48.3 - 60.5]	11.6 [8.6 - 14.7]	25.6 [20.6 - 30.6]	1.3 [0.3 - 2.4]	1.3 [0.0 - 2.7]	5.8 [2.5 - 9.2]
Others	455	33.8 [27.4 - 40.2]	13.7 [8.7 - 18.8]	37.6 [31.8 - 43.3]	1.3 [0.1 - 2.4]	4.6 [1.4 - 7.8]	9.0 [4.9 - 13.2]
Gender							
Males	7265	39.0 [36.1 - 41.9]	11.7 [10.3 - 13.1]	36.5 [34.1 - 38.8]	1.7 [1.3 - 2.2]	3.3 [2.6 - 4.1]	7.7 [6.3 - 9.2]
Females	7267	41.5 [38.0 - 45.1]	11.8 [10.7 - 12.9]	36.6 [33.7 - 39.5]	0.9 [0.6 - 1.2]	2.4 [1.9 - 2.9]	6.8 [5.4 - 8.2]
Total	14589	40.3 [37.3 - 43.3]	11.8 [10.7 - 12.9]	36.5 [34.1 - 38.9]	1.3 [1.0 - 1.6]	2.9 [2.3 - 3.4]	7.3 [5.9 - 8.6]

[1]Data are from 1999 National Youth Tobacco Survey.

Paper P-802

ODS, YES! Odious, NO! – An Introduction to the SAS Output Delivery System

Lara Bryant, University of North Carolina at Chapel Hill, Chapel Hill, NC
Sally Muller, University of North Carolina at Chapel Hill, Chapel Hill, NC
Ray Pass, Ray Pass Consulting, Hartsdale, NY

ABSTRACT

ODS (originally pronounced 'odious', but now pronounced 'ya gotta love it') is the new SAS System facility, starting with Version 7, that you can use to format your PROC and DATA output in ways just recently only dreamed about. ODS offers greatly enhanced flexibility and ease of use for both new SAS users and experienced SAS users new to ODS. This paper will discuss the basics of ODS, emphasizing methods of converting standard PROC output to the following "destinations":

- **Listing** - default (the only way to get PROC output up to now)
- **HTML** - HyperText Markup Language (probably the best tool available for information exchange today)
- **Output** - SAS data sets (no more PROC PRINTTO!)
- **Printer** - available experimentally in V7, and for production in V8. Produces both Postscript and PCL output on all hosts, and on PC hosts additionally produces output for any printer supported by the host operating system. Note with Version 8.1, PDF (Postscript Display Format) is also available as production.

- **RTF** - for importing into MS Word. (in production now with V8.1, but not covered in this paper.

For more information on RTF see:

http://www.sas.com/rnd/base/news/odsrtf/index.html

Also not covered in this paper, the following destinations are available as experimental in Version 7 and Version 8:

- **LaTex** - a driver that produces your output marked up using LaTex.
- **XML** - a driver that produces XML
- **HTML STYLESHEET** - lets you use HTML CSS (Cascading Style Sheets)

For more information on these experimental destinations see:

http://www.sas.com/rnd/base/topics/expv8/index.html

Prior to ODS, all SAS output results were lumped together in a single "listing" output. With the advent of ODS, each PROC now produces one or more data components which are then combined with different formats to produce one or more output objects. These output objects are then sent to one or more destinations as defined above. In this paper we will demonstrate how you select the output objects to send to each destination, and the syntax for each destination. By the end of the paper you will have a working knowledge of ODS and feel comfortable enough to easily create at least three new kinds of output in SAS!

INTRODUCTION

Creating output objects that can be sent to destinations (e.g. HTML) is often just a matter of running procedures in your existing SAS program with just a few extra lines of code (sometimes only one line). When you run a procedure or DATA step, ODS combines the resulting data with a template (or table definition) to create an output object, or a series of output objects coming from various parts of the procedure's output. ODS allows you to choose specific output objects created from a procedure or DATA step to send to an output destination. ODS provides default table definitions for most (but not all!) procedures and for the DATA step. You can also create or modify your own table definition with PROC TEMPLATE. The output object is formatted according to its content and the destination you send it to. You can also send your output to more than one destination. For example, you can create an output data set from a PROC MEANS procedure that is also displayed on an HTML page.

OPENING AND CLOSING DESTINATIONS

The Listing, HTML, and Output destinations can be open or closed. By default, the Listing destination is open, and the HTML, Output, and Printer destinations are closed. The statement for opening the Listing destination is:

```
ods listing;
```

The commands for opening the HTML, Output, and Printer destinations are more detailed, and therefore are presented in this paper in the overview of each destination. To close a destination, the syntax is

```
      ods <destination> close;
e.g.  ods listing close;
```

You may want to close the Listing destination to free up resources that would otherwise be used to send output objects to this destination.

SELECTION AND EXCLUSION LISTS

For each destination, the SAS System maintains a list of the objects that are to be sent there. The SAS System also maintains an overall list of objects that are to be sent to all open destinations. If you are selecting objects to send to a destination, SAS maintains a SELECTION list. If you are selecting objects that you do not want sent to a destination, SAS maintains an EXCLUSION list for that destination. Generally you need only select or exclude objects for a particular destination, rather than trying to maintain both a SELECTION and an EXCLUSION list for that destination. The same holds true if you are creating an overall selection or exclusion list -- you only need one or the other.

There are two ways that these SELECTION and EXCLUSION lists can be modified:

- explicit modification from a command by you
- automatic modification by ODS at certain points (step boundaries) in the SAS program

For more information on step boundaries see "SAS Language Reference Concepts Version 8," pg. 271.

Explicit Modification

To explicitly modify the overall SELECTION and EXCLUSION lists, you may use the following syntax:

```
ods <options>;
```

To explicitly modify a specific destination's SELECTION and EXCLUSION lists, you may use the following syntax:

```
ods listing <options>;
ods html    <options>;
ods printer <options>;
```

where the options are

```
select  <specific output objects>
select  all
```

```
select  none
exclude <specific output objects>
exclude all
exclude none
```

The default values for the destinations are as follows:

```
Overall list       - select all
Listing destination - select all
HTML destination - select all
Printer destination - select all
Output destination - exclude all
```

Changing the overall list is helpful if you want to exclude an object from all destinations. For example, rather than typing,

```
ods html exclude all;
ods printer exclude all;
```

You could simply type:

```
ods exclude all;
```

Automatic Modification

When you do NOT explicitly modify a SELECTION or EXCLUSION list, ODS automatically sets defaults as noted above at every step boundary. (A step boundary signals the end of the preceding step, for instance a "run;" statement or a "quit;" statement or a new DATA or PROC step.) When you do use explicit modification, ODS, by default, maintains the modifications for *one use only*, reverting back to defaults at the boundary. This can be overcome by using the PERSIST option.

Persist Option with SELECT OR EXCLUDE

Consider the following code:

```
ods listing select BasicMeasures;

proc univariate data='A:/meddat';
run;
proc univariate data='A;/meddat';
run;
```

As a result of this code, ODS would select only the "BasicMeasures" statistics for the first PROC UNIVARIATE. The RUN statement ends the procedure (this is a step boundary, but even if you did not specify the RUN statement, the beginning of the second PROC UNIVARIATE would end the first PROC UNIVARIATE). Either way, you would only have "BasicMeasures" printed for the first PROC. After the first PROC, the list is automatically set to its default value, which is SELECT ALL for the default Listing destination. The second PROC would therefore include all statistics generated by the PROC UNIVARIATE. Obviously, if you only wanted "BasicMeasures" throughout, it would be tedious to have to specify the desired list after every procedure. ODS provides a way around this. By adding the PERSIST option to the SELECT/ EXCLUDE statement, you only have to specify the SELECTION/ EXCLUSION list once for it to be maintained throughout the program (or at least until the next encountered SELECT or EXCLUDE command). So if we run the following code

```
ods listing select BasicMeasures (persist);
proc univariate data='A:/meddat';
run;
proc univariate data='A;/meddat';
run;
```

the BasicMeasures statistics will be selected for both PROC UNIVARIATEs. The PERSIST option can also be used for an HTML or Printer list, for example:

```
ods html select BasicMeasures (persist);
```

The PERSIST syntax for the Output destination is more involved, and is explained in the Output destination section of this paper.

Resetting Lists for RUN-Group Processing

In the previous examples, a RUN statement would end the PROC and reset the SELECTION/EXCLUSION list to the default value if the PERSIST option was not specified. However, there are several procedures that are not terminated by a RUN statement (RUN-Group processing), such as PROC DATASETS, PROC GLM, and PROC REG. In these cases, unless a QUIT statement is encountered, the PROC will continue to run. This may produce some unexpected results on your SELECTION/EXCLUSION list. For example, consider the following code, (the FILE= option is discussed below in "file types"):

```
ods html file='A:/new.htm';
ods html select Anova;

proc reg data='A:/aggrec';
   model inpdol=age;
run;

ods html select FitStatistics;

proc reg data='A:/aggrec';
   model outpdol=age;
run;

ods html close;
```

In the program above, ODS would create "Anova" statistics for the first PROC REG. This would remain intact through the RUN statement because a RUN statement does not end a running PROC REG. When ODS reaches the second PROC REG, it would end the first PROC and set the SELECTION list to its default value of SELECT ALL. Therefore, rather than having the desired "FitStatistics" for the last PROC REG, ODS would create ALL the statistics. The simple solution is to specifically end the first PROC REG with a QUIT statement as follows (the SHOW statement is discussed below):

```
ods html file='A:/new.htm';
ods html select Anova;

proc reg data='A:/aggrec';
   model inpdol=age;
run;
ods html show;
quit;

ods html show;
ods html select FitStatistics;

proc reg data='A:/aggrec';
   model outpdol=age;
run;

ods html show;
quit;

ods html close;
```

This program produces the desired results: "Anova" statistics for the first PROC REG and "FitStatistics" for the second PROC REG.

ODS SHOW STATEMENT

At any point in your program, you can use the ODS SHOW to see what ODS has on the SELECTION/EXCLUSION list for a specific destination. The following syntax,

```
ods <destination> show;
```

requests that the SELECTION/EXCLUSION list for a particular destination appear in the log. If no destination is specified, the OVERALL list is displayed. In the example immediately above, the log would contain

```
Current HTML select list is:
1. Anova
```

after the first SHOW statement, and

```
Current HTML select list is:
1. FitStatistics
```

after the last one.

ODS TRACE STATEMENT

Part of the power of ODS is that you can indicate which output objects to create, and even tell ODS to send the output objects created by the same procedure to different destinations. For example, rather than all of the statistics, you may want only the mean, standard deviation, and median generated by the PROC UNIVARIATE. However, in order to specify which output objects to select, you must know the name of the object produced by your SAS program. ODS provides a method of viewing the name of each output object created. The syntax,

```
ods trace on < / listing | label > ;
```

displays a "trace record" of the name and other information about each output object produced by the program in the SAS log. The LISTING option instructs ODS to put the trace record directly above the output to which it refers. This is extremely useful for determining the name and path of the output objects of interest. The LABEL option instructs ODS to include the label path in the trace record. The code below illustrates both the TRACE statement and the LABEL option:

```
ods trace on / label;
proc univariate;
    var meddol;
run:
ods trace off;
```

The SAS Log that results from this program is shown below. Note that you not only get the name of the output object, but since you specified the label option, you also get the label path of the output object:

```
Output Added:
-------------
Name:       Moments
Label:      Moments
Template:   base.univariate.Moments
Path:       Univariate.meddol.Moments
Label Path: "The Univariate Procedure"."meddol"
            "Moments"
-------------

Output Added:
-------------
Name:       BasicMeasures
Label:      Basic Measures of Location and
            Variability
Template:   base.univariate.Measures
Path:       Univariate.meddol.BasicMeasures
Label Path: "The Univariate Procedure"."meddol"
            "Basic Measures of Location and
            Variability"
-------------

Output Added:
-------------
Name:       TestsForLocation
Label:      Tests For Location
Template:   base.univariate.Location
Path:       Univariate.meddol.TestsForLocation
Label Path: "The Univariate Procedure"."meddol"
            "Tests For Location"
-------------
```

```
Output Added:
-------------
Name:       Quantiles
Label:      Quantiles
Template:   base.univariate.Quantiles
Path:       Univariate.meddol.Quantiles
Label Path: "The Univariate Procedure"."meddol".
            "Quantiles"
-------------

Output Added:
-------------
Name:       ExtremeObs
Label:      Extreme Observations
Template:   base.univariate.ExtObs
Path:       Univariate.meddol.ExtremeObs
Label Path: "The Univariate Procedure"."meddol".
            "Extreme Observations"
-------------
```

Fortunately, although you can then specify the output object by using the full path name, you can also specify the output object by using any part of the path that begins immediately after a period and continuing to the end. For example, if you want to send the Quantiles and Moments for all variables to a web page, you could enter,

```
ods html select quantiles moments;
```

or if you just want the Quantiles and Moments for the MEDDOL variable only:

```
ods html select meddol.quantiles
                meddol.moments;
```

The label path can be used in the same way. You can also specify an output object with a mixture of labels and paths, such as

```
ods html select meddol."quantiles";
```

Often it is easier to select the variables in the PROC step, and the desired statistics in the ODS step. For example, rather than typing,

```
ods html select meddol.quantiles
               inpdol.quantiles
               hosp.quantiles
               ambul.quantiles;
proc univariate;
run;
```

an easier method that gives the same results would be

```
ods html select quantiles;
proc univariate;
   var meddol inpdol hosp ambul;
run;
```

Note, once you "turn on" the ODS TRACE ON statement in your SAS session, ODS will continue to write trace records until you issue the statement ODS TRACE OFF. This means that if you start a new procedure or even a new program in the same SAS session, TRACE ON will be in effect, until you turn it off. Also note that you must have a run statement between the TRACE ON and TRACE OFF statements in order for the trace record to be created.

ODS HTML DESTINATION

File types

The HTML destination can produce four kinds of files (web pages):

1) **BODY** file: This is a required file that contains the output object(s) generated from the PROCs or DATA steps. Basically, this is where you store the results that will ultimately be displayed on your HTML report or web site. If your SAS job creates an output object that is routed to an HTML destination, ODS places the results within HTML <TABLE>

tags, where they are stored as one or more HTML tables. If your SAS job creates a graphic object, the BODY file has an <IMG> (image) tag that references graphic output objects. Note that the BODY file can be specified with either the BODY= or the FILE= parameter.

2) **CONTENTS** file: The CONTENTS file contains a link to each of the output objects that are stored in the BODY file, and is specified by the CONTENTS= parameter.

3) **PAGE** file: This is useful if you have a lot of output, and you do not want it to all be stored on one long page. The PAGE file contains a link to each separate page (of the BODY file) of HTML output that ODS creates from a PROC or DATA step. The PAGE file is similar to the CONTENTS file, except that the CONTENTS file has a link to each output object, whereas the PAGE file has a link to each page of output that is created. The CONTENTS and PAGE files will be identical if you specify in the NEWFILE parameter that you would like each output object placed on a separate BODY file. An example that illustrates the NEWFILE parameter is presented later in this paper. You specify the PAGE file with the PAGE= parameter.

4) **FRAME** file: Provides a simultaneous view of all files included in the ODS HTML statement. You specify the FRAME file with the FRAME= parameter.

The syntax for creating these files is

```
ods html file-type = 'file-specification'
                     <(parameters)>;
```

Here is an example of ODS HTML statements which generate a BODY file and a CONTENTS file:

```
ods html body     = 'c:\temp\body.htm'
         contents = 'c:\temp\contents.htm';
```

In the above code, the BODY file could also have been specified with a FILE= parameter. Note that the BODY file, and only the BODY file, is *required* as an HTML output destination.

Additional HTML Parmeters

1) **PATH=** : As mentioned, you use the BODY= parameter to tell ODS where to store an HTML file. In addition, you can use PATH= to tell ODS in what directory to store all the HTML files that you create. The PATH= option may refer to an external (quoted) file specification, a SAS fileref or a SAS libname.catalog. For example,

```
ods html path = 'C:\MyDocuments'
         body = 'body.htm'
         contents = 'contents.htm';
```

Note that if you use the PATH= statement, you must do so before specifying the HTML pages.

2) **URL=** sub-parameter: You can improve on PATH= by including a Uniform-Resource-Locator (URL) sub-parameter that will use the given URL instead of the file name for all the links and references that it creates to the file. This is helpful if you want to create a FRAME file, and/or will be moving the files around. For example:

```
ods html path = 'C:\MyDocuments'
         (url = 'http://www.unc.edu/~jismith')
         body = 'body.htm'
         contents = 'contents.htm';
```

Note that the URL= sub-parameter of the PATH= option is enclosed in parentheses. You can also specify the URL= sub-parameter in the parameter for the BODY file, as in the following:

```
ods html path ='C:\MyDocuments '
         body ='body.htm'
        (url ='http://www.unc.edu/~jismith');
```

The results will be identical.

3) **ANCHOR=** : Each output object in the BODY file is identified by an HTML <ANCHOR> tag. These anchor tags allow the CONTENTS, PAGE and FRAME files to link to, or reference the output objects in the BODY file. You can change the base name for the HTML anchor tags with the ANCHOR= parameter. The syntax for this option is:

```
anchor = 'anchor-name';
```

Since each anchor name in a file must be unique, ODS will automatically "increment" the name that you specify. For example, if you specify

```
anchor = 'tabulate';
```

ODS names the first anchor TABULATE. The second anchor is named TABULATE1; the third is named TABULATE2, and so on. The anchor names are only of interest to you if you need to write to the HTML page; otherwise you need not concern yourself with them. However, you do need to remember to *always* specify a new anchor name each time you open the BODY file so that the same anchor tags are not written to the file again.

4) **NO_TOP_MATTER** and **NO_BOTTOM_MATTER** parameters: These parameters circumvent the default action of writing some HTML to the top and bottom of the file that is open for HTML output. The benefit of these parameters is that the HTML BODY page is "cleaner" when viewed by the browser.

5) Descriptive text parameter: This parameter allows you to include comments in between the output of your PROCs. You specify the descriptive text inside parentheses next to the BODY=, CONTENTS=, PAGE=, or FRAME= options. Adding comments to your HTML page is helpful for many reasons. For example, you might like to point out some of the interesting results you obtained.

EXAMPLE 1, Putting it all together. The following code places output from several procedures on the same HTML page and uses many of the HTML parameters discussed above - including incorporating descriptive text between the output objects. The SAS statements are numbered for comments following the code:

```
 1)  libname  health 'C:Data';
 2)  filename web    'C:\Data\body.htm';
 3)  ods listing close;
 4)  ods html path = 'C:\Data'
          (url = 'http://www.unc.edu/~jismith/')
         body = web (no_bottom_matter);
 5)  proc univariate data=health.meddat;
        var inpdol outpdol;
 6)  run;
 7)  ods html close;
 8)  filename web 'C:\Data\body.htm' mod;
 9)  data _null_;
 10)    file web;
 11)    put '<h3> We want to put comments in
             after the first procedure. </h3>';
 12) run;
 13) ods html body   = web (no_top_matter
                            no_bottom_matter)
              anchor = 'univ';
 14) proc freq data=health.meddat;
        table site;
 15) run;
 16) ods html close;
 17) data _null_;
        file Web;
        put '<h3> We also want comments after
             the second procedure is run.</h3>';
 18) run;
 19) ods html body   = web (no_top_matter)
              anchor = 'freq';
 20) ods html close;
```

And now the comments:

(1) Identifies the location of the SAS catalog (C\:Data) containing the SAS data set used for the PROCS.

(2) The FILENAME statement creates a fileref (WEB) for the BODY file, where all the output will be stored. Recall the default list for HTML is SELECT ALL. Since no selection commands are specified, everything included in the program will be sent to the HTML file at 'C:\Data\body.htm'

(3) The Listing destination is closed to free up resources.

(4) The NO_BOTTOM_MATTER option suppresses any default HTML at the bottom of ' C:/Data/body.htm'

(5) All statistics created by PROC UNIVARIATE will be generated for the variables INPDOL and OUTPDOL.

(6) Remember that a RUN statement goes after the PROC, and before closing the HTML destination.

(7) The HTML destination must be closed to append to it later.

(8) This references the BODY file used above, and MOD indicates that we want to append to the file.

(9) This DATA _NULL_ step writes some descriptive HTML code to the BODY file via the PUT and FILE statements.

(13) This opens the HTML destination 'C:\Data\body.htm' as identified by the fileref WEB, and suppresses any default HTML code on the top and bottom of the file. The ANCHOR= option creates a base name for the HTML anchor tags. You should *always* specify a new anchor name each time you open the BODY location so that the same anchor tags are not written to the file again.

(19) Open the HTML destination again in order for the new output to be written to the HTML file. The ANCHOR statement provides a new base name.

The resulting HTML page is shown below. The name of the HTML file that is created is 'body.htm' and it is stored in 'C:\Data'. Since we did not specify a template, the default template is used.

The UNIVARIATE Procedure
Variable: INPDOL

Moments			
N	1000	Sum Weights	1000
Mean	247.50903	Sum Observations	247509.03
Std Deviation	1354.43565	Variance	1834495.93
Skewness	9.68499218	Kurtosis	119.342445
Uncorrected SS	1893922159	Corrected SS	1832661439
Coeff Variation	547.226762	Std Error Mean	42.831016

Basic Statistical Measures			
Location		Variability	
Mean	247.5090	Std Deviation	1354
Median	0.0000	Variance	1834496
Mode	0.0000	Range	23018
		Interquartile Range	0

Tests for Location: Mu0=0				
Test	Statistic		p Value	
Student's t	T	5.778734	Pr > |t|	<.0001
Sign	M	43.5	Pr >= |M|	<.0001
Signed Rank	S	1914	Pr >= |S|	<.0001

Quantiles (Definition 5)	
Quantile	Estimate
100% Max	23018.39
99%	5618.30
95%	1538.71
90%	0.00
75% Q3	0.00
50% Median	0.00

25% Q1	0.00
10%	0.00
5%	0.00
1%	0.00
0% Min	0.00

Extreme Observations			
Lowest		Highest	
Value	Obs	Value	Obs
0	1000	11367.8	722
0	999	12175.4	983
0	998	13119.0	162

The UNIVARIATE Procedure
Variable: OUTPDOL

Moments			
N	1000	Sum Weights	1000
Mean	111.39919	Sum Observations	111399.19
Std Deviation	248.217815	Variance	61612.0838
Skewness	9.53237587	Kurtosis	139.188553
Uncorrected SS	73960251.2	Corrected SS	61550471.7
Coeff Variation	222.81833	Std Error Mean	7.84933652

Basic Statistical Measures			
Location		Variability	
Mean	111.3992	Std Deviation	248.21782
Median	44.0000	Variance	61612
Mode	0.0000	Range	4523
		Interquartile Range	107.15000

Tests for Location: Mu0=0				
Test	Statistic		p Value	
Student's t	t	14.19218	Pr > |t|	<.0001
Sign	M	390.5	Pr >= |M|	<.0001
Signed Rank	S	152685.5	Pr >= |S|	<.0001

Quantiles (Definition 5)	
Quantile	Estimate
100% Max	4522.680
99%	970.105
95%	412.975
90%	267.525
75% Q3	117.150
50% Median	44.000
25% Q1	10.000
10%	0.000
5%	0.000

1%	0.000
0% Min	0.000

Extreme Observations			
Lowest		Highest	
Value	Obs	Value	Obs
0	998	1338.90	221
0	997	1415.49	58
0	989	1627.62	147
0	978	3535.44	414
0	975	4522.68	415

We want to put comments in after the first procedure.

The FREQ Procedure

SITE	Frequency	Percent	Cumulative Frequency	Cumulative Percent
1	1000	100.00	1000	100.00

We also want comments after the second procedure is run.

(Continued Additional HTML Parmeters)

6) **NEWFILE=** parameter: In the HTML output shown above, you might have wanted a separate page (file) for each table, rather than having all tables on the same page. For this purpose, you can use NEWFILE= to specify the starting point for each new BODY file. The syntax for this option is,

```
newfile = <starting point>;
```

where a starting point can be:

NONE – write all output to the BODY file that is currently open
OUTPUT – start a new BODY file for each output object
PAGE – start a new BODY file for each page of output
PROC – start a new BODY file for each new procedure
BYGROUP – start a new BODY file for each new bygroup.

Just as ODS "increments" the name of the anchor, ODS will also automatically increment the names of the new files. For example, if the original BODY file is named RESULTS, each new BODY file that is created based on the NEWFILE parameter will be called RESULTS1, RESULTS2, etc.

7) **PAGE=** parameter: If the NEWFILE= parameter is specified, you may also want to include the PAGE= parameter in your HTML statements:

```
page=<file-specification> ;
```

The file specified will contain a description of each page of the BODY file as well as links to the BODY files.

EXAMPLE 2. Putting it all together (again). The following program illustrates the NEWFILE= parameter and the PAGE= parameter, as well as some HTML options already discussed.

```
   libname health 'C:/Data';
1) ods listing close;
2) ods html
       path    ='C:/Data'
       (url    ='http://www.unc.edu/~jismith')
       file    ='file.htm'
       contents='contents.htm'
       frame   ='frame.htm'
       page    ='page.htm' (no_top_matter)
       newfile =page;
3) ods html select Moments;
   proc univariate data=health.meddat;
      var dentdol drugdol ;
   run;
   proc print data=health.meddat;
      var site person contyr;
   run;
4) ods html close;
```

(1) The first ODS statement closes LISTING as a destination for output. This is done to conserve resources

(2) The following things happen in this ODS statement:

- the PATH= specifies where to store your HTML files;
- the URL= sub-parameter tells ODS to use this URL for links and references;
- the FILE= tells ODS the location of the body file;
- the CONTENTS= tells ODS to use this file for links to the body file for every HTML table that is created in a PROC or DATA step.
- the FRAME= parameter puts all files included in the ODS HTML statement on one screen.
- the PAGE= parameter tells ODS to use this file to store links to the BODY file for every page of HTML that ODS creates from a PROC or DATA step.
- the NEWFILE= option tells ODS to create a new BODY file for each new page of output. In this case this would mean a new file of output for each variable in the UNIVARIATE procedure and a new file for the PROC PRINT output. The name of each new file is based on the name specified by FILE= option. The BODY files that are created in this example are FILE.HTM, FILE1.HTM, and FILE2.HTM.

(3) This ODS statement instructs ODS to send only the 'Moments' statistics from the PROC UNIVARIATE to the HTML output

destination.

(4) The final ODS statement closes the HTML destination in order for output to be sent there.

Results of EXAMPLE 2

All files created from the above example are shown on the next page. The Table of Contents file comes from the CONTENTS= 'CONTENTS.HTM' parameter. Each of the references under the procedure titles is a hypertext link to the location of the respective table in the BODY file.

Below the Table of Contents file is the Table of Pages file created from the PAGE= 'PAGE.HTM' (NO_TOP_MATTER) option. Each page reference (PAGE 1, PAGE 2, PAGE 3) is a hypertext link to that page in the BODY file.

Next to the Table of Contents file and the Table of Pages file are each of the BODY files that are created. The first BODY file that is created is called FILE.HTM, and it contains the Moments data for the variable DENTDOL. This page is created first because DENTDOL is the first variable listed in the PROC UNIVARIATE, and PROC UNIVARIATE is the first PROC in the program. The second BODY file created is called FILE2.HTM and it contains the Moments data for DRUGDOL. The last page, FILE3.HTM is from the PROC PRINT (note: not all observations are included in order to conserve space).

By clicking on a reference on either the Table of Contents display or the Page file display, we can link to each of the BODY files. The Frame page, FRAME.HTM, combines the PAGE file, CONTENTS file, and whichever BODY file you create, onto one page.

We have reviewed a few of the parameters used with the HTML destination. Others can be found in the "The Complete Guide to the SAS® Output Delivery System, Version 8."

ODS OUTPUT DESTINATION

The ODS OUTPUT statement is used to specify an action, or to create one or more data sets. When you first start SAS Version 7 or Version 8, the Output destination is closed and the exclusion list is set to EXCLUDE ALL. You can change these default actions with the ODS OUTPUT statement.

Specifying an action

When used to specify an action, the syntax of the ODS OUTPUT statement is

```
ods output <action>;
```

where the action choices are

CLEAR - set the list for the OUTPUT destination to EXCLUDE ALL.
SHOW - display the selection or exclusion list that apply at this point in the program in the SAS log
CLOSE - close the OUTPUT destination. Once the destination is closed, you cannot send output to this destination.

Creating output data sets

You open the Output destination by specifying the data set(s) that you would like created. To create a single output data set, the syntax is,

```
ods output <output-object> = <sas data set>;
```

where the output-object can be identified with the use of the ODS TRACE statement. This example illustrates creating an Output file:

```
ods listing;

ods trace on / listing;
ods output BasicMeasures = measures;

proc univariate data = meddat;
      var meddol suppdol;
run;
ods trace off;
```

Here we have both the Listing destination and the Output destination open.. Although all the PROC UNIVARIATE statistics will be sent to the Listing destination (whose default value is SELECT ALL), only the BasicMeasures statistics will be sent to the Output destination. You can look in the log or in the SAS Explorer window (in this case in the WORK library), to see that ODS has created the MEASURES data set. This newly created SAS data set will have the BasicMeasures statistics for both the MEDDOL and SUPPDOL variables. By looking in the Results window and clicking on BasicMeasures, you will see in the Output window the BasicMeasures statistics. Unlike the HTML destination, you do not have to close the Output destination to have objects sent there.

To create a separate output data set for each variable used in a procedure or data step, use the following syntax:

```
ods output <output-object> (match_all)
           = <sas data set>;
```

For example, the following code will select the OneWayFreqs statistics from the PROC FREQ. The OUTPUT statement creates a different data set for each variable in the PROC FREQ procedure because of the MATCH_ALL option, and bases the name of these data sets on the name STATS:

```
ods output onewayfreqs (match_all) = stats;
proc freq data='A:/test';
run;
ods output close;
run;
```

When this program is run, the data sets created are STATS, STATS1, STATS2 ... STATSN for however many variables there are in the data set TEST. You may find, however, that you would like to combine the data sets for each variable into one data set. This is easily done.

contents.htm:

Table of Contents

- The Univariate Procedure
  ·DENTDOL
  ·Moments
  ·DRUGDOL
  ·Moments
- The Print Procedure
  ·Data Set IN.AGGREC

page.htm:

Table of Pages

- The Univariate Procedure
  ·Page 1
  ·Page 2
- The Print Procedure
  ·Page 3

file.htm:

The UNIVARIATE Procedure
Variable: DENTDOL

Moments			
N	100	Sum Weights	100
Mean	52.38	Sum Observations	5238
Std Deviation	226.742535	Variance	51412.1774
Skewness	8.17753631	Kurtosis	73.6960078
Uncorrected SS	5364172	Corrected SS	5089805.56
Coeff Variation	432.879984	Std Error Mean	22.6742535

file2.htm:

The UNIVARIATE Procedure
Variable: DRUGDOL

Moments			
N	100	Sum Weights	100
Mean	14.8721	Sum Observations	1487.21
Std Deviation	34.3546782	Variance	1180.24391
Skewness	3.32771071	Kurtosis	13.1047585
Uncorrected SS	138962.083	Corrected SS	116844.147
Coeff Variation	231.000855	Std Error Mean	3.43546782

file3.htm:

Obs	SITE	PERSON	CONTYR
1	1	MA250247	01
2	1	MA250247	02
3	1	MA250247	03
4	1	MA250247	04
5	1	MA250247	05
6	1	MA250255	01
7	1	MA250255	02
8	1	MA250255	03
9	1	MA250255	04
10	1	MA250255	05
11	1	MA250263	01
.	.	.	.
.	.	.	.
.	.	.	.
93	1	MA25162A	05
94	1	MA251638	01
95	1	MA251638	02
96	1	MA251638	03
97	1	MA251638	04
98	1	MA251638	05
99	1	MA251646	01
100	1	MA251646	02

First you create a macro variable, which stores a list of the data sets that are created in the ODS OUTPUT statement. In a separate DATA step, you combine the data sets by concatenation. This is illustrated in the example below.

```
ods output OneWayFreqs (match_all=name)=stats;
```

To concatenate the data sets, you specify

```
data all;
   set &name;
run;
```

A little advice about using the MATCH_ALL option. In the following program, separate data sets are created for the Moments data, but not for the Basic Measures data. All the variables are included in one data set for the Basic Measures statistics.

```
ods output BasicMeasures Moments
           (match_all)=moments;
```

To create output separate output data sets for both sets of statistics, we would need to specify:

```
ods output BasicMeasures (match_all)= measures
           Moments (match_all) = moments;
```

To create a permanent data set with the OUTPUT statement, use the following syntax:

```
libname in 'A:/';
ods output BasicMeasures = in.measures;
```

OUTPUT Parameters

PERSIST parameter: This parameter is useful if you are creating output data set and want the data set definition to endure even when the procedure or DATA step ends, until you explicitly modify the list. The syntax is:

```
ods output
output-object<(MATCH_ALL<=macro-var-name>
PERSIST=PROC|RUN)>=<SAS-data-set> ;
```

The PERSIST parameter specifies when to close any data sets that are being created, and when to remove output objects from the SELECTION list for the OUTPUT destination. The PERSIST parameter can only be used in conjunction with the MATCH_ALL parameter.

PROC argument: The PROC argument to the PERSIST parameter preserves the list of definitions that are specified in the ODS OUTPUT statement across step boundaries. This means that the list of output objects specified in the ODS OUTPUT statement is preserved even after the procedures or DATA steps have completed. You must explicitly modify the list to change the definitions; e.g. with

```
ods output exclude all;
```

RUN argument: The RUN argument to the PERSIST parameter serves exactly the same function as the PROC statement, as it also keeps the data sets open. The following is an example of a program that implicitly uses the RUN argument but does not specify the PERSIST option (although it was intended to).

```
ods output OneWayFreqs(MATCH_ALL=name)=stats;
proc freq data=health.test;
run;
proc freq data=health.test2;
run;
```

In this case, the data sets are not created for the second PROC FREQ. Without the PERSIST option, the second procedure is treated as a step boundary, and a data set for the variables in HEALTH.TEST2 is not created. This is easily corrected by explicitly specifying:

```
ods output OneWayFreqs(match_all=name
           persist=proc) = stats;
```

CONCLUSION

The purpose of this paper was to help you get started using the Output Delivery System. Our examples illustrate that you can create web pages with the addition of as little as one line of code to your existing SAS program. With the addition of a single OUTPUT statement you can also create one or more SAS data sets. With a few additional words you can select your object objects and send them to more than one destination at a time. The best way to convince yourself, though, is to visit our website and submit the examples presented in this paper for yourself. The website can be found at:

http:// www.unc.edu/~lkbryant/odsworkshop

REFERENCES

SAS® Version 8 Software.

SAS® Institute, The Complete Guide to the SAS® Output Delivery System, Version 8

ACKNOWLEDGEMENTS

We wish to thank Paul Kent, Base SAS R&D Director, and Chris Olinger, Base SAS Software Manager, for teaching classes on these subjects at our UNC site and at local SAS users group meetings. We also wish to thank the SAS Technical Support Division for answering our questions so promptly regarding items in this paper.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors at:

Lara K. Bryant
Jordan Institute for Families
CB 3550
301 Pittsboro St.
Chapel Hill, NC 27599
Email: lbryant@email.unc.edu

Sally S. Muller
Jordan Institute for Families
CB 3550
301 Pittsboro St.
Chapel Hill, NC 27599
Email: sally@email.unc.edu
Work: 919-843-7798
Fax: 919-967-7015

Ray Pass
Ray Pass Consulting
5 Sinclair Place
Hartsdale, NY 10530
Email: raypass@att.net
Work: 914-693-5553
eFax: 914-206-3780

Changes & Enhancements for ODS by Example (through Version 8.2)

Sandy McNeill, SAS®, Cary, NC
Presented by David Kelley

ABSTRACT

The purpose of this paper is to show through example some of the new ODS features and options that have been added for SAS Version 8.2. Some of the highlights are: production PDF destination; template options which allow the invocation of a macro or evaluation of an expression using values from other columns on the same row; in-line formatting for the PRINTER and RTF destinations which allows a finer control over page breaks, subscripting, superscripting, formatting of particular words in a cell instead of the entire cell; and style control for titles and footnotes for the PRINTER and RTF destinations. This paper will talk about each of these topics along with some examples.

INTRODUCTION

With each new version of SAS, the ODS developers stay hard at work adding new features or improving existing features to hopefully make your jobs easier at producing output the way you need it from SAS®. This paper will describe and show examples of some of these new features that we have added for the latest version of SAS Software®.

PDF DESTINATION

I debated calling this a new production destination since some users have been using this in its experimental form, but PDF is now a production destination. This is native PDF without the need for the Adobe Acrobat Distiller. Prior to Version 8.2, you could obtain PDF output by outputting to a postscript file and then invoking the Adobe Acrobat Distiller to distill the postscript into PDF. However, now there is no need for the distiller since we write the native PDF code. The syntax for this is very easy:

```
ods Printer PDF file='foo.pdf'
```

Where foo is, of course, the filename.

Even though you no longer need a distiller, you will still need some type of viewer to view the PDF output. A common viewer for the PCs is Adobe® Acrobat Reader and a common viewer for Unix is ghostview.

STYLE CONTROL FOR TITLE/FOOTNOTE STATEMENTS

In SAS Version 8.1, some of the graph options allowed on the title and footnote statements were passed on to HTML. What this allowed was the stylistic customization of your title and footnote statements such as changing the font, color, height, and justification. Keep in mind that I am talking about titles and footnotes in HTML located OUTSIDE the graph image. If indeed you are running a graph proc, use the options NOGTITLE and NOGFOOTER to place the titles and footnotes outside of the graph image.

In addition to the options on the title/footnote statement, some of the GOPTIONS were also respected. Unfortunately, we ran out of time for Version 8.1 and did not get to add this functionality into all the destinations. The good news is that this same capability is now available in Version 8.2 for the Printer and RTF destinations. The graph options that are respected on the title/footnote statements are: COLOR, FONT, HEIGHT, and UNDERLIN. The GOPTIONS that are supported are: FTITLE, FTEXT, GUNIT, HTITLE, and HTEXT.

Here is an example using the PDF destination with several of these options being used:

```
Title font=courier height=8pt color=blue
     j=left 'Catch the ' font=times
     height=12pt 'Wave';
ods printer pdf file='catch.pdf';
proc print data=sashelp.class;run;
ods printer close;
```

The GOPTIONS work as documented in the SAS/Graph® documentation. The values set in the GOPTIONS will be used unless the corresponding style attribute is specified on the title/footnote statement. In that case, the value actually specified on the title/footnote statement will win. For example, if I set the GOPTION HTITLE=5pt, this is setting the HEIGHT style attribute. If you do not specify a HEIGHT option on your TITLE statement, then the height of your text will be 5 points. However, if you specify HEIGHT=10pt on the TITLE statement, then the HEIGHT option on the TITLE statement supercedes the GOPTION. Now if you are familiar with ODS and style templates, you are probably asking yourself "But what about the style attributes specified in the style template? The order of precedence is:

Style template
Goptions
Option specified on the title/footnote statement

Here's what this means: Say you are using a style with a particular font size for titles and footnotes (the default font size for titles is 5 for HTML (styles.default) and 13pt for the PRINTER destinations (styles.printer)). So let's say we are using the PRINTER destination with a default font size of 13pt. If you do not specify HTITLE in the goptions and if you do not specify HEIGHT on the title statement, then the title will be in 13pt. However, if you specify HEIGHT=8pt on the title statement, then the font size for the title will be 8 pt. If you use GOPTIONS HTITLE=8pt and do not use a HEIGHT option on the title statement, then the size will be retrieved from the goptions statement. The goptions HTITLE overrides the font size in the ODS style that is being used.

One caveat to the interaction with the ODS styles: by default, the font for the titles and the footers in the default ODS style template is BOLD and ITALIC. We received some feedback from SAS/GRAPH users that since there is no option on the title or footnote statement to remove the BOLD or ITALIC, but there are options to turn on those attributes, they would like the BOLD and ITALIC turned off automatically when a FONT is specified. This only is in effect when you use the options NOGTITLE or NOGFOOTER on the ODS statement. You can, of course, turn off the BOLD and ITALIC for all your titles and footnotes by modifying the style template.

Here's an example of using the default style and specifying a font which will turn off the BOLD and ITALIC that is received from the default style. :

```
Ods html file='foo.html' nogtitle ;
Title font=COURIER 'This is a title';
```

```
proc gchart data=sashelp.class;
vbar sex;
run;
ods html close;
```

If you want to change the font for a particular title, but you still want BOLD and ITALIC, then just use those options on the title statement.

```
Ods html file='foo2.html' nogtitle;
Title font=courier bold italic 'Bolded
Title';
Proc gchart data=sashelp.class;
Vbar sex;
Run;
Ods html close;
```

Now on to other examples.

Here is an example in which I am creating a new style called SmallTFfont. I am going to be using the RTF destination, so this style will inherit from Styles.RTF and it will specify a font size of 6 pt for both the title and the footnote statements.

```
Proc template ;
Define style SmallTFfont;
Parent = Styles.RTF;

Style SystemTitle from SystemTitle /
    Font = ("Arial, Helvetica", 6pt, bold );
Style SystemFooter from systemFooter /
    Font = ("Arial, Helvetica", 6pt, bold );
End;
Run;
```

TITLE CUSTOMIZATION EXAMPLE 1

The font size of the title statement will be 6pt because we are using the new smallTFfont that we defined above. There are two text strings with one of them left justified and the other one right justified. We did not specify a font, so the font weight will be BOLD because it is picking up the BOLD from the SmallTFfont. The foreground color of the text will be BLACK, which the style SmallTFfont inherits from Styles.RTF which inherits from Styles.Printer.

```
Title j=right 'WAVE'  j=left 'Catch the';
Ods rtf file='titles1.rtf' style=smallTFfont;
Proc Print data=sashelp.class; run;
Ods rtf close;
```

TITLE CUSTOMIZATION EXAMPE 2

The font size of the title statement will be 9pt because now we are specifying a HEIGHT option on the title statement. This HEIGHT option will override the font size that was specified in the style SmallTFfont. All the rest of the attributes will remain the same as in Title Example 1. The height attribute remains 9pt until the end of the statement or until another height option is specified.

```
title HEIGHT=9pt  j=right 'WAVE' j=left
     'Catch the';
Ods rtf file='titles2.rtf' style=smallTFfont;
Proc Print data=sashelp.class; run;
Ods rtf close;
```

TITLE CUSTOMIZATION EXAMPLE 3

Now we are using a font option. Because of this, the BOLD font weight will be cleared. The only attributes that change in this example are the font name and the font weight which is no longer BOLD.

```
title HEIGHT=9pt FONT=courier j=right
     'WAVE' j=left 'Catch the';
Ods rtf file='titles3.rtf' style=smallTFfont;
Proc Print data=sashelp.class; run;
Ods rtf close;
```

TITLE CUSTOMIZATION EXAMPLE 4

We don't like a title statement that is not bolded, so we are going to turn the font weight BOLD back on. The rest of the attributes will remain the same. There is no option to turn the bold off, so once you turn it on, it is on for the remainder of the statement.

```
title BOLD HEIGHT=9pt FONT=courier j=right
     'WAVE' j=left 'Catch the';
Ods rtf file='titles4.rtf' style=smallTFfont;
Proc Print data=sashelp.class; run;
Ods rtf close;
```

TITLE CUSTOMIZATION EXAMPLE 5

Better example of interaction between the style template and the title statement. Create a new style and modify the SYSTEMTITLE style element such that the text of the titles will be the color blue. Then use some options on the title statement and see how the color of the titles is blue unless the color is overridden on the statement. "Catch" is blue from the blue foreground set in the SystemTitle. 'the' is green from the title statement color option. 'WAVE' is blue again from the style template and the varying heights from the title2 statement

```
proc template;
define style ex5; parent = styles.rtf;
style SystemTitle from SystemTitle /
     foreground = blue;
end;
run;

title j=left font='Comic Sans MS'
     'Catch ' color=green 'the';
title2 j=center font='Comic Sans MS'
     height=24pt 'W'
     height=20pt 'A'
     height=16pt 'V'
     height=12pt 'E';
ods rtf file='titles5.rtf' style=ex5;
Proc Print data=sashelp.class;run;
Ods rtf close;
```

IN-LINE FORMATTING AND PAGE BREAK CONTROL

The PRINTER and RTF destinations now support the ability to insert simple formatting text within a given cell or paragraph. The types of formatting available are: specifying a style option to customize the font, color, etc; specifying a superscript and subscript; specifying a dagger or sigma character; specifying raw text to be inserted into the document for the open destination or you can specify that the raw text be targeted to a particular destination.

To use any of these special formatting tricks, you need to use an escape character. By default, '03'x is expected as the escape character. However, using this everywhere you want special formatting is a pain, so we added a special ODS statement just for specifying the escape character that you would like to use.

```
Ods escapechar = '\';
```

This statement above would specify the backslash character as the escape character to be used when specifying the use of in-line formatting.

Assuming we are using the above statement and have specified the backslash as our escape character, here is a table of the formatting codes and their definitions.

\w	Preferred line break. If the line breaks, it breaks there; but if there's enough room, it won't break
\z	Error code. Formats the output like a SAS error message. Also available: \1z = error \2z = warning \3z = note \4z = fatal
\m	Set a mark. The position is remembered and if the line wraps, it will wrap to this position.
\-2n	Wrap to mark
\n	Wrap to left margin
\[arg]n	Wrap where the value specified for the arg indicates how far to advance vertically.
\S = {style attributes}	Lets you specify style attributes. See discussion about the style option in this paper under New Template Features, or see the Style statement documentation for Proc Template in the on-line documentation.
\{super text}	Superscript what is denoted by text
\{sub text}	Subscript what is denoted by text
\R</tag>"raw-text"	Insert raw text into document for the current output destination. The tag is optional and represents a particular destination. If the tag value is present, then the raw text only appears in the output for the specified destination. The values for tag would be: RTF, PDF, PS

I am going to go over an example, but for more information see the paper written by Brian Schellenberger (the in-line formatting guru) at http://www.sas.com/rnd/base/topics/odsprinter/qual.pdf.

IN-LINE FORMATTING EXAMPLE

Example of In-Line formatting using both the Postscript destination and the RTF destination. This example demonstrates the ability to subscript, superscript, and change the formatting of words within a particular cell.

```
data b;
   x= "\3z\mIn 8.2, you can put " ||
      "\S={font_weight=bold}bold\S={} " ||
      "and " ||
      "\S={font_style=italic}italic\S={} " ||
      "text into your cells. It's " ||
      "\S={font_style=italic}n\S={}" ||
      "\{super 2} the fun." ||
      "\-2nWhere do we end up?";
   y=2;
   z=3;
   run;

proc template;
define style foo;
parent = styles.rtf;
style systemtitle from systemtitle /
   protectspecialchars = off;
end;
run;

ods escapechar = '\';

ods printer ps file='InLineFormatEx.ps';
ods rtf file='InLineFormatEx.rtf'
     style = foo;

proc report nowd data=b;
   title2 '001 PROC REPORT: Using Functions';
   title3 'In a title too:
        \{super super} \{sub sub}';
   title4 '\{dagger} \{sigma}';
   run;

   ods rtf close;
   ods printer close;
```

A couple things are worth pointing out about this example. This example uses the MARK (escapechar m) formatting; however, this will currently only work for the PRINTER destinations such as postscript and PDF. If you run this example and look at the RTF output, you'll see that the second line "Where do we end up" is not indented, but it IS indented in the PRINTER destinations. Another difference with the destinations is that for RTF, you must tweak the SystemTitle style element and turn off the PROTECTSPECIALCHARS attribute if you want to use the in-line formatting in the title statement (or tweak SystemFooter for footnote statements). You don't, however, have to do this for the PRINTER destinations. We are looking into making this more consistent for Version 9.

CONTROLLING PAGEBREAKS FOR RTF AND PRINTER

Another option that has been added to the PRINTER and the RTF destinations is the ability to control page breaks. Currently, each procedure call starts a new page. Sometimes you don't want that, and with the number of questions that we have received asking if it is possible to control the page breaks, I'd say the number of you wanting this functionality is pretty high.

The option to control page breaks for the PRINTER and the RTF destinations is STARTPAGE= and is an option on the ODS statement. The values for this option are: NOW, ON, OFF. So if you use

Ods printer ps file='foo.ps' startpage=no;

The implicit page breaks before each procedure will be suppressed. We will, however, still go to a new page when we run out of room on the page.

PAGE BREAK CONTROL EXAMPLE

This example is to demonstrate the usage of the STARTPAGE option to control page breaking. On our initial ODS statement, we specify STARTPAGE=NO which tells ODS to suppress the page breaks which you would normally get for each procedure invocation. Before the third PROC PRINT, we issue another ODS statement, but this time with STARTPAGE=NOW. This will start a new page at the next procedure call, but after that the page breaks are still suppressed until a STARTPAGE=YES or a STARTPAGE=NOW is processed.

```
Data x;a=1;b=2;c=3;run;
Ods printer ps file='PrinterPageBreakEx.ps'
     Startpage = no;
/* Page 1 */
Proc print data=x; run;
Proc Print data=x; run;
Ods printer startpage = now;
/* Page 2 */
Proc print data=sashelp.class; run;
Proc print data=x; run;
Ods printer startpage = yes;
/* Page 3 */
Proc print data=x; run;
/* Page 4 */
Proc print data=x; run;
Ods printer close;
```

TEMPLATE OPTIONS

Several new options have been added to Proc Template for Version 8.2. One of these options is the ability to use a format name as the attribute value in the STYLE option. This functionality has been available in Proc Tabulate in Version 8.1,

but for consistency this has now been made available in Proc Report, Proc Template, and in the new Proc Print styles. The nice this about this is that you can write your user-defined formats for traffic lighting, and then use these formats across several different procedures.

Here is a Proc Template example to illustrate the use of creating a user-defined format and then using that format as the value for a style attribute. One of the reasons for using this technique is to perform traffic lighting. This example uses the dataset exprev and the formats found under Proc Print style example 3.

```
proc template;
   define table RegionReport;
      define column Region; end;
      define column State;end;
      define column month;end;
      define column revenues;
         style = {background=revfmt.
                flyover=revfly.};
      end;
   end;
run;

ods html file='TemplateEx1.html';
ods listing close;

title 'Regional Activity';
data _null_;
   set work.exprev;
   file print ods=(template='RegionReport'
       columns=(region
                state
                month
                revenues));
   put _ODS_;
run;

ods html close;
ods listing;
```

A second new functionality available in Proc Template is the EXPRESSION function which can be used to evaluate an expression which is given as the value for a style attribute. The expression to be evaluated within the parentheses follows the same rules as expressions used in the TRANSLATE and the CELLSTYLE…AS functions. The powerful thing about this is that you can get to the values in other columns on the same row instead of just being able to use the global symbol _VAL_ to access the value of the current cell.

Here's a second Proc Template example which illustrates the use of the EXPRESSION function. In this example, we use both an expression and the global symbol _VAL_ as parameters in the EXPRESSION function. The global symbol _VAL_ is the current value of the cell. Notice in the expression how we are accessing the values from other columns.

```
ods listing close;

ods html file='TemplateEx2.html';

data colors;
  length lightness saturation hue $ 20;
  input lightness $ & saturation $ & hue $ &;
  datalines;
dark  grayish  blue
moderate  reddish  purple
dark  moderate  red
medium  strong  orange
light  greenish  yellow
strong  bluish  green
;

proc template;
   define table colortab;
   define header tabhd;
      text 'Expression function';
      style={foreground=#FF00FF};
   end;
   column lightness saturation hue
       lightnesshue lightnessSaturHue;
   define lightness;
      header='Lightness';
   end;

   define saturation;
      header='Saturation';
   end;

   define hue;
      header='Hue';
      style={background=expression("_val_")};
   end;

   define lightnesshue;
      compute as lightness||' '||hue;
      header='Foreground is lightness and
                  hue';
      style = {background =
         expression("lightness||' '||hue")};
   end;

   define lightnessSaturHue;
      compute as lightness || ` ` ||
                saturation || ' ' || hue;
      header='Background is lightness
              saturation hue';
      style = {background =
              expression("lightness ||
                  ' ' || saturation ||
                  ' ' || hue")};
   end;
end;

data _null_;
   set colors;
   file print ods=
     ( template='colortab'
       columns(
         lightness
         saturation
         hue
        )
      );

   put _ods_;
run;

ods _all_ close; /* closes ALL the open */
         /* destinations - even listing */
ods listing;     /* reopen the listing */
```

A third new functionality that's very cool is the ability to invoke a macro by using the RESOLVE function. &_Val_ is a global macro variable which is set with the value of the current cell before the call to the macro. This lets you use the value of the current cell within the macro.

This third Proc Template example illustrates the invocation of a macro called DOURL which is evaluated and create the attribute value for the URL style attribute. After running this example, you should see that each of the columns headers is a unique link.

Column A will have a link to http://www.yourcompany.com/a, column B will have a link to http://www.yourcompany.com/b, and column C will have a link to http://www.yourcompany.com/c.

```
data x;
   a=1;
   b=2;
   c=3;
   run;

%macro dourl;
   http://www.yourcompany.com/&_val_
%mend;

proc template;
   define table testit;
   column a b c;

   define a;
      define header hdra;
         style={url = resolve('%dourl')};
      end;
      header = hdra;
   end;

   define b;
      define header hdrb;
         style={url = resolve('%dourl')};
      end;
      header = hdrb;
   end;

   define c;
      define header hdrc;
         style={url = resolve('%dourl')};
      end;
      header = hdrc;
   end;
end;
run;

ods listing close;
ods html file='TemplateEx3.html';
data _null_;
   set x;

   file print ods=
     ( template='testit'
       columns(
         a
         b
         c
        )
      );

   put _ods_;
 run;

ods html close;
ods listing;
```

PROC PRINT STYLE OPTION

For Version 8.2, Proc Print now has a style option which allows style customizations just like Proc Report, Proc Tabulate, and Proc Template. The general form of the style option is exactly the same as for the other Procedures:

STYLE<(location-name(s))> =
<style-element-name>[style-attribute-specification(s)]

Where style-attribute-specification(s) is:

```
Style-attribute-name = Style-attribute-value
```

Here are tables which show the values that can be used for the LOCATION-NAMES and also the default style element that is in effect if you do not specify any STYLE-ELEMENT-NAME. These tables are broken down by what is valid for different statements, so there is a table for the Proc Print statement, a table for the VAR statement, a table for the ID statement, and a table for the SUM statement.

Location-Names and Default-Style-Element-Names for the PROC PRINT Statement

Location Name(s)	Affects	Default Style Element
DATA COLUMN COL	Default for all data cells of all columns	Data
HEADER HEAD HDR	Default for all header cells of all columns	Header
OBS OBSDATA OBSCOLUM N OBSCOL	Data cells of OBS column	RowHeader
OBSHEADER OBSHEAD OBSHDR	Header of OBS column	Header
TOTAL TOT BYSUMLINE BYLINE BYSUM	The SUM line containing totals for each BY group	Header
BYLABEL BYSUMLABEL BYLBL BYSUMLBL	The label for the by-variable on the line containing SUM totals	Header
GRANDTOTAL GRANDTOT GRAND GTOTAL GTOT	The SUM line containing the grand totals for the whole report	Header
TABLE REPORT	Output data table (not byline)	Table

Location-Names and Default-Style-Element-Names for the VAR Statement

Location Name(s)	Affects	Default Style Element
DATA COLUMN COL	Data Cells	Data
HEADER HEAD HDR	The column header cell	Header

Location-Names and Default-Style-Element-Names for the ID Statement

Location Name(s)	Affects	Default Style Element
DATA COLUMN COL	Data Cells	RowHeader
HEADER HEAD	The column header cell	Header

HDR		

Location-Names and Default-Style-Element-Names for the SUM Statement

Location Name(s)	Affects	Default Style Element
DATA COLUMN COL	Data cells	Data
HEADER HEAD HDR	The column header cell	Header
TOTAL TOT BYSUMLINE BYLINE BYSUM	Data cell containing sum	Header
GRANDTOTAL GRANDTOT GRAND GTOTAL GTOT	Data cell containing sum over whole report	Header

Some of the more widely used style-attribute-names are:

- Font
- Font_face
- Font_size
- Font_style
- Font_width
- Foreground
- Background
- URL
- Just
- Preimage
- Postimage

A complete list of the style-attribute-names is documented in "The Complete Guide to the SAS Output Delivery System", specifically within the Style Statement section under The TEMPLATE Procedure. These style-attribute-names are the same attribute names used wherever you might see the STYLE statement.

So what does all this mean? How do you use it? Better to explain this with examples. I am going to use the HTML destination for these examples.

PROC PRINT STYLE EXAMPLE 1

Make the background of the OBS column and the background of the column headers the color blue.

```
ods html file='PrintStyleEx1.htm';
Proc Print data=sashelp.class
   style(OBS HEADER OBSHEADER) =
            {background=blue};
Run;
Ods html close;
```

PROC PRINT STYLE EXAMPLE 2

A little fancier. Shows output using by and pageby.

```
proc sort data=sashelp.class out=class;
     by sex age;
run;
ods html file='PrintStyleEx2.htm';

proc print data=class n='Number of
observations for the sex and age '
     style(N)={foreground=black
               font_weight=bold}
     style(OBS HEADER OBSHEADER) =
        {background = mediumred
         foreground = black}
     /*TOTAL makes the whole TOTAL line red*/
     style(TOTAL) = {background = white
                     foreground = mediumred}
     style(GRANDTOTAL) = {background = BLACK
                          foreground = white}
     style(BYSUMLABEL)= {background=mediumred
                  foreground = white
                  font_weight = bold};
var name height weight ;
sum height weight;
by sex age ;
pageby sex;
run;
ods html close;
```

PROC PRINT STYLE EXAMPLE 3

Since the by-line is generated outside of Proc Print, the style of the by-line is not controllable from Proc Print. You would have to modify the style element ByLine in the style template that you are using. In these HTML examples, the default style template is styles.default. However, a way around this with Proc Print is to use the ID statement and use the same variables on the ID statement as on your BY statement.

Use ID statement, which gets rid of the byline. Notice that I removed the style for the OBS and the OBSHEADER and placed a style on the ID statement.

```
proc sort data=sashelp.class out=class;
     by sex age;
run;
ods html file='PrintStyleEx3.htm';
proc print data=class n='Number of
observations for the sex and age '
     style(N)={foreground=black
               font_weight=bold}
     style(HEADER)={background=mediumred
                    foreground=black}
/* TOTAL makes the whole TOTAL line red */
     style(TOTAL)={background=white
                   foreground=mediumred}
     style(BYSUMLABEL)={background=mediumred
                        foreground=white
                       font_weight=bold};
var name height weight ;
sum height weight;
by sex age ;
id sex age / style(HEADER DATA) =
     {background=mediumred
      foreground=black};
pageby sex;
run;
ods html close;
```

PROC PRINT STYLE EXAMPLE 4

This last example shows how to do traffic lighting with Proc Print. This capability is possible now through the use of the style statement and user-defined formats. Once the formats are defined to bind a particular style (in this case a background) to a range of values, then the format is used as the attribute's value.

In this example, we have four user-defined formats:

- Revfmt – format for revenue numbers
- Expfmt -- format for expense numbers
- Revfly – format for revenue flyover (popup msg when hovering over the cell)
- Expfly – format for the revenue flyover

Their color schemes are just the opposite from each other: the lower revenue numbers are shaded red to point out the fact that the revenues were not so good, but the lower expense numbers are indicated as green.

```
data exprev;
   input Region $ State $ Month monyy5.
         Expenses Revenues;
   format month monyy5.;
   datalines;
Southern GA JAN95 2000  8000
Southern GA FEB95 1200  6000
Southern FL FEB95 8500 11000
Northern NY FEB95 3000  4000
Northern NY MAR95 6000  5000
Southern FL MAR95 9800 13500
Northern MA MAR95 1500  1000
;

options nodate pageno=1 linesize=70
    pagesize=60 nobyline;
proc sort data=exprev;
   by region;
run;

proc format;
 value revfmt
     low-5000   = 'light red'
      5000-10000 = 'yellow'
      other      = 'green';
 value revfly
     low-5000   = 'Your revenues are
                   dangerously low'
     5000-10000 = 'Your revenues are all
                   right'
     other      = 'GREAT JOB! Keep up the good
                   work';
 value expfmt
     low-5000   = 'green'
     5000-8000  = 'yellow'
     other      = 'red';
 value expfly
     low-5000   = 'Great job controlling those
                   expenses'
     5000-8000  = 'You had better start
                   controlling expenses'
     other      = 'I''m bringing in the
                   comptroller';
run;

ods html file='PrintStyleEx4.htm';
proc print data=exprev noobs
   n='Number of observations for the state: '
 'Number of observations for the data set: ';
var revenues /
     style(COLUMN) = {background=revfmt.
                      flyover=revfly.};
var expenses /
     style(COLUMN) = {background=expfmt.
                      flyover=expfly.};
sum expenses revenues ;
by region;
id region;
run;

ods html close;
```

I will admit that there is a small defect in Version 8.2 with using the user-defined formats as style attribute values. If you use a format as a style attribute value, then you cannot define another format for that variable. For instance, in the example above where I use the format revfmt, you could not also have a format statement for revenues, such as:

Format revenues comma9.;

This problem will definitely be fixed at Version 9.

CONCLUSION

I hope that this paper has given you an insight into some of the new features awaiting your use in SAS Version 8.2. ODS continues to grow from input from YOU, the valued users, as we strive to make SAS an invaluable tool to help you in your jobs.

REFERENCES

SAS Institute, Inc., *The Complete Guide to the SAS® Output Delivery System, Version 8,* Cary, NC: SAS Institute, Inc., 1999. 310 pp.

Schellenberger, Brian., *Presentation-Quality Tabular Output via ODS.,* SAS Institute, Inc., 2000. http://www.sas.com/rnd/base/topics/odsprinter/qual.pdf

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Mrs. Sandy McNeill
SAS Institute, Inc.
Cary, NC 27513
Email: Sandy.McNeill@sas.com
ODS specific questions can be directed to:
ods@sas.com

SAS® on the Web: How do I get There from Here?

Carol Martell, UNC Highway Safety Research Center, Chapel Hill, NC
Ruth Marinshaw, UNC Academic Technology and Networking, Chapel Hill, NC
Eric Rodgman, UNC Highway Safety Research Center, Chapel Hill, NC

ABSTRACT

Do you ever wonder what it would take to start using those SAS® skills in a web application? Several pieces have to be in place if you want to play. This tutorial will tell you what those pieces are and how to either ask for them or implement them yourself. Your system and web server administrators may not be SAS users. Being informed will help you make friends while asking for what you need!

INTRODUCTION

SAS/IntrNet® software, one of the SAS web technology products, is a set of CGI tools. We examine two of the available tools, Application Dispatcher and htmSQL, in the Solaris environment. For each tool there must be a machine functioning as the Web Server and a machine function as the SAS Server. The Web Server and SAS Server may be on one machine or two different machines. Configuration options are plentiful. For this paper we simply choose one option for each tool and describe the installation and configuration steps. You should read the documentation available on the SAS Web site: http://www.sas.com/rnd/web/intrnet/doc.

APPLICATION DISPATCHER

The Application Dispatcher is, in a sense, exactly what its name implies. A request to run an application is dispatched, or sent, from a browser window through a broker to a SAS session and the results are returned to the browser window. Configurations tell the broker where the SAS session(s) and application libraries are. In some situations, the SAS session is not yet running and must be launched. In others, SAS is already running and session communication is via a port. These various modalities of sessions are defined as named services in the broker configuration file, and the desired service name is included in the web request. Each service is started, whether by request or in advance, by SAS code that includes specification of the program library locations.

Using the Application Dispatcher allows SAS to be the back end for an HTML-based interface that returns the results from a SAS job to the browser window. For example, you might present in the browser window an HTML page that displays the names of all variables from a SAS data set. The end user might select the variables of interest to him, then select procedures of interest, and submit those choices. This input is sent to a CGI program (broker) on a web server. The broker takes this information and passes it on to a SAS Application Server. The Application Server is running SAS; it takes the input passed from the Broker and invokes the requested SAS program. SAS output is then passed back to the broker and streamed to the user's browser window. The end user does not have to know how to use SAS; indeed, the end user does not need to know that SAS was run. What the end user sees is that he requested information, and it was displayed for him.

The Application Dispatcher has three basic component requirements: the browser; a CGI program, referred to as the broker; and a SAS Application Server. **To use the Application Dispatcher, you must have a web server available, and you must also have SAS running on a server.** For the purposes of this paper, we are using SAS 8.2 and an Apache web server, both running in a Solaris environment.

INSTALLATION AND CONFIGURATION STEPS

BRIEFLY

SAS Server:
1. Install SAS/IntrNet components from the Version 8 installation media.
2. Add a service to the /etc/services file
3. Configure and start the service, using inetcfg.pl

Web Server:
1. Install the Application Broker, contained in the CGI Tools, downloadable from the SAS Web site.
2. Add the service definition to the broker.cfg file

IN-DEPTH

The steps to configuring Application Dispatcher Services as a socket service follow:

1. Install SAS 8.2 on your SAS server. Be sure to install the SAS/IntrNet components with this installation.

2. Make sure that you have a web server running and available for use. You will need to know the URL for accessing files on the web server. From or with your web server administrator, determine where the web server's cgi scripts are stored. For installation of SAS/IntrNet components on the web server, the person doing the installation will need to have write permission to that cgi script directory. On many servers, this will be /cgi-bin in the directory where the web server software is installed. In addition, determine what the document root is on your web server (typically, /htdocs in the directory where the web server software is installed).

3. On your SAS server, download the latest CGI tools for Release 8.2 of SAS/IntrNet software from http://www.sas.com/rnd/web/intrnet/index.html. You will be instructed to download them to a temporary directory on your SAS server.

4. Extract the files from the compressed download file and run the INSTALL script that was included in the download. As noted earlier, the user running this script must have write access to directories on the web server and the SAS server. As part of the INSTALL process, you will specify a location for sample files, the path for cgi scripts, and the URL for those. The INSTALL process puts the Application Broker and broker configuration files

in the cg-bin directory on the web server.

5. On the SAS server, run the **inetcfg.pl** utility in !SASROOT/utilities/bin . The script will prompt you for a location to store SAS/IntrNet service files (for example, /local/pkg/default). Select the **socket** type of service, and use the service name of **default**. For the most basic installation, only create one service to run on one server. Since you are setting up a socket service, you will be prompted to identify a particular port to use. After consulting with your system administrator, identify a port number.

6. Ask your system administrator to create an entry in /etc/services that identifies the port number you referenced in inetcfg.pl and also names the service (as above, this name is **default**). For example, if you specified port 5228 for service default, a line such as the following would be added to /etc/services on your SAS server:

```
default     5228/tcp        #SAS socket srvc
```

7. After running **inetcfg.pl** and modifying /etc/services, your web server administrator should modify the **broker.cfg** file in the /cgi-bin directory on the web server. Several different parameters in this file will need to be changed. First, you will need to modify the socket service entry to reflect the appropriate parameters for your system. For example, your entry might look like this if you specified to use Port 5228 on the SAS server named myserver.unc.edu:

```
SocketService default "Reuse existing
session"
ServiceDescription "Pages reference this
generic server when they don't care which
service is used."
       ServiceAdmin "Your Name"
       ServiceAdminMail your_name@wherever
       Server myserver.unc.edu
       Port 5228
      #  Remove the following line for any
servers before V8.1
      FullDuplex   True
```

Also modify **broker.cfg** to change the SelfURL entry to the URL for your web server and broker. For example, if I used webserver mywebserver.unc.edu and my cgi scripts are stored in the cgi-bin directory, my entry would be

```
SelfURL http://mywebserver.unc.edu/cgi-bin/broker
```

You can also change the Administrator and AdministratorMail entries in **broker.cfg** to identify the person responsible for the service.
You should also modify the entry DefaultService to show the service you defined (for us, default):

```
DefaultService  default
```

Many other settings are configurable in this **broker.cfg** file. You and your administrators should read the document http://www.sas.com/rnd/web/intrnet/doc/dispatch.pdf to discuss what other options to modify. But the options shown above are the most basic.

Start your socket service by running the **start.pl** script on the SAS server. This script was created when you ran inetcfg.pl . The **start.pl** script will be in the directory that you identified in inetcfg.pl as the root directory for SAS/IntrNet services. (In my example above, I used , /local/pkg/default for my root location, so I would find **start.pl** in /local/pkg/default.)
Test whether your service is available by pointing your browser to http://yourwebserver/cgi-bin/broker? (Note that the question mark is part of the URL!!)

8. At this point, you are not finished with your configuration. In that same root directory for SAS/IntrNet services, inetcfg.pl created a file, **appstart.sas**. You now need to configure that file to include the librefs and associated paths for programs that will be run through the Application Dispatcher. The way the Application Dispatcher works is that the end user displays an HTML-coded page. The end-user uses check boxes, text entry fields, or other features to select information that SAS will process. The information is passed to the broker (on the web server). Coded in the information sent to the broker is a reference to the appropriate SAS program to run on the SAS server. That reference typically is *libref.program.sas*. In the **appstart.sas** file, you must have those librefs defined to SAS will know where to find the requested program. We recommend that you define program libraries here, not data libraries. It is a better practice to define data libraries with librefs in the SAS programs that run for a given request, rather than defining them in the appstart.sas file. The appstart.sas file can be modified at any time, but if you are running a socket service, you should stop and restart your service when you modify appstart.sas .

9. You will want to work with your system administrator to ensure that the start.pl script, which starts the Application Dispatcher services, is run whenever the system is rebooted. Your system administrator can automate this process. You should also discuss ways to verify that the service is running normally. SAS provides some utilities for administrative functions like this.

htmSQL

With htmSQL, the program and results are together in one file. A query selects items to display. Query execution is triggered with a browser request and results are displayed in the format dictated by the developer. Input files contain a mix of HTML and htmSQL directives. When a browser requests one of these input files, the htmSQL directives are processed, yielding results that are displayed in the browser window. Viewing the page source will not reveal the htmSQL directive code, since only the results are sent to the browser window. Access to data is through a SAS/SHARE® server.

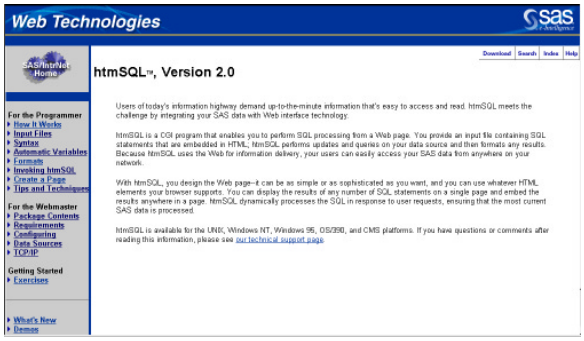
INSTALLATION AND CONFIGURATION STEPS

SAS Server installation and configuration steps:
1. Define a service on the SAS application server
2. Start a SAS/SHARE server

Web Server installation and configuration steps:
1. Install the htmSQL CGI tool on the Web server
2. Configure the installation by modifying htmSQL.cfg
2. If the SAS Server is a different machine, define a service identical to that defined on the SAS Server

Documentation for htmSQL is on the SAS Web site:

http://www.sas.com/rnd/web/intrnet/htmSQL



IN-DEPTH

THE SAS SERVER AND htmSQL

Data is accessed in htmSQL through a SAS/SHARE® server. SAS/SHARE is a sophisticated product not covered in the scope of this paper. We describe a minimal configuration in the following three steps and encourage the reader to further investigate SAS/SHARE capabilities.

1. Each SAS/SHARE server uses a defined service on the SAS application server. In the Solaris environment, a service is defined by adding a line to the /etc/services file (your UNIX administrator has 'write' permission)

```
myshare    5555/tcp    # SAS/SHARE server
```

The /etc/services line above defines a service named 'myshare' on port 5555, an arbitrarily chosen unused port. (When the Web server is a different machine, the same entry should appear in /etc/services on both machines)

2. The SAS/SHARE server references the defined service. Submit the following SAS code to start a SHARE server named 'myshare'. For our example, we choose to define a library when we start the server by preceding PROC SERVER with the LIBNAME statement.

```
LIBNAME mylib 'path to my library';
PROC SERVER ID=myshare AUTHENTICATE=OPT
COMAMID=TCP;
```

AUTHENTICATE=OPT prevents us from having to provide authentication with each htmSQL page. The COMAMID parameter sets the communication access method for the SHARE server.

3. Because the SHARE server must be running to respond to data requests, some form of automatic resubmission of the PROC SERVER is a good idea.
Suppose the PROC SERVER job is named servit.sas.
A script, scheduled to run periodically, could look for the process named servit and resubmit the job if that process did not exist:

```
#!/bin/sh
# looking for job named servit
if (ps -ef | grep servit | grep -vc grep)
then
 echo "need not run job"
 exit 1
else
 echo "must run job"
 /bin/sas /path/servit.sas
 exit 1
fi
```

If the preceding script were stored in the executable file named restartit, the crontab entry to check every minute would be:

```
* * * * * /…/restartit > /dev/null 2>&1
```

THE WEB SERVER AND htmSQL

Files to be installed on the Web server may be downloaded from the SAS Web site. All of the CGI Tools for SAS/IntrNet are included in a single tar file named websrv.tar.
The executable file, named htmSQL, belongs in the Web server directory for CGI executables. The Web server administrator will know the location.
The execution of htmSQL is configured by the file htmSQL.cfg, which belongs in cgi-bin directory. A default template named htmSQL.cfg_v8 is included in websrv.tar. Copy the template to htmSQL.cfg. Helpful documentation comments are included in the file, which can be modified with a text editor. Most options can remain at the default setting. The DATASRCFILE option points to a file of predefined data sources and associated data libraries. This data source file is not required and we do not use it for our example, since we define a libref in the PROC SERVER code. We ensure that the DATASRCFILE option is commented out in htmSQL.cfg.
It is convenient to have the Web server automatically recognize htmSQL files by the filename extension. The Web server administrator can do this by adding two lines to httpd.conf for an Apache Web server. The first line causes files with extension .hsql to be processed with htmSQL. The second line provides the path to the htmSQL executable:

```
AddHandler htmSQL .hsql
Action htmSQL /exepath/htmSQL
```

When the Web server is not the same machine as the SAS server, ask the Web server administrator to duplicate (exactly) the service definition in /etc/services.

USING htmSQL

Directives are enclosed in curly braces. Most function in pairs to delimit sections of code: {query}…{/query}. A {query} section contains one or more {sql} sections. Each {sql} section can have one associated {eachrow} section to format the results and one {norows} section for different action if the query results are empty. While the {query} section is for read access, the {update} section allows write access. Each {update} section contains one or more {sql} sections, each of which may have an associated {success} and {error} section.
Variables are preceded by & and enclosed in curly braces: {&var1}. Variables may be passed in as name-value pairs with the browser request or come into existence in {sql} sections. There are automatic variables as well.

OTHER INSTALLATION POSSIBILITIES

SAS provides you with three types of Application Dispatcher services: launch (a new SAS session is begun whenever a request is made), socket (SAS is running all the time, waiting for a request) and pool (multiple SAS servers are available). You should read about these in detail before configuring your installation. The choice of service type may be influenced by how your web and SAS services are structured. For example, a launch

service requires that the web server and SAS server be on the same machine. For a pool service, the web and SAS server may or may not be the same machine.

The most basic htmSQL setup does not include the use of a data source file. Consider the following issues to decide your requirements. Using the file can provide extra security in several ways by hiding the server name and port, the SHARE server user access password, or the userid and password for a secure mode server from the developer. The server name, port or service, and authentication elements can be placed directly into htmSQL code, sidestepping the need for the data source file. If authentication is required, it is more convenient to have a data source file and avoid having to authenticate for each query or update. Another consideration is that htmSQL referencing a predefined data source need not be updated if the mapping behind that data source changes. One would create a data source file use the dsdef script in websrv.tar.

SAMPLE CODE

We begin with a simple listing of a variable from the first 10 observations of a table.

Application Dispatcher code:

```
libname cps 'path/to/data';
data _null_ ;
set cps.countyofservice(obs=10) ;
file _webout ;
if _n_ =1 then do ;
  put '<br>' ;
  put '<br>'
   'List of first 10 CPS Service Counties'
   '<br><br>' ;
end ;
put servicecounty '<br>' ;
run ;
```

Having saved this code in a file named ex1.sas in the program library 'mylib', we can invoke this from a browser window by entering (substituting the web server name and path for …):

```
http://…/broker8?_program=mylib.ex1.sas
```

This could be placed on a web page as a hyperlink:

```
<a href=
http://…/broker8?_program=mylib.ex1.sas>
run the job</a>
```

It could be the action for a web form:

```
<form action=http://…/broker8 method="post">
<input type="hidden" name="_program"
    value="mylib.ex1.sas">
```

The results come to the browser window and look like this:

List of first 10 CPS Service Counties

Alamance
Ashe
Bertie
Brunswick
Buncombe
Cabarrus
Caldwell
Carteret
Caswell
Chowan

The htmSQL code to do the same thing:

```
List of first 10 CPS Service Counties<br><br>

{query server="myshareserver"}

{sql}
   select servicecounty
   from mylib.countyofservice(obs=10)
{/sql}

{eachrow}
   {&servicecounty} <br>
{/eachrow}

{/query}
```

This code, placed in a file named 'ex1.hsql' in a web server directory and displayed in a browser window, yields identical results:

List of first 10 CPS Service Counties

Alamance
Ashe
Bertie
Brunswick
Buncombe
Cabarrus
Caldwell
Carteret
Caswell
Chowan

One obvious advantage to htmSQL is that put statements do not come into play. Suppose your results needed to have a hyperlink to a file and the name of that file is found in a variable named var1. Also suppose that you want the visible text for the link to be var1. In the Application Dispatcher coding put statements can get complicated:

```
Put
'<a href="pathtodirectory/' var1 '">' var1
'</a>';
```

The equivalent code in htmSQL would be:

```
<a href="pathtodirectory/{&var1}">{&var1}</a>
```

SAS procedure output can be dynamically produced using the Application Dispatcher but not htmSQL.

DATA step coding can be used in the Application Dispatcher, but only SQL queries may be used in htmSQL.

ODS HTML can be used with the Application Dispatcher but not with htmSQL.

These sorts of differences will come under consideration as you plan your web applications.

CONCLUSION

Given a programming goal, SAS provides numerous pathways to that goal. Environmental circumstances can bring your goal closer or make it a little harder to reach. We have discussed installation, configuration and use of two SAS/IntrNet CGI tools. Your programming skills and your web and SAS server environment can influence the product mix you choose for your web application.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors at:

Carol Martell
UNC Highway Safety Research Center
730 Airport Rd, CB# 3430
Chapel Hill NC 27599-3430
Work Phone: 919-962-8713
Fax: 919-962-8710
Email: carol_martell@unc.edu
Web: www.hsrc.unc.edu

Ruth Marinshaw
Applications Support Group
Academic Technology & Networks
38 Phillips Hall, CB #3455
Chapel Hill NC 27599-3455
Work Phone: 919-962-4314
Fax: 919-962-5664
Email: ruth@unc.edu
Web: http://help.unc.edu

Eric Rodgman
UNC Highway Safety Research Center
730 Airport Rd, CB# 3430
Chapel Hill NC 27599-3430
Work Phone: 919-962-8709
Fax: 919-962-8710
Email: eric_rodgman@unc.edu
Web: http://www.hsrc.unc.edu

XML and SAS®: An Advanced Tutorial

Greg Barnes Nelson, STATPROBE Technologies, Cary, NC

ABSTRACT

One of the goals for SAS applications developers has been to develop three-tier and n-tier applications where the application logic (business rules) is separate from the data, which, in turn, is isolated from the user interface. In a previous paper (Barnes Nelson, 1999)[1] we discussed how to implement this logic separation using the SAS Component Language. This paper extends that line of thinking by introducing SAS developers to XML**. eXtensible Markup Language**, or XML, is a protocol of sorts that can be described as a technique for separating data from its presentation. In this paper, we will discuss XML in the context of SAS applications and how it can be used in the preparation and presentation of data. We will explore some of the features of XML that makes it a good partner for SAS-based applications.

INTRODUCTION TO XML

XML has been referred as the next ASCII; kind of like HTML; a boon to corporate information exchange and technologies' next savior. This paper is an attempt to look beyond the hype and begin to understand the ideas behind XML, its purpose and implications for technology and business improvements. There have been 100's of articles written about its benefits, and yes we'll cover some of those here as well, however the focus of this paper is to discuss the role of XML in SAS applications. Specifically, we will discuss XML and related TLAs♣; the benefits of XML; common tasks that can benefit from an XML architecture; and wrap it all within the context of SAS. By the time we are finished here, you should benefit from the practical uses of XML and the SAS code that we used to produce them.

DEFINITIONS

As people are introduced to new technology, it is often confusing when terms are introduced before they are defined. Yet, when definitions come before they are used in context, the full benefit of the term is not realized. Throughout this paper, we will do our best to define terms within the context that they are used. We will also provide an annotated glossary at the end which, not only explains the term or phrase, but identifies its context and implications, if any. If you see a term in **bold** text, you can rest assured that we will provide greater depth to its meaning in the glossary at the end of this paper.

An obvious place to start in a paper about XML is with XML itself. XML stands for **eXtensible Markup Language**. XML is a language that is used to create other "languages". Many people have taken XML, for example, and created a standard "language" to describe an industry vocabulary (see for example, BizTalk.com and HRMML).

RELATIONSHIP TO SGML

XML has its roots in a more complicated **meta-language** called **SGML**, or the Standard Generalized Markup Language. SGML was formally approved as a standard in 1986 but had been in use at IBM for several years in the production of their technical publications. SGML is the father of another widely used standard that propelled the Internet into widespread adoption: **HTML**. SGML is a language that publishers, technical writers and library automation personnel have been using to create "documents" such as museum catalogs, technical publications and product catalogs from manufacturing specifications.

SGML is, by definition, a markup language. **Markup** refers to the additional information that is added to the text of a document to enrich either its meaning or presentation. In word processing packages, for example, markup is used to control the way information is presented (or how it looks when viewed or printed.) But we don't always have to use markup to control the way the document is presented. In the case of XML, markup is used to describe the actual contents – that is, to enrich the data by describing its use, context or definition.

RELATIONSHIP TO HTML

If we remember back to our first HTML lesson, you will recall that if you wanted to display the words "Hello World" in a browser, you had to markup the contents of the HTML with special tags.

```
<HTML>
   <HEAD>
      <TITLE>My First HTML Document</TITLE>
   </HEAD>
   <BODY>
      <P>Hello World</P>
   </BODY>
</HTML>
```

We needed that much HTML simply display the text "Hello World". We soon realized that by adding additional **tags**, we could change the characteristics of the text – making it bigger, bolded, blinking, and blue. The challenge that faced content providers soon became – "how do we write this simple passage without having to know the nuances of the markup language"? – the solution: the HTML editor. Not yet satisfied, we wanted it more interactive – the solution: JavaScript. Ah yes, now we need someone else to write the code – the solution: SAS HTML formatting macros. Faster, better, easier! – the solution: SAS/IntrNet (Application Dispatcher & htmSQL) and Java Server Pages.

HTML IS OLD, XML IS COOL!

HTML has and will continue to serve a very important role in delivering information across the globe. HTML won't go away anytime soon, just like printed books will not disappear. We accept HTML for the gifts that it has brought us – displaying static information on our web browsers. But what about my cell phone or PalmPilot™ - my webTV or web-enabled refrigerator? What about getting real data like stock quotes, movie times and SAS output! XML was designed from the ground up to help us describe, transfer and deliver data. Similar to its cousin, HTML, XML is a markup language. XML's markup doesn't tell us how the data should be presented, rather it tells us what it is, how it can be used – its role is to describe the content, rather than how to display it.

The Problem with HTML

HTML was designed primarily for delivering information over the web. It is, at its roots, a language used to describe what information will look like when rendered through a web browser. Both the content and formatting of the information is tied together in the HTML language tags.

In HTML, we have a finite number of tags that we can use to markup our documents. These tags are used for controlling how the document should be presented. For example, in our previous example, we could have made the text "Hello World" bolded by added the <B> tag.

```
<P><B>Hello World</B></P>
```

In XML we are not limited by the number of tags that someone else has thought of to describe our data – that's our job! We use the tags as we see fit to describe the content and context of the data. For example, the following is XML document describes the first observations from some data that we may have about our customers.

---

[1] Barnes-Nelson, G.S. (April, 1999) Extending the Life of Your AF Application: Exploiting the Model-Viewer Paradigm. Invited paper at the annual convention of the International SAS Users Group (SUGI), Miami Beach, FL.

♣ Three letter acronyms ✪

```
<?xml version="1.0" ?>
<customer-data>
<contact-information>
  <cust-id>137000</cust-id>
  <name>Kraft, Ms. Rose</name>
  <gender>Female</gender>
  <age>34</age>
  <income>32,340</income>
  <status>Married</status>
   <address>
        <street ORDER="1">869
        Veterans Blvd.</street>
        <street
        ORDER="2">Business
        Research</street>
        <city>Rutherford</city>
        <state>NJ</state>
        <zip>70702</zip>
     < region
        >NORTHEAST</region>
  </address>
 </contact-information>
 </customer-data>
```

XML Document 1. XML document produced with a SAS DATA STEP.

This XML document was produced dynamically from a SAS DATA Step[2]. But notice that instead of the usual tags like <B>, <P>, <A HREF> and so on, we have custom tags that we have used to describe our data. The first line tells us that we are dealing with an XML document. The second line has a tag called <customer-data>, which tells us that we are dealing with our customers. Within this, we find that we have some customer information, designated with the <customer-information> tag. Within each customer, we have collected a variety of information – for example, their name <name>, gender <gender>, income <income>, marital status <status>and another section that contains their address <address>.

Although not a terribly complicated example, we have exposed one of the key benefits of XML: simplicity. This document represents a hierarchy of information with easily understood patterns. Because of this simplicity, both humans and the computers can access it, understand it, and translate it into useful information.

In our example, the hierarchy dives only 3 levels deep: Customer Data, one or more customers (customer-information) and one more level for address information. There is no reason that an XML document could not represent a complex hierarchy with multiple, nested levels of information. We will explore a more complicated example later in this paper where we pull information from multiple tables to create a complete customer history profile.

INFORMATION TECHNOLOGY CHALLENGES

Despite incredible advances in technology, there are some persistent challenges that face us as technologists trying to solve real-world business problems.

XML will likely not be the technology that saves us from painstaking processes to make our data cleaner, more accessible or provide a richer context for information and its delivery across the web or across the room. However, if applied appropriately, XML can help solve some common barriers to productivity.

Multiple Views of the Same Data

A common challenge faced by many organizations requires data to be formatted differently depending upon its use. For example, account history or a customer's profile may be generated for the Customer Service department in order to interact with customers on the phone. The Finance department, however, may require a different view of the data in order to invoice the customer. The Sales and Marketing departments need yet a different view of the customer – requiring both granular data for each customer as well as highly summarized data for database marketing (buying history and cross-selling campaigns). Each department requires a similar, but different view of the data. In addition, data for each of these applications may be housed in different systems and defined somewhat differently.

XML can help us by providing a framework for understanding the customer in terms of a patterned hierarchy -- essentially giving us a common definition of a "customer". By defining a standard such as this, departments can exchange information about a customer easily and quickly -- regardless of where and how the data was stored. Additions or modifications to the source data have little impact on the XML document if care is taken to retain the way that the customer is defined according to the XML document. Because the data is separated from the way that it is presented, any changes in process or business rules, wont cause the systems to break -- especially when those systems cross-organizational boundaries. In addition, new views or representations of the data that are required can be generated without changing the underlying XML data.

We will explore later different methods to render or display XML data, but one of the benefits of XML is that once data has been delivered to the client application, it can be manipulated, transformed and presented in a variety of ways – all without having to request the XML document from the server a second or third time.

Application Integration

Implicit in the first point above is the idea that data can be integrated from disparate sources and/ or multiple applications. In our case, for example, we may have data that is housed in one or more source systems (e.g., ERP systems, billing systems, Sales-Force Automation, database marketing/ data mining databases). Despite this "separateness", data from these diverse sources and/ or applications can be brought together using a common meta-language that defines our customer. In our fictitious customer application, we have various applications connected over a network. When one application wants to access information from another application, an XML-formatted document is sent across the wire to the requesting application.

Because the definition of customer has been defined as a standard, information about the customer can be exchanged among companies much more easily as the mechanism for data interchange doesn't rely on, nor expose, the internal business systems. Data can be sent, for example, to a partner with only parts of the "customer" that they care to have the partner see. Invoicing might be shared with your billing supplier or the customer directly through the web, e-mail or other electronic means (PDAs, Cell Phones, etc.).

Information Optimization

Another business challenge that is often faced is the assimilation of huge amounts of unstructured data into meaningful context. As humans, we can process data very efficiently when the data doesn't appear to have a pattern to it. In our customer application, we may want to include all sorts of information about our customers from diverse data sources outside of our firewalls. Take for instance the following scenario:

As we cruise the web, we find an article about a new product that a potential customer is developing. Our company creates products and services that would be a good fit for this new potential client. As we read this article, we can parse the text, assimilate its meaning and make judgments about what we have read. Next, we decide to get an independent assessment of their company – we request information from Dunn & Bradstreet that will show us how they have done financially, who holds senior management positions, where they have offices, etc.

A trip to their web site affords us an opportunity to get another perspective: to get a sense of their culture – from the words they use to describe their company to the opinions they have about their own products. Finally, we incorporate some of what we have learned about this company into our own Sales Forces Automation system so that our sales representatives will be more knowledgeable as they interact with this new potential customer.

XML to the Rescue

Much of what we have described above can be best characterized

[2] See Appendix A for an example of producing XML using a SAS Data Step.

as unstructured data. A contemporary solution to this problem would be a Knowledge Management system. A non-technology solution would be to create new positions whose job would be to "surf and assimilate". But XML does offer some key advantages to this problem.

Smart Agents. Suppose instead of you sitting behind the terminal searching for documents, you tell the computer the kinds of things you were interested in and have a computer do the work for you – filtering, cataloging and storing the retrieved information. The XML paradigm keeps the information separate from the presentation rules, allowing for intelligent agents to scan through a document's content and ignore the style sheet if one is present.

Meaningful Searches. HTML-based search tools use keywords and text to manage the information about the millions of web sites in existence. XML-based search tools, however, use the inherent data structure in the XML documents themselves as well as the meta-data contained both in the XML document as well as the **Document Type Definition** (DTD). Given the amount of information on the web, technology that gives us more precise control over searches should help sift through information more cleanly than before. If we were to conduct a search on the web today – say on SAS – we would get between 268,000 and over a million hits depending on the HTML-based search engine we used. The topics returned range from SAS Institute to Scandinavian Airlines to Surfers Against Sewage. Because XML provides a context for our searching, we could specify SAS in the context of <company>, <software> or other relevant elements.

Granular Updates. Since the structure of an XML document is a known hierarchy of information, we are able to update information by sending only parts of the document each time there is a change. By using this feature of XML, we don't have to resend the entire hierarchy to the requesting application.

Technology Optimization

At the time of the writing of this paper, there are literally billions of pages of information available on the web on over millions of web sites globally. Most of the web pages are written using HTML. As we have learned earlier, HTML is a great tool for constructing documents to be displayed over the web. As you surf the web, you request a document from a web server using a Uniform Resource Locator (URL). Assuming the document can be located on the specific server, the page is downloaded to your browser for rendering. Once downloaded, the communication between your machine and the web server is essentially broken. That is, the relationship between the browser and the server is connectionless.

Continuing with our customer scenario, let's assume that we have downloaded a table of information that lists all of our customers, what products they have purchased, how much they have spent with us, where their home office is, who their sales rep is and so on. If we wanted to sort the information by any of the data that we have in the table, we would have to send a request back to the web server to have it re-processed and re-rendered. This approach places a tremendous burden on the web server to handle fairly simple requests. A more efficient approach would be to have the client machine handle the local manipulation of the data where it could be sliced-and-diced, sorted, filtered and rendered differently.

By combining XML and XSL (**eXtensible Stylesheet Language**), we can achieve this goal of local computation and manipulation (more on XLS later.) Once an XML document is downloaded, we can achieve this level of interactivity on the client side with a single request from the web server. Figure 1 shows an example of an interactive document that is built entirely of XML data delivered to the client by a single request to the server. For more information on this example, refer to the article entitle "Transform Your Data with XSL" found at

http://www.xml-zone.com/articles.asp

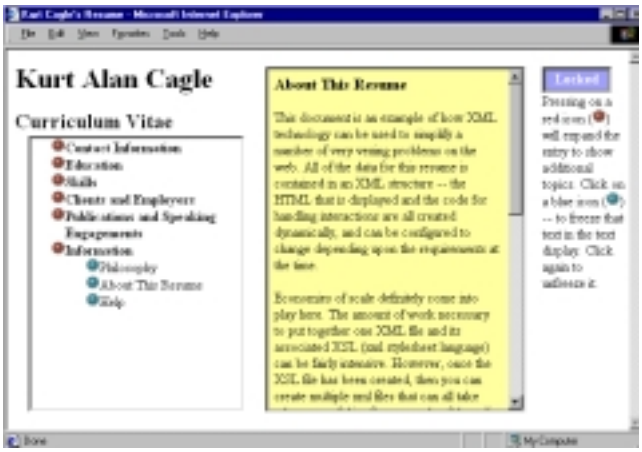


Figure 1. Dynamic XML application.

In addition to the flexibility gained through local processing and manipulation, we can realize efficiency gains by off-loading the requests from the server. As the above example shows, data can be translated from this interactive version into a print version or XML only version so others who can process this XML can incorporate the data into their applications. In fact, there is a movement to unify the format of resumes so that human resources information can be shared universally (see the Human Resources Management Markup Language in the glossary for more information.)

UNDERSTANDING THE XML LANDSCAPE

So what? We have an XML document that is self-describing. We still can't print invoices, e-mail with it or build applications with it, right? What we have discovered thus far in our exploration of XML is the simple case of the XML document. The beauty of an XML document is that it can be *written*, *read* and *rendered* by countless applications. These three, oversimplified, tasks help define some of the technologies that surround XML. Before we discuss these three tasks and how to perform these tasks with the SAS System, it is important to understand the rules that govern XML documents.

Rules to Live By

Unlike HTML, XML has a rigorous set of rules that govern how a document should be constructed. A well-formed XML document is one that has all of the characteristics that it is supposed to have. There are a few simple rules about how an XML document should be constructed.

- ✓ Beginning and ending tags must match. That is, you cannot have an <ADDRESS> tag without one that ends the expression </ADRESS>.
- ✓ Elements can be nested within each other, but they cannot cross boundaries. For example, we can have

```
<Customer>
    <ID>109</ID>
    <Name>Greg Barnes Nelson</Name>
</Customer>
```

but not:

```
<Customer>
    <ID>109
    <Name>Greg Barnes Nelson</ID>
    </Name>
</Customer>
```

- ✓ XML tags are case sensitive. Each of the following, for example, is considered a different element.

```
<Customer> <CUSTOMER> <customer>
```

✓ Because each element must have a beginning and ending tag, empty elements are signaled by either a closing tag or a />. These two lines are equivalent:

```
<Customer/>
<Customer></Customer>
```

✓ Just like in HTML, there are some reserved characters that cannot be used. These include:

<	<	&	&amp;
>	>	"	&quot;
'	&apos;		

Each XML document must have a root element that denotes the top of the hierarchy or root. In our example <customer-data> represents the root element and cannot occur anywhere else in the document.

Assuming an XML document follows all of these rules, browsers or other parsers that can read XML are guaranteed to be able to read your document. This is one of the primary differences from HTML. Because all of these rules are adhered to, our XML document is said to be well-formed.

In addition to being well-formed, documents that have something called a **Document Type Definition** (DTD) and are well-formed are **valid**. Although it is not necessary that a document be valid (i.e., have a DTD), is it often useful to document them in a DTD so that other people or applications can benefit from knowing how they are supposed to be used.

A DTD essentially describes the document's rules – that is, which elements are present and the relationship among the elements. DTDs, although optional, help to validate the incoming XML document when the application doesn't have a built-in definition of the XML document.

Writing XML

As SAS applications developers, one of our primary tasks in the near future will be to learn how to create an XML document. Just as we have learned how to write HTML from our SAS applications, XML documents can be created programmatically using the SAS language.

In SAS Version 8.0 (M01), there exist several lightly documented methods for writing XML natively. We can write XML from a Data Step (see Appendices A and B for examples), from output that we generate using SAS' Output Delivery System (ODS) or using the Version 8 XML libname engine. Let's discuss each of these with an example of each.

SAS DATA STEP

In addition to providing a very powerful engine for creating complex documents, the DATA STEP will be perhaps the most familiar method to most SAS programmers. As depicted in Appendix B, we can programmatically create an XML document from a SAS data set. Despite its simplicity, we can extend this example to combine a much richer XML document by combining multiple data sets to create the patterned, hierarchical output that characterizes the XML document. In Appendix B, we find a more complicated example of this as we pull in data from multiple data sources.

In our simple case (see XML Document 1 shown earlier), we created a simple XML document that displayed our customer data with basic name and contact information as well as some brief demographics. But there is no reason that you couldn't extend the example to include past orders, billing and shipping address or marketing constructs such as Life-Time Value, Segmentation, etc. By using the power of the SAS Data Step, we can programmatically include additional content as well as metadata from SAS' dictionary tables and formats. The example shown below was created to show how we would create a complete customer history from multiple tables (see Appendix B.) Here we show an XML document that contains several customers, their contact information, demographics and all of their past invoices. We have collapsed the view to show only the high-level portion of the invoices. Below, we show an expanded invoice section.

```
<?xml version="1.0" ?>
<customer-data>
   <cust-info>
        <cust-id>137000</cust-id>
        <name>Kraft, Ms. Rose</name>
       <demographics>
            <gender>Female</gender>
            <age>34</age>
            <income>32,340</income>
            <status>Married</status>
        </demographics>
       <address>
            <street ORDER="1">869
                 Veterans Blvd.</street>
            <street ORDER="2">Business
                 Research</street>
            <city>Rutherford</city>
            <state>NJ</state>
            <zip-code>70702</zip-code>
            <region>NORTHEAST</region>
        </address>
       <invoices>
          <invoice INVOICE-ID=">107707"
                INVOICE-
                DATE="07MAR1994">
             .. .. .. ..
          <invoice INVOICE-ID=">135872"
                INVOICE-
                DATE="08AUG1994">
             .. .. .. ..
          <invoice INVOICE-ID=">243377"
                INVOICE-
                DATE="24APR1998">
       </invoices>
    </cust-info>
   <cust-info>
   .. .. .. ..
  </customer-data>
```

XML Document 2. Customer History XML document.

Here we show one particular invoice for August 8, 1994.

```
<invoice INVOICE-ID=">135872" INVOICE-
      DATE="08AUG1994">
<lineitems>
  <line-item ID="Item1">
       <product-code CAT="Toys">TY1200</product-code>
        <quantity>4</quantity>
   </line-item>
  <line-item ID="Item2">
       <product-code CAT="Toys">TY2100</product-code>
       <quantity>1</quantity>
   </line-item>
  <line-item ID="Item3">
       <product-code CAT="Toys">TY2300</product-code>
      <quantity>1</quantity>
   </line-item>
  <line-item ID="Item4">
      <product-code CAT="Toys">TY4100</product-code>
     <quantity>1</quantity>
   </line-item>
 </lineitems>
</invoice>
```

XML Document 3. Customer history XML document with expanded view of an invoice.

ODS

Starting with Version 7, SAS programmers were able to take full advantage of the Output Delivery System or ODS. The goal of ODS was to rules that governed what output should contain versus how it should be presented. Although experimental in Version 8.00, we can now direct any output that can be created and send it to an XML document. The default XML engine for ODS in Version 8.0 (M01) produces a proprietary XML document that

adheres to SAS Institute's Version 8.0 DTD. The code to create a sample XML document with PROC TABULATE is shown below. The XML document that is created as a result of this contains a tremendous amount of metadata, which can be used to describe the output and its potential relationship to other output objects.

```
ods xml file="c:\xmltabulate.xml";

 proc tabulate data=SASUSER.CLASS  ;
   table   ALL
     ,(sex age height weight) * n =' '
     ;
   class  sex age height weight;
 run;
ods xml close;
```

In Version 8.01, additional experimental engines will be developed that support the DocBook[3] standard as well as a few variants of HTML output (with CSS and a bare-bones HTML version). By combining SAS Institute's raw XML documents with the appropriate DTD, we have a powerful method for transforming and filtering output in other applications that can read, write and render XML documents.

XML Libname Engine

Although experimental in Version 8.0 (M01), the XML LIBNAME engine provides an easy method of writing XML documents directly from libname references. Those tasks that you can perform with a standard libname such as updating data are available through this engine. The development at SAS Institute is ongoing in this area, but we can see a simple example of taking a SAS dataset and writing it out to an XML document.

```
libname sampdata 'C:\mylib' ;
libname DestXML XML 'output.xml';

data DestXML.dsetanything ;
  set sampdata.customer
      (label="My customer information");
  addr1=urlencode(addr1);
  label custnum ="Customer-Information";
run;
```

The first row of the XML document that is produced from these statements is presented below. We had to use the urlencode function in our code to encode any special XML element names such as the ampersand found in one of our addresses (see "Rules the Live By" section above for examples of these.) When we created the XML files, we specified a data set name just like we would for a permanenet SAS data set (dsetanything). This name is used in the XML as a descriptor for the document. However, with the default engine, which is generic mode XML[4], variable labels, formats and lengths are not written to the document. We can use a different engine by using the XMLTYPE= option and specify the values values of GENERIC|ORACLE or HTML or OIMDBM[5].

```
<?xml version="1.0" ?>
<TABLE>
   <ROW>
      <CUSTNUM>137000</CUSTNUM>
      <NAME>Kraft, Ms. Rose</NAME>

         <ADDR1>869%20Veterans%20Bl
         vd.%20%20%20%20%2</ADDR
         1>
      <ADDR2>Business Research</ADDR2>
      <CITY>Rutherford</CITY>
      <STATE>NJ</STATE>
      <PHONE>201-507-2211</PHONE>
      <REGION>NORTHEAST</REGION>
      <AGE>34</AGE>
      <INCOME>32340</INCOME>
      <MARRIED>1</MARRIED>
      <SEX>0</SEX>
      <ZIP>70702</ZIP>
   </ROW>
.. .. ..
</TABLE>
```

XML Document 5. XML document Created with the XML Libname Engine (Version 8.0 M01).

Reading XML

The primary value of an XML document is that it can be easily interpreted by someone else – especially by computer. The exchange of information through XML as the medium requires that the recipient be able to read, process and possibly transform the information into something usable. There are a variety of **XML parsers** available which can read and interpret an XML document (see for example IBM, Microsoft and Sun Microsystems.)

A parser, or engine that reads an XML document is typically embedded in an application. Its job is to read the document and convert the content into constructs that the application understands. For SAS application developers, we can write our own parser using SCL or Base SAS, but it would be much simpler to use an XML parser that was written by someone else. IBM's parser is one of the most powerful and is available as a Java applet.

Which parser you use will depend on your application. A likely scenario for SAS application developers would include a back-end or middle-ware parser would interpret the document and apply programming logic to load a database, construct an HTML page or render a PDF document.

Through the use of the XML Libname engine, we demonstrate below the conversion of an XML document to a SAS dataset. Once parsed, we can then use our traditional SAS programming constructs to develop robust client/ server or web-based applications. The general form of the XML syntax for reading an XML document is given here. (Remember, this is experimental!)

```
libname myXML XML 'C:\mylib\class.xml';
proc datasets dd=myXML; run;

data readback;
  set myXML.row;
run;

Proc print data=readback;
run;
```

Rendering XML

Although XML was designed primarily as a way to solve the problem of exchanging web documents, it is clear that XML has potential for solving other sorts of data exchange problems that are not limited to the web. In Version 8.1 of the SAS System, we will see support for WAP (**Wireless Application Protocol**) that will enable XML documents to be sent over wireless protocols to such devices as PDAs and cellular telephones.

Despite its youth, XML already has a rich set of tools, which allow us to render XML documents in a variety of ways. If our browser supports the viewing of XML documents, we can open them directly. IE 5 is currently the only browser that supports XML natively. Because of this fact, rendering HTML on the server and pushing it to the client will probably be the most common method of rendering XML until there is a larger base of browser support for XML.

Other types of transformations include:

- ✓ Converting XML into Scalable Vector Graphics (SVG) based on the W3C's SVG markup language to produce pie charts and other graphical formats from XML documents.
- ✓ Rendering XML into PDF documents using James Tauber's FOP (Formatting Objects into PDF).
- ✓ Converting XML documents into TeX files which can be

---

[3] For more information about DocBook, see http://www.oasis-open.org/docbook/

[4] This is compatible with the Oracle8i XML implementation.

[5] The OIMDBM is an industry standardization and attempts to express formatting information in its output. see http://www.mdcinfo.com/OIM/OIM10.html

rendered on a variety of platforms.

- ✓ Converting XML into speech (VoiceXML) which can be used create audio tracks by a variety of software tools.
- ✓ Rendering XML into an HTML document.

Microsoft Internet Explorer 5.0

Because we are most familiar with the visual presentation of the web, we will explore rendering XML documents in a web browser using Microsoft's Internet Explorer (IE5). This first example shows the use of Microsoft's XML Data Source Object (DSO), which is a Java applet that can be used to bind an XML data source to a web page (in IE5).

The HTML used to produce this page is fairly straightforward. The key to this page is the APPLET reference where we pass it a parameter pointing it to the proper XML data source. This example shows XML's simplicity – we have not done anything else to this document to make it appear in the table except what you see below. The XML document referenced here is the same one that we showed in XML Document 1 above.

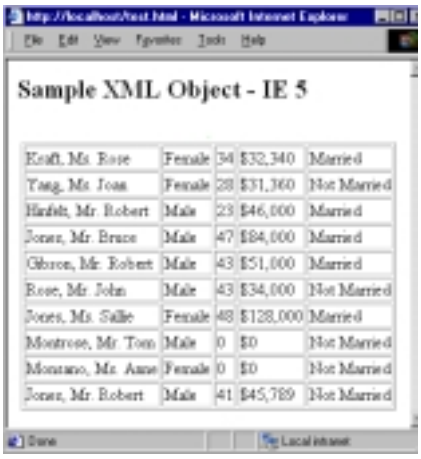


Figure 2. XML Document rendered through Microsoft's DSO.

```
<html><head></head><body>
<h2> Sample XML Object - IE 5</h2>
 <center>
<APPLET code=com.ms.xml.dso.XMLDSO.class id=Customer
width=0 height=0 MAYSCRIPT="true">
<PARAM NAME="url" VALUE="http:///localhost/customer.xml">
</APPLET>
<table border=1 datasrc=#Customer>
  <tr>     <td><span datafld="name"></span></td>
           <td><span datafld="gender"></span></td>
           <td><span datafld="age"></span></td>
           <td>$<span datafld="income"></span></td>
           <td><span datafld="status"></span></td>
  </tr></table></center></body></html>
```

HTML Segment 1. HTML code for binding and XML data source to a web page.

Expanding on this simple example where we bind our customer XML document to a web browser, we could also display it in a form view with navigations that allow us to move through the dataset (forward, backward) as well as add/ modify and delete records. For more information on the XML **Data Source Object** (DSO), please refer to the glossary at the end. Here we provide references to on-line resources.

Doing it in Style

Those familiar with HTML may also be familiar with **Cascading Style Sheets** (CSS). CSS was an early attempt at separating the information contained in HTML from how it was presented. Instead of marking up a page header with font specifications that control the size, orientation and other font characteristics, we can use Cascading Style Sheets and a class definition to control this. This allowed us to create basic HTML documents whose look and feel could be controlled by an external style sheet.

XML has a similar construct for handling the complex presentation requirements of the web. In XML, one can either use Cascading Style Sheets (CSS) or the **Extensible Stylesheet Language** (XSL) to present data in a browser. Figure 3 shows an XML document formatted with a CSS (left) and an XSL (right).

XML Document rendered with a CSS.

XML Document rendered with an XLS.

Figure 3. XML Document rendered with Cascading Style Sheets versus an Extensible Style Sheet.

Despite the fact that Cascading Style Sheets were designed for HTML, it is just as good at formatting XML documents for the web. The XML document that was used was identical to that described in XML Document 1 referenced earlier in this paper. The only difference was the addition of a header describing where we can find the CSS. The CSS can be found in Appendix D.

```
<?xml version="1.0" ?>
<?xml-stylesheet type="text/css" href="Labels.css"?>
<customer-data> and so on ... ...
```

XLS (extensible Style Sheet) is a style sheet technology designed specifically for XML. With XSL, we are able to control our document's presentation much more than before. As demonstrated here (one in Figure 3-Right and again in Figure 4), we have two XML documents that are identical except for the reference to its XSL style sheet. Note the clear separation of the content (XML document) from its presentation (through its XSL reference.)

The XML header for this document is shown here:

```
<?xml version="1.0" ?>
<?xml-stylesheet href="Labels.xsl" type="text/xsl"?>
```

This second rendering below shows the exact same document – the only difference: the XML header points us to a different XSL document.

```
<?xml version="1.0" ?>
```

<?xml-stylesheet href="Table.xsl" type="text/xsl"?>

Figure 4. XML Document rendered with Table.xls.

Transformation Through XSL and XSLT

In addition to being able to render XML documents, XML provides us with a rich toolset for transforming documents as the style sheet is applied. Through the use of both XSL and XSTL (**XML Transformation Language**) we can move text from one place in an XML document to another (for example, moving the value of first name and last name around); sorting elements (see, for example Figures 3 and 4 – note the order of the names in each); generating text and performing calculations (such as the count of all of the line items in an invoice); and numbering items in a list. Both XSL and XSTL provide a rich set of tools for manipulating and managing the information contained within an XML document.

CONCLUSION

XML brings a tremendous amount of power and flexibility to both client/ server and web-based applications. As we have seen in this paper, there are a number of compelling benefits to both developers and to the organizations they serve. For the SAS developer, XML offers a rich toolset for communication across application and organizational boundaries. We are able to structure data in the context of meaningful hierarchy in a way that other people and software can easily understand. Because of its patterned structure, data can be shared literally with any application or user that requires it. As data and its corresponding structure changes, the XML document and its structure change with it.

THE FUTURE OF XML AND SAS

Although no production SAS Institute applications have been delivered to date using XML, we can easily incorporate XML into our applications by using common programming elements such as SCL and the DATA Step or combining other technologies in concert with the SAS System.

As more vendors, like SAS Institute, incorporate XML into both applications and lower level language support, we should expect to see a wide variety of uses for XML in our applications. In the short term, we can expect to see more experimental engines in BASE SAS with ODS and the LIBNAME engines in Version 8.1.

Potential Applications

Beyond simple tasks like reading and writing, we would hope to see a variety of application level support for XML. Here are just a few ideas.

Native SAS XML Tree-Viewer. Having a built-in editor for viewing/ editing XML documents that have been created within SAS. For example, one could pull up an XML document from the File -> Open Menu to display a tree-view for editing/ viewing the XML structure.

SAS/ IntrNet Extensions. For web-based applications, native support for both parsing XML and writing XML to the Application Dispatcher sessions for use in subsequent pages would be a logical extension of this technology. The XSL language is common in many regards to what we see in htmSQL as it handles a records sets in the {eachrow} directive. I would expect native htmSQL to be able to navigate an XML document hierarchy.

Messaging Transport. With respect to messaging, we should be able to expect will provide a transport mechanism (both receiving and sending) XML-based messages between both SAS and non-SAS based clients and servers.

Balanced Scorecard. For applications, like Balanced Scorecard, we could use XML to deliver metrics from throughout the organization in a single, common format.

In general, XML can be used wherever data is being exchanged between multiple applications and/or organizations to facilitate the understanding and assimilation of the data in its new context.

By allowing SAS developers to fully exploit the benefits of XML, the SAS System will continue its leadership role in the integration, analysis, presentation and decision support for years to come.

ACKNOWLEDGMENTS

The author would like to sincerely thank several people for their rule in gentle and thoughtful review of this manuscript. Specifically, we would like to thank Ian Whitlock, Don Henderson, Chris Olinger and Anthony Friebel for their support.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Please feel free to contact the author at:

```
<?xml version="1.0" ?>
<business-card>
    <author>
        <name>
            <first-name>Greg</first-name>
            <middle-initial>S.</middle-initial>
            <last-name>Barnes Nelson</last-name>
         </name>
         <title>Director</title>
        <company>
            <comp-name>STATPROBE Technologies</comp-name>
            <web-site>www.statprobetechnologies.com</web-site>
           <address>
                <street>117 Edinburgh South, Suite 202</street>
                <city>Cary</city>
                <state>NC</state>
                <zip-code>27511</zip-code>
            </address>
         </company>
        <contact-methods>
            <business-phone>919-465-0322 x351</business-phone>
            <business-fax>919.465.0323</business-fax>
            <b-email>greg.barnesnelson@statprobe.com</b-email>
            <personal-email>gregbn@ix.netcom.com</personal-email>
         </contact-methods>
     </author>
 </business-card>
```

...or simply email me at greg.barnesnelson@statprobe.com.

APPENDICES

Appendix A. Producing a simple XML document from a SAS Data Set

Simplecustomerview.sas .....

```
filename outxml "c:\simplecustomer.xml";

Data _null_;
  file outxml;
  set sampdata.cust10 NOBS=Lst;
  length gender $6. marital_status $11.;

  %let tab="     ";
  addr1=htmlencode(addr1);
  addr2=htmlencode(addr2);

          if sex=0 then gender='Female';
```

```
                else gender='Male';

          if married=0 then marital_status="Not Married";
                       else marital_status="Married";

          if _n_=1 then do;
                    put '<?xml version="1.0" ?>';
                    put '<customer-data>';
          end;

          put '<contact-information>';

           put &tab'<cust-id>' custnum '</cust-id>';
           put &tab'<name>'    name '</name>';
           put &tab'<gender>' gender '</gender>';
           put &tab'<age>'     age '</age>';
           put &tab'<income>' income '</income>';
           put &tab'<status>' marital_status '</status>';
           put &tab &tab '<address>';
           put &tab &tab '<street ORDER="1">'
               addr1 '</street>';
           put &tab &tab '<street ORDER="2">'
                addr2 '</street>';
           put &tab &tab '<city>' city '</city>';
           put &tab &tab '<state>' state '</state>';
           put &tab &tab '<zip-code>' zip '</zip-code>';
           put &tab &tab '<region>' region '</region>';
           put &tab &tab '</address>';

          put '</contact-information>';

          if _n_ = lst then do;
                    put '</customer-data>';
          end;
  run;
```

Appendix B. Producing a simple XML document from Multiple Data sets

CustomerInvoiceView.sas

Because of the length of this program, the code for this example can be found on-line at http://www.statprobetechnologies.com/XML

Appendix C. Cascading Style Sheet used to format an XML document

Labels.css

```
contact-information {
  display: block;   width: 350px;   padding: 10px;   margin-bottom: 10px;
border: 4px double black;   background-color: white;   color: black;  text-
align: center;}

name {
  display: block;  font-family: Times, serif;  font-size: 16pt;
  font-weight: bold;}

street {
  display: block;  font-family: Times, serif;  font-size: 14pt;}
city, state, zip-code {
  display: inline;  font-family: Times, serif;  font-size: 14pt;}

cust-id, gender, age, income, status {
  display: none;}
```

Appendix D. eXtensible Style Sheet used to format an XML document as Mailing labels

Labels.xls

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
  <xsl:template match="/">
    <html><head><title>Customer Address Book XML
Example</title></head>
      <body bgcolor="#FFFFFF">
    <xsl:for-each order-by="+ name" select="customer-data/contact-
information">
        <xsl:apply-templates select="name"/>
        <xsl:for-each select="address">
           <xsl:apply-templates select="street"/>
           <xsl:apply-templates select="city"/>
           <xsl:apply-templates select="state"/>
           <xsl:apply-templates select="zip-code"/>
        </xsl:for-each>
      </xsl:for-each>
    </body>
   </html>
  </xsl:template>

  <xsl:template match="name"><h2><xsl:value-of/></h2>
  </xsl:template>

  <xsl:template match="street"><xsl:value-of/><br/>
  </xsl:template>

  <xsl:template match="city"><xsl:value-of/>,
  </xsl:template>

  <xsl:template match="state"><xsl:value-of/>
  </xsl:template>

  <xsl:template match="zip-code"><xsl:value-of/><br/>
  </xsl:template>

</xsl:stylesheet>
```

Appendix E. eXtensible Style Sheet used to format an XML document as a Table

Table.xls

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
  <xsl:template match="/">
    <html><head><title>Customer Address Table Example</title></head>
      <body bgcolor="#FFFFFF">
<h1> <font color="blue">Sorted in decending order</font> </h1><p/>
<table align="center" border="1">
            <tr>
              <th>Name</th>
              <th>Address</th>
              <th>City</th>
              <th>State</th>
              <th>Zip</th>
            </tr>
<xsl:for-each order-by="- name" select="customer-data/contact-
information">
            <tr>
        <td><xsl:apply-templates select="name"/></td>
          <xsl:for-each select="address">
             <td><xsl:apply-templates select="street"/></td>
              <td><xsl:apply-templates select="city"/></td>
              <td><xsl:apply-templates select="state"/></td>
              <td><xsl:apply-templates select="zip-code"/></td>
          </xsl:for-each>
            </tr>
      </xsl:for-each>

            </table>
    </body>
   </html>
  </xsl:template>

  <xsl:template match="name"><h3><xsl:value-of/></h3>
</xsl:template>

  <xsl:template match="street"><xsl:value-of/><br/>
  </xsl:template>

  <xsl:template match="city"><xsl:value-of/>,
  </xsl:template>

  <xsl:template match="state"><xsl:value-of/>
  </xsl:template>

  <xsl:template match="zip-code"><xsl:value-of/><br/>
  </xsl:template>
</xsl:stylesheet>
```

APPENDIX B. ANNOTATED GLOSSARY AND BIBLIOGRAPHY

This glossary provides some of the most common terms and phrases used in the context of creating, processing and rendering XML documents. An up-to-date version of this table can be found at http://www.statprobetechnologies.com/XML

Term	Acronym	Description / Purpose	References
Cascading Style Sheet	CSS	CSS is a style sheet language designed as an early attempt to help developers separate the content in HTML from the way that it was presented. CSS can use tag definitions such as <H1> to apply formatting across a document or to sections that are specifically identified within a document. CSS can be used to format an HTML document as well as an XML document.	http://webreview.com/guides/style/style.html http://www.w3.org/Style/
Data Source Object	DSO	Data Source Objects (DSOs) are objects that can imbed structured data, including XML, into HTML pages. This is a proprietary technology that is embedded into Microsoft's browsers.	http://msdn.microsoft.com/workshop/Author/databind/datasources.asp
Dynamic HTML	DHTML	Dynamic HTML provides a mechanism to control client- or browser-based interactivity through programmatic access to the HTML elements. Programs are typically written in client-side scripting languages like JavaScript and VBScript	http://msdn.microsoft.com/library/backgrnd/html/msdn_dynhtml.htm
Document Object Model	DOM	The document object model is a standard objected oriented application programming interface (API) that gives developers programmatic control of XML document content, structure, formats, etc. The XML DOM is a recommendation from the W3C and provides access through scripting languages like VBScript and JavaScript.	http://www.w3c.org/TR/REC-DOM-Level-1 http://www.w3.org/DOM/
Document Type Definition	DTD	The document type definition defines the valid syntax for a class of XML documents. It provides a list of the element names, which elements can appear in combination with which other ones and so on.	http://www.whatis.com/dtd.htm
eXtensible Markup Language	XML	XML is the new language of the web. It provides a framework for delivering structured data from a wide-variety of applications to the desktop for local computation and presentation. It is an ideal format for exchanging data within and between applications and organizations.	http://www.ucc.ie/xml/faq.html http://www.oasis-open.org/cover/xml.html http://www.w3.org/XML http://msdn.microsoft.com/xml/general/xmlfaq.asp http://www.alphaworks.ibm.com http://www.xml.com/pub/98/10/guide0.html http://www.webdeveloper.com/html/html_xml_1.html
Extensible Stylesheet Language	XSL	A working draft that describes a language that can be used to provide flexible document presentation rules. Similar to CSS, it applies formatting rules to a document that is separate from the XML document itself. Written in XML, XSL contains instructions for getting data out of a document and converting it into another.	http://www.xmlmag.com/upload/free/features/xml/1999/01win99/kc2win99.asp http://www-4.ibm.com/software/developer/education/transforming-xml http://www.w3.org/Style/
Human Resources Management Markup Language	HRMML	HRMML, created by Structure Methods, Inc., is an application whose aim is to unify the manner in which human resource information is represented and shared.	http://www.structuredmethods.com/hrxml
Hyper Text Markup Language	HTML	The language of the web. HTML is used to format documents so that they can be viewed over the web through a browser such as Internet Explorer or Netscape.	For the differences between XHTML and HTML see: http://webreview.com/wr/pub/1999/07/16/feature/index2.html
Metadata		"Data about data". Metadata is that which describes data in terms of its meaning or use. In SAS, we have the SQL dictionary tables, which describe tables, variables, etc. These dictionary tables could be described as metadata.	http://www.mdcinfo.com/

Term	Acronym	Description / Purpose	References
Meta-Language		A language that is used to describe or create other languages. XML isn't really a language at all, but rather a language the allows people to create languages.	http://www.whatis.com/meta.htm
Standard Generalized Markup Language	SGML	SGML is the international standard for defining the structure and content in electronic documentation. XML is a simplified version of SGML and was designed to take some of the complexity out of SGML for web-delivery.	http://www.whatis.com/sgml.htm
Tags		In a document markup language a tag is used to describe some information. For example, the tag <b> tells us that what follows should be bolded. In XML, we create our own tags like <customer> or <invoice>.	http://k12unix.larc.nasa.gov/training/tags.html
Valid		In addition to being well-formed, a valid XML document has a DTD or XML schema that defines its content and use.	http://www.xml.com/pub/98/10/guide3.html
Well formed		An XML document is said to be well formed if it follows all of the rules defined by the XML specification. Although well formed documents essentially follow the rules, they don't have a DTD associated with them.	http://msdn.microsoft.com/xml/general/what-is-xml.asp
Wireless Application Protocol	WAP	A new technology, originating in Europe, which maintains the specifications for how data should be encoded, transferred and processed with wireless applications. An example would be data transferred from a web server to a PalmPilot™.	http://www.wapforum.org/ http://www.wapforum.org/faqs/index.htm
XHTML	XHTML	XHTML is a specification approved by the W3C that allows users to create documents in HTML, but use some XML tags for embedding things like mathematical equations.	http://www.w3c.org/TR/xhtml1 http://webreview.com/wr/pub/1999/07/16/feature/index.html?wwwrrr_19990716.tx
XML Linking Language	Xlink	A draft language specification that describes how XML documents can be linked to one another (similar to HTML's linking <A HREF>.)	http://www.w3.org/TR/1998/WD-xptr-19980303
XML Namespaces	Namespaces	Currently a recommendation from the W3C as a standard to prevent the overlap of names used by different software vendors and/ or applications. For example, we may use the element name <status> to mean marital status whereas in our invoice, the name <status> may refer to whether the account is active or not. Namespaces help provide context to the names of the elements used.	http://www.w3c.org/TR/REC--xml-names
XML Parser		An XML parser reads a string of XML data and generates a structured tree (hierarchy). Some parsers validate the document against a DTD or schema.	http://www.zdnet.com/pcmag/features/xml98/parsers.html
XML Schema		Similar to an XML DTD but its rules are more rigorous. An XML schema provides a definition for how an XML document should be interpreted.	http://www.schema.net
XML Transformations	XSLT	A working draft from the W3C that describes ad language that can be used to transform XML content from one data format to another.	http://www.xmlmag.com/upload/free/features/xml/1999/01win99/kc2win99.asp

Multiprocessing with Version 8 of the SAS® System

Cheryl Doninger, SAS® Institute Inc.

INTRODUCTION

With the ever increasing need to get more done in the same amount of time, naturally we want our computer applications to take advantage of the multi-processors available in many of today's desktop and server platforms as well as to be able to multi-process across platforms available in a network. By dividing time-consuming tasks into multiple independent units of work and executing these independent units of work in parallel, a job can be performed in substantially less time than if each task is performed sequentially. A new feature has been added to Version 8 of the SAS System that allows your SAS jobs to take advantage of SMP hardware and to also allow parallel processing with the remote hosts in your network. This feature is part of the SAS/CONNECT® product and is called MP CONNECT. MP CONNECT allows you to perform disjoint units of work in parallel and coordinate all of the results into your original SAS session for the purpose of reducing the total elapsed time necessary to execute a particular application.

This paper will introduce the concept of multiprocessing and illustrate its benefits. Test results will be presented to show tangible proof of the time savings that are possible by modifying your existing jobs as well as designing new jobs to use MP CONNECT for multiprocessing. The syntax and options that enable MP CONNECT will be covered. And finally, the test case that was used to collect the results presented in this paper will be included in an appendix.

WHY MULTIPROCESSING

In this paper the term *multiprocessing* refers to dividing an application into independent sub-units of work that can be executed simultaneously. The primary purpose of multi- or parallel processing is to complete a job in less total elapsed time than it would take to execute the same job serially. With this capability IT staffs are challenged to have their applications take advantage of the multiple processors available in today's server platforms. In addition to exploiting standalone machines, IT staffs also benefit by multiprocessing across their networks. However, generally, the SAS System is a single-threaded application that executes on a single processor or single machine at a time.

Independent parallelism is possible when the execution of Task A and Task B do not have any interdependencies. An example of this in SAS procedure terms would be if an application needed to run PROC SORT against two different SAS data sets and then merge the sorted data sets into one final data set. Because there is no dependency between the two data sets that initially need to be sorted, the two PROC SORTs can be performed in parallel and when they both finish, the merge can take place. It is this type of parallelism that is addressed by MP CONNECT. MP CONNECT provides a convenient interface for spawning *n* SAS sessions to simultaneously execute *n* tasks as independent processes and coordinate the execution and results of all *n* tasks into the original SAS session. The *n* SAS sessions or processes can either all execute on the same machine with each session or process running on a separate processor or they can be directed to any number of remote machines. The remote machines can have either single or multiprocessor capabilities.

The following scenarios are meant to present real life applications that are ideally suited for independent parallelism and therefore MP CONNECT. Hopefully, these scenarios will help you to think of similar or additional SAS tasks which you could modify or develop to benefit from MP CONNECT.

Figure 1 illustrates a data warehouse scenario which requires extracting data from three unique and possibly remote data sources, combining that data, and then processing it. By performing the three extractions in parallel and then merging the three resulting data sets, you effectively decrease the time it takes to complete the entire task to the time of the longest extraction plus the time to perform the merge.

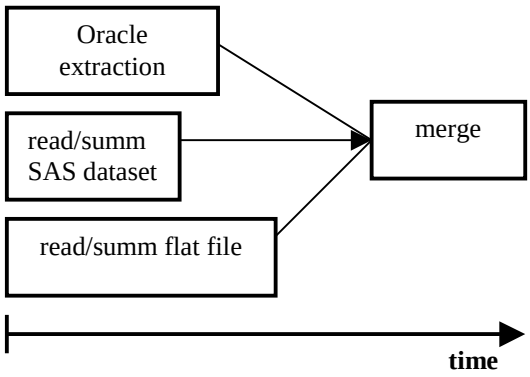


Figure 1. Data Warehouse Scenario

Figure 2 illustrates a SAS analysis example which requires running several independent SAS procedures against a single or possibly multiple data sources. Running these procedures in parallel can drastically reduce the total amount of time required to execute the whole job. Essentially, the time needed to execute the job now becomes the execution time of the longest running procedure.

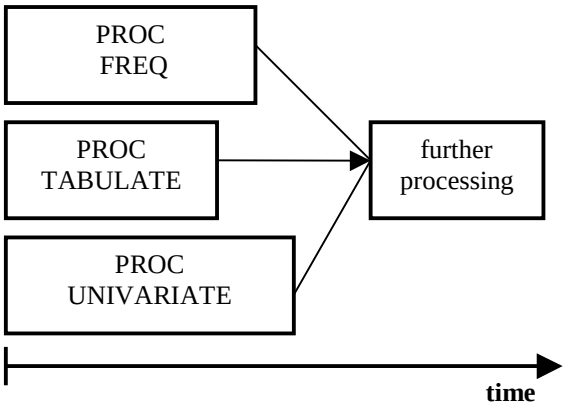


Figure 2. Multiple Analyses of Single SAS Data Set

Figure 3 is a HOLAP scenario which requires the creation of multiple MDDBs. Since the creation of an MDDB is an independent task and one which can require a significant amount of processing time, running the creation of multiple MDDBs in parallel is an excellent application for MP CONNECT. Depending on the number and size of the MDDBs that you create, the benefit of running these tasks in parallel with MP CONNECT could be very substantial.

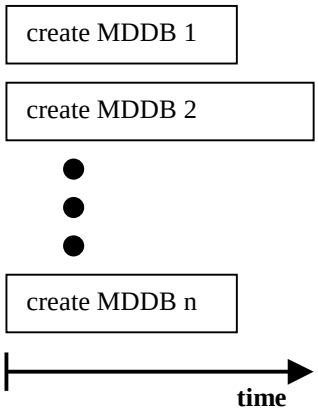


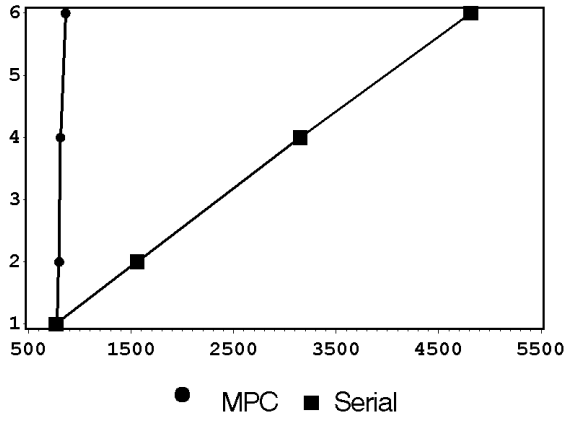
Figure 3. Parallel creation of Multiple MDDBs

TEST RESULTS SHOW THE BENEFIT

The test that was used to create the results presented in this section consists of a variety of Base SAS procedures run against data sets that vary in size. This test was first run using MP CONNECT to spawn *n* SAS processes and each process executed one instance of the test simultaneously. Then the test was run by executing *n* serial iterations of the same test. It was implemented using the macro facility so that the size of the data set and the number of processes created by the test (or in the case of serial execution, the number of times the test was re-iterated) could be easily varied. These tests were all run on a multi-processor machine, rather than remote machines on a network, in order to have a more consistent environment for the purpose of collecting test results. Version 8.0 of the SAS System was used to collect these results.

In the following graphs the x-axis represents time. The y-axis represents the number of parallel SAS processes or the number of iterations of the serial test.

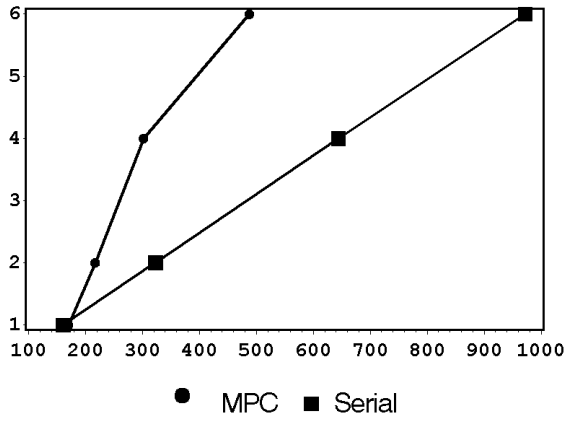
The first set of tests was run on a Sun Enterprise 10000 with twelve 400 MHZ Ultrasparc processors and the results are summarized in Graph 1.



Graph 1. Sun Enterprise 10000 Test Results

These test results show a remarkable time savings using MP CONNECT instead of serial execution. These results are also extremely positive with respect to scalability. For the serial case, increasing the number of iterations of the test produced a linear increase in the time necessary to complete the job. With MP CONNECT, however, there was a negligible increase in elapsed time as additional SAS processes were added.

The second set of similar tests was run on a Compaq ProLiant 8000 8 way server with 550 Mhz, 1 MB cache cpus running Windows NT. The results are summarized in Graph 2.



Graph 2. Compaq ProLiant Test Results

These results also show a remarkable time savings using MP CONNECT instead of serial execution. And, these results are very positive with respect to scalability. For the serial case, adding additional iterations to the test resulted in a linear increase in the time necessary to complete the job. With MP CONNECT, however, there was only a small increase in elapsed time as additional SAS processes were added.

GUIDELINES

It is important to note that not all applications are good candidates for the independent parallelism that MP CONNECT provides. Here are a few guidelines to help you determine if a particular application, or portions of it, can benefit by using MP CONNECT:

1. Determine if your data source(s) can be processed separately and independently. It may be worth while to duplicate some data in order to achieve this independence.
2. Look for ways to segment your job, or some portion of it, into sub-tasks that do not access joint non-sharable resources.
3. Ensure that your I/O subsystem can handle the additional data requirements introduced by parallel execution. You may need to spread your I/O across physical disks and/or I/O channels.

However, for those jobs that can be separated into independent units of work, the time it takes to complete the entire job can be drastically reduced by using MP CONNECT.

MP CONNECT –THE DETAILS

Prior to Version 8, SAS/CONNECT was a synchronous client/server tool with the emphasis on the ability to connect a SAS session running on a local machine to a SAS session running on a remote machine. With Version 8, MP CONNECT allows you to perform multi-processing with the SAS System by establishing a connection between multiple SAS sessions and enabling each of the sessions to asynchronously execute tasks in parallel. You also have the ability to merge the async tasks back into your local execution stream at the appropriate time. In addition, establishing connections to processes on the same local machine has been greatly simplified. This gives you the ability to exploit MP/SMP hardware as well as network resources to perform parallel processing of self-contained tasks and easily coordinate all the results into the original SAS session.

You can use MP CONNECT to spawn *n* SAS processes where *n* is the number of independent units of work that you wish to perform in parallel. SAS processes that are spawned on a single multi-processor machine are independent unique processes just as they are if they are initiated on a remote host. On Windows and Unix, for example, each SAS session is a separate process having it's own unique SASWORK library. Each process also assumes the user context of the parent, or the user that invoked the original SAS session, and possesses all of the rights and privileges associated with that parent. On MVS each SAS session is an MVS BPX address space that inherits the same STEPLIB and USERID as the client address space. The client's SASHELP, SASMSG, SASAUTOS, and CONFIG allocations are passed to the new session as SAS option values.

Normally, with SAS/CONNECT, the SIGNON command or statement is used to establish a connection between two SAS sessions. The SIGNON interface has been simplified to facilitate establishing connections to SAS sessions on the same machine. This is possible because less information is required for communication between processes on a single machine versus inter-process communication across a network. This new SIGNON interface eliminates the need for a script file on the local host, the need to re-specify userid/password information, the need to have a spawner running, and the need to perform any access method file configuration such as transaction programs, etc. The only parameters that you are required to specify with the SIGNON statement is the command to be used to invoke the SAS session and a name to associate with this new SAS process. A new option, SASCMD, is used to specify the SAS command that is used to invoke the "remote" SAS session. The SASCMD option can be specified in a global options statement as well. In addition, the NOTIFY option can be used on the SIGNON command or statement to request the display of a notification message window to report the completion of any subsequent asynchronous RSUBMITs.

Once the SIGNON has been executed and a connection established to one or more SAS sessions, either on the same machine or a remote machine, then the RSUBMIT command or statement can be used to asynchronously execute multiple independent tasks and reduce the overall execution time of your SAS job. In fact, the autosignon feature of RSUBMIT can be exploited to reduce the required syntax for spawning a new SAS session, sending it a unit of work, and terminating this session on completion to just a single RSUBMIT/ENDRSUBMIT block. This is possible because the RSUBMIT statement now accepts all the same parameters that the SIGNON statement does. The exact syntax and an example of this will be given in the next section.

The RSUBMIT statement is used along with the WAIT=NO option to identify that a unit of work should be asynchronously executed by the newly spawned SAS session. By executing the RSUBMIT asynchronously, you can start a long running task in one new SAS session and immediately be able to begin another task in another SAS session rather than wait until the first remote task is complete before regaining control of your original SAS session. And if NOTIFY=YES was specified during the signon process, a notification message window will be displayed in the local session when the asynchronous task completes execution.

For each asynchronous process, the default action is for SAS/CONNECT to spool the accumulated log and output lines from the remote process until you request the data by using the RGET command, executing a synchronous RSUBMIT, or issuing a SIGNOFF to terminate the remote SAS process. The RDISPLAY command can be used to view the current contents of the spooled log and output windows without merging the contents into the local log and output windows. Once the RGET command is issued, the accumulated log and output lines are retrieved and merged with the local log and output lines. Alternatively, you can use

the LOG= and OUTPUT= options on the RSUBMIT to either direct the log and output lines to a file or to purge them.

Another very important piece to this asynchronous multi-processing is the ability to synchronize any or all of your asynchronously executing tasks with subsequent local execution. This capability is provided with the WAITFOR statement which lets you suspend your local SAS processing pending the completion of any or all of your asynchronous tasks. For example, if you initiated two SAS sessions to simultaneously sort two data sources and then need to merge the sorted output, you could issue the WAITFOR statement in your original SAS session subsequent to the two async sorts and prior to the final data step used to merge the data.

The LISTTASK statement can be used to view the state of any or all active connections. It is most useful for viewing the state of asynchronously executing tasks.
And finally, the KILLTASK statement lets you maintain complete control by enabling you to terminate any or all asynchronous tasks with a single statement.

The following sections detail the statements and options that enable multi-processing with the SAS System.

SIGNON Command/Statement

The SIGNON command and the SIGNON statement are used to invoke another SAS session and establish a connection between the two SAS sessions. The newly created SAS session can be created either on the same machine or on a machine that is remote with respect to the parent or client SAS session. If the SAS session is being invoked on a remote machine, either a spawner and/or a script file can be used to provide the information necessary to invoke this new SAS session. If the SAS session is being spawned on the same machine as the originating or parent SAS session, then the new *SASCMD* option can be specified to provide the command necessary to invoke SAS. This option can be specified either with a global options statement, with SIGNON or, if the autosignon feature is being used, with RSUBMIT.

SASCMD="SAScmd";

where SAScmd specifies the command to be used to spawn a new SAS process. Additional SAS options can be specified within the quotes. If you need to execute additional host commands prior to the SAS invocation, it is recommended that you write a host specific script that contains your host commands and the SAS invocation, and specify this script as the SASCMD value.

On OS/390 hosts the *SASCMD* option is specified as follows:

SASCMD=":SAS-system-options";

where specifying any non-blank value will cause the UNIX *fork* command to spawn an MVS BPX address space which inherits the same STEPLIB and USERID as the client address space. The client's SASHELP, SASMSG, SASAUTOS, and CONFIG allocations are passed to the spawned SAS session as SAS option values. Any additional SAS invocation options can be specified following the colon. The XMS access method is used by the two sessions for communication.

The NOTIFY option can be used to request the display of a notification message window upon completion of an asynchronous RSUBMIT.

NOTIFY=YES|NO

where a value of YES causes a notification window to be displayed with the following message:

Asynchronous task JOB has completed.

where JOB is the remote session identifier. To acknowledge the message and to close the window, click the OK button.

RSUBMIT Command/Statement

The RSUBMIT command and the RSUBMIT statement cause SAS programming statements that are entered in the local environment to execute in a remote SAS session. Even though the statements execute in the remote environment, all responses and output are displayed in your local SAS log and output windows as they would be if you executed the program in the local SAS session.

RSUBMITs are processed in either *synchronous* or *asynchronous* mode.

Synchronous mode means that the remote processing must complete before the local session can continue processing. Therefore, the RSUBMIT must run to completion before you regain control. Synchronous processing is the default processing mode.

Asynchronous mode allows you to start an RSUBMIT in the background to a remote host and to regain local control immediately to continue with local processing or remote processing to another host.

The following RSUBMIT options enable asynchronous RSUBMITs:

CONNECTWAIT | CWAIT | WAIT=value

where value specifies whether this particular RSUBMIT is to be executed synchronously or asynchronously. This can also be specified with the CWAIT global option by submitting the following options statement:

OPTIONS CWAIT=NO;

The valid values for the WAIT= option are:

YES | Y	indicates a synchronous RSUBMIT.
NO | N	indicates an asynchronous RSUBMIT.

If WAIT=NO is specified, it may also be useful to specify the MACVAR= option. This will allow you to programmatically test the status of the current asynchronous RSUBMIT by determining whether it has completed or is still in progress.

CMACVAR | MACVAR=value

where value specifies the name of the macro variable to associate with this remote session. If specified on the

RSUBMIT command/statement, the MACVAR= option overrides any previous MACVAR= specifications for this remote session. The macro variable is NOT set if the RSUBMIT command fails due to incorrect syntax. Other than this one exception, the macro variable (value) is set to one of the following values:

0	Indicates that the RSUBMIT is complete.
1	Indicates that the RSUBMIT failed to execute.
2	Indicates that the RSUBMIT is still in progress.

Note: If a synchronous RSUBMIT (WAIT=YES) is issued while an asynchronous RSUBMIT (WAIT=NO) is still in progress, all spooled log and output statements are merged into the local log and output windows and the RSUBMIT continues as if it were synchronous. That is, you do not regain local control until the RSUBMIT has completed. If you don't want this to happen, use the MACVAR= option in the SIGNON or the RSUBMIT statements so that you can check the progress of RSUBMIT without causing it to execute synchronously.

LOG=KEEP | PURGE | filename | fileref
OUTPUT=KEEP | PURGE | filename | fileref

directs the SAS log or the SAS output that is generated by an asynchronous remote submit to the backing store, to be purged, or to a specified file.

KEEP spools log or output lines to the backing store for later retrieval with the RGET, RDISPLAY, or SIGNOFF statements. This is the default.

PURGE deletes all of the log or output lines that are generated by the current remote session. PURGE is used for economizing disk resources. If you can anticipate a large volume of log data or output data to the backing store that you do not want to receive, and you want to preserve disk space, setting PURGE can be useful.

filename | fileref specifies a file that is the destination for the log or output lines. The file is opened for output at the beginning of the asynchronous RSUBMIT and is closed at the conclusion of the asynchronous RSUBMIT. A subsequent RSUBMIT to the same file that you specify by using the LOG= or the OUTPUT= option overwrites the previous contents of that file with the contents of the current RSUBMIT.

WAITFOR Statement

The WAITFOR statement is used to make the local SAS session wait for the completion of one or more asynchronously executing tasks. If more than one task is specified, then the WAITFOR statement can include either the __ANY__ or the __ALL__ option. The __ANY__ option suspends the SAS session until the completion of any of the specified tasks (a logical OR). The __ALL__ option suspends the SAS session until the completion of all of the specified tasks (a logical AND). You can also specify a timeout value with the TIMEOUT option. If the specified tasks have not finished processing by the timeout value specified, the local session regains control and the tasks continue to execute asynchronously. In addition, the &SYSRC macro variable is set to indicate that a timeout occurred. If the specified tasks finish processing before the timeout value specified, the WAITFOR statement returns control to the local SAS session.

RGET Command/Statement

The RGET command and the RGET statement cause the spooled log and output from the execution of an asynchronous remote submit to be merged into the local log and output windows. When an asynchronous remote submit executes, the log and output statements are not merged into the local log and output windows. By default, they are spooled for retrieval at a later time. However, if the LOG or OUTPUT options have been used on the RSUBMIT to redirect the lines to an external file or to purge them, they will not be available for the RGET statement to merge into the local log and output windows.

If the RGET command or RGET statement is executed while the asynchronous remote submit is still in progress, all currently spooled log and output lines are retrieved and merged into the local log and output windows, and the remote submit continues processing as if it had been submitted synchronously. That is, you will NOT regain control in your local SAS session until the remote submit has completed. If you don't want the remote submit to become synchronous, but you want to check its progress, use the MACVAR option in the SIGNON or the RSUBMIT statement. This allows you to check the progress of an asynchronous remote submit without causing it to execute synchronously. Or, if you are running interactively, you could use the LISTTASK statement to check the status of the asynchronous remote submit before issuing the RGET statement.

RDISPLAY Command /Statement

The RDISPLAY command and the RDISPLAY statement create two windows for each asynchronous process that is executing to display the spooled log and output lines that have been generated. One window displays the log lines and the other window displays the output lines.

When an asynchronous remote submit executes, the log and output lines are not merged into the local log and output windows. By default, they are spooled to disk for retrieval at a later time. RDISPLAY allows you to view the spooled log and output lines created by the asynchronous remote submit without merging them into the local log and output windows. The log and output lines continue to scroll into the windows created by the RDISPLAY command as they are produced by the remote processing. The RGET command or statement, a synchronous RSUBMIT, or the SIGNOFF command or statement must be executed to actually merge the spooled lines into the local log and output windows. However, if the LOG or OUTPUT options have been used on the RSUBMIT to redirect the lines to an external file or to purge them, they will not be available for the RDISPLAY statement to display.

LISTTASK Statement

The LISTTASK statement lists information about any or all active connections. The LISTTASK statement displays

information such as the task name and it's current state of execution. A task can be in one of the following states of execution:

- READY
- RUNNING SYNCHRONOUSLY
- RUNNING ASYNCHRONOUSLY
- COMPLETE

When you SIGNOFF from a remote session or task it is removed from the list of tasks that LISTTASK maintains.

KILLTASK Statement

The KILLTASK statement is used to immediately terminate one or more asynchronously executing tasks. You can either specify a list of task names separated by spaces or the keyword __ALL__ if you want to terminate all of the asynchronously executing tasks. This statement is valid for tasks that are executing asynchronously on the same host and for tasks and SAS sessions that are executing asynchronously on remote hosts. KILLTASK causes any log or output lines that have accumulated in the backing store to be flushed to the parent or local Log and Output windows.

CONNECT Monitor Window

The SAS Explorer now provides a menu selection that enables you to monitor SAS/CONNECT tasks. Use the following path to access the SAS/CONNECT Monitor window from the SAS Explorer:

View -> SAS/Connect Monitor

The SAS/CONNECT Monitor window displays information about the tasks in two columns: Name and Status. An example follows:

Name	Status
Task1	Complete
Task2	Running Asynchronously
Task3	Running Synchronously

The list of tasks is dynamically updated as new tasks start, and the Status field changes from "Running" to "Complete," as appropriate. When you use the SIGNOFF statement to terminate a connection, the task is automatically removed from the window.

You may also terminate a task that is running asynchronously by clicking the task in the monitor window and selecting the Kill option from the menu that displays when you right-click the mouse button. Similarly, you can select the Rdisplay option from the menu to display LOG and OUTPUT windows for a task that is running asynchronously.

Example 1

The following example is a simplistic illustration of 2-way multiprocessing; the original SAS session executes a data step in parallel with an additional SAS session which executes a PROC SORT.

First, an asynchronous rsubmit is executed using the autosignon feature in order to spawn an additional SAS session and instruct it to execute the PROC SORT.

```
OPTIONS AUTOSIGNON=YES;
RSUBMIT SORTASK WAIT=NO SASCMD='SAS';
LIBNAME ANNUAL 'path to annual sales lib';
PROC SORT DATA=ANNUAL.SALES
        OUT=ANNUAL.SORT1;
  BY REGION;
RUN;
ENDRSUBMIT;
```

The local SAS session can then be used to simultaneously perform additional processing because the above PROC SORT is being processed asynchronously by a separate SAS session. In the following section, a data set is created and then the WAITFOR statement is used to synchronize the completion of the above PROC SORT with a subsequent data step to merge the two data sets.

```
LIBNAME LOC 'path to monthly sales lib';
DATA LOC.MARCH;
DO I = 1 TO NREGIONS;
/* create local data set */
END;
RUN;

WAITFOR SORTASK;

LIBNAME ANNUAL 'path to annual sales lib';

DATA TOTAL;
SET ANNUAL.SORT1 LOC.MARCH;
RUN;
```

Example 2

The following example assumes that you have a multi-processor machine on which you would like to run MP CONNECT tasks in parallel. You do not have direct access to this machine but are able to signon to it from your local workstation. The following code is a template that could be used to signon to a multi-processor server and then remote submit several tasks to run in parallel on that server.

```
FILENAME RLINK <script file for server>;
%LET HOST1=<my.smp.server.box>;
/* signon from workstation to server */
SIGNON HOST1;

/* rsub multiple tasks to server */
RSUBMIT;

 OPTIONS AUTOSIGNON=YES SASCMD="SAS";
 RSUBMIT TASK1 WAIT=NO;
   /* stmts processed by TASK1 */
 ENDRSUBMIT;

 RSUBMIT TASK2 WAIT=NO;
   /* stmts processed by TASK2 */
 ENDRSUBMIT;

 . . .

 RSUBMIT TASKn WAIT=NO;
   /* stmts processed by TASKn */
 ENDRSUBMIT;
```

```
WAITFOR __ALL__ TASK1 TASK2 TASKn;

ENDRSUBMIT;
```

FUTURE RESEARCH & DEVELOPMENT

We are currently working on several enhancements to facilitate the use of MP CONNECT including but not limited to:

- specifying a remote session id on %syslput statements to direct execution to a specific remote session,
- detecting active asynchronous tasks upon normal SAS termination and prompting for the appropriate action to take, and
- allowing spawned or remote SAS sessions to have easy access to libraries defined to the local or parent SAS session.

In addition, we have prototyped an implementation of *pipeline parallelism.* Pipeline parallelism is possible when Task B requires output from task A, but it does not require all of the output before it can begin execution. In SAS terms, this means that two SAS procedures could run in parallel such that one proc feeds it's output to the next proc as input. For example, a SAS data step could feed observations, as they are created, to PROC SORT allowing the SORT procedure to run in parallel with the data step minimizing the total time required to complete both steps. In other words, a procedure does not write it's output to disk, but rather pipes it as input to the next procedure. We are currently researching the benefits and considerations for this type of parallelism.

CONCLUSION

This paper presents the new Version 8 MP CONNECT feature that enables you to perform parallel- or multi-processing with the SAS System. This feature provides you with a straight forward syntax to allow you to make minimal modifications/additions to your existing or new SAS jobs in order to substantially decrease the total elapsed time necessary to execute a job.

It is strongly recommended that each SAS application be evaluated for potential benefits before implementing the MP CONNECT feature. For those jobs that perform time consuming tasks that can be separated into independent units of work, MP CONNECT can be used to decrease the time of execution to a fraction of what is required to execute the same job serially. MP CONNECT is also extremely scalable which means that you will continue to recognize tremendous time savings as the number of SAS processes running in parallel approaches the number of processors on your system or in your network.

Author

Cheryl Doninger
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
(919) 677-8000
Cheryl.Doninger@sas.com

APPENDIX – THE TEST CASE

This appendix contains the SAS job that was run to collect the test results that are presented in this paper. This test was originally used for another purpose and was run serially. The test was modified to use MP CONNECT for the purpose of collecting results for this paper. The statements that were required by MP CONNECT have been highlighted with bold font in order to illustrate just how minimal the changes were to this particular test. Notice that there are only six such statements. Similar minor additions could be made to your existing SAS jobs to take advantage of your MP/SMP hardware as well as remote hosts on your network to dramatically reduce the total execution time of your jobs.

```
options fullstimer ;
options autosignon=yes;
options sascmd='sasv8';

data _null_ ;
 host=sysget('HOST') ;

 call
symput('numloop',compress(trim(scan("&syspar
m",1,"."))));
 call symput('datsz',trim(scan("&sysparm",2,".")))
;
 call symput('host',trim(host)) ;

run ;


%macro benchds(numrecs);
  libname foo v8 '/tmp';
  data foo.tstdata;
   /* create a data set with 110 variables and obs
equal
       value of numrecs passed into the macro
*/
  %mend ;

%benchds(&datsz) ;

%global time1;
%macro pretime;
   data _null_;
     time=put(time(),6.);
     call symput('time1',time);
   run ;
%mend;

%macro postime(sec);
   data _null_;
     time=time()- %str(&time1);
     put '*';
```

```
      put "******time elapsed(&sec) = " time 6.;
      put '*';
run ;
%mend;

%macro shutdown() ;
  %local runid;
  %let runid=1 ;
  %let remsessions=;
  %do %while(&runid le &numloop);
     %let remsessions=&remsessions rem&runid;
     %let runid=%eval(&runid+1) ;
  %end;

waitfor _all_ &remsessions;

%postime(0);

  %let runid=1 ;
  %do %while(&runid le &numloop);
      proc printto log="mploop&runid..log"
                print="mploop&runid..lst" new ; run;
      rget rem&runid;
      signoff rem&runid;
      %let runid=%eval(&runid+1) ;
      proc printto; run;
  %end;


%mend ;

%macro startup(subsys) ;
%pretime;

proc datasets library=work ;
  delete stats1 stats2;
quit;

   %put APR HEADER os=&sysscp;
   %put APR HEADER host=&host;
   %put APR HEADER ver=&sysvlong;
   %put APR HEADER subsys=&subsys;
   %put APR HEADER numobs=&datsz;
   %put APR HEADER duration=&numloop;

%mend ;

%macro runtest(runid) ;
/*****************************************
  *    create a new MP CONNECT session to
 *    handle this iteration
 *****************************************/
   rsubmit rem&runid wait=no;
   libname foo v8 '/tmp';

/*****************************************
*    Step CONTENTS_2
*****************************************/

   proc contents data=foo.tstdata;

/*****************************************
 *   STEP SORT_3
  *****************************************/

   proc sort data=foo.tstdata out=out1 tagsort ;
    by descending x1_10 stname8 ;
   run ;

/*****************************************
  *   STEP FREQ_4
  *****************************************/

   proc freq data=foo.tstdata;
    tables stname20 onein10 onein100 onein1k ;
    title 'proc freq ';
   run ;

/*****************************************
 *   STEP SUMMARY_5
  *****************************************/

   proc summary data=foo.tstdata print ;
    title 'proc summary full data set ' ;
    var _numeric_ ;
   run ;

/*****************************************
  *   STEP SUMMARY_6
 *****************************************/

   proc summary data=foo.tstdata print ;
    title 'proc summary three state names
subsetted by where clause' ;
    var _numeric_ ;
    where stname20='ALABAMA' or
stname20='CALIFORNIA' or stname20='TEXAS'
;
   run ;

/*****************************************
  *  STEP DATA_7
  *****************************************/

   data temp;
    set foo.tstdata;
    if stname20='ALABAMA' or
stname20='CALIFORNIA' or
stname20='TEXAS';
   run ;
```

```
/*****************************************
 *   STEP SUMMARY_8
 *****************************************/

   proc summary data=temp print ;
    title 'proc summary three state names
subsetted by data set' ;
    var _numeric_ ;
   run ;


/*****************************************
 *  STEP DATA_9
 *****************************************/

   data _null_;
    set foo.tstdata;
    where onein100='y';
   run ;

/*****************************************
*   STEP DATA_10
 *****************************************/

   data _null_ ;
    set foo.tstdata;
    if onein100='y' ;
   run ;

/*****************************************
*   STEP DATA_11
 *****************************************/

   data _null_ ;
    set foo.tstdata;
    where x1_1000=9 or x1_1000=99 or
x1_1000=999 ;
   run ;

/*****************************************
*   STEP DATA_12
 *****************************************/

   data _null_ ;
    set foo.tstdata;
   run ;

/*****************************************
 *   STEP TRANSPOSE_13
 *****************************************/

   proc transpose data=foo.tstdata (keep=f1
x1_10) out=trans ;
    by x1_10 notsorted ;
    var f1 ;
   run ;
```

```
/*****************************************
 *   STEP SORT_15
 *****************************************

   proc sort data=foo.tstdata out=out1 ;
    by x1_100 stname8 ;
   run ;


 *****************************************
 *   STEP SUMMARY_16
 *****************************************/

   proc summary data=foo.tstdata;
    title 'proc summary full data set ' ;
    var _numeric_ ;
    output out=stats1 ;
   run ;

/*****************************************
*   STEP SUMMARY_17
 *****************************************/

   proc summary data=foo.tstdata;
    title 'proc summary three state names' ;
    var _numeric_ ;
    where stname20='ALABAMA' or
stname20='CALIFORNIA' or stname20='TEXAS'
;
    output out=stats2 ;
   run ;

/*****************************************
*   STEP SUMMARY_18
*****************************************/

   proc summary data=temp  ;
    title 'proc summary three state names' ;
    var _numeric_ ;
    output out=stats2 ;
   run ;
endrsubmit;
%mend ;

%macro duration();
  %local runid ;
  %let runid=1 ;
  %do %while(&runid le &numloop);
        %runtest(&runid) ;
        %let runid=%eval(&runid+1) ;
   %end;
%mend;

%startup(Perf);
%duration ;
```

```
%shutdown;
```

The Metamorphosis of a Study Design

Marge Scerbo, CHPDM/UMBC
Craig Dickstein, Intellicisions Data Inc.

Abstract

In a perfect world, there would be perfect data, perfect analysts, and perfect programmers creating perfect outcomes to every possible study. Unfortunately, one, two, or all of these factors are usually imperfect. Data are, especially data in large volumes, rarely flawless. Researchers and analysts designing studies may have great ideas of studies to undertake, but may have little idea of whether it can be done or how to do it. Programmers may be incredibly facile with the software, but rarely comprehend all the intricacies needed to complete a study. Thus, study designs are not often etched in stone. Most likely, they are the outcome of a long and tedious process of checks and balances.

This paper will take the reader through the process of developing a study design, using SAS software to provide results on which to base outcomes. A health care policy issue will be used as the basis for the discussion, but the ideas should carry across many industries.

Introduction

A good programmer analyst must work with a variety of methodologies within a single project. The 'programmer' portion of the brain is organized, methodical, and logical. The 'analyst' is quite a different story; patience, foresight, and a depth of understanding of both the data and the outcome, beyond merely understanding code structures, is required. In a sense, the analyst must be a mind reader and a magician.

Health care data are a world unto themselves. There are vast amounts of administrative (billing) data produced daily. Except for the payment and/or collection of bills, these data are largely underused or misused.

Note that the study discussed in this paper is fictional, and that none of the data can be associated with any state or institution.

Background

Health care billing data are in three primary formats: HCFA-1500, Pharmacy, and UB-92.

HCFA-1500 records contain professional service information provided by an 'individual' practitioner. The place of service for these claims and encounters can encompass many venues, including doctor's offices, laboratories, and hospitals. Usually, one record is analagous to one procedure in a physician's office, and so one visit may include multiple records.

Pharmacy data are the cleanest, most efficient, and easiest to manipulate. Most pharmacy data are collected at the 'point of sale' (POS), right in the drug store. These records contain information about the drug, the prescription, the provider, and the patient. One record is essentially one prescription.

On the other hand, although they represent the largest percent of health care dollars, UB-92 data are not easy to use. These files are uniquely produced by facilities: acute care hospitals, hospices, nursing homes, emergency rooms, and outpatient clinics. These data are far more complicated when used for analysis. There is no clear definition for an inpatient data record; this depends on the database design of the keeper of the data.

Initial Study Design Proposal

HMOs (Health Maintenance Organizations) and MCOs (Managed Care Organizations) are based on the premise that by providing good preventive care and case management, fewer facility charges will be incurred and overall cost will decline. In this vein, HMO and MCO management is always looking at the bottom line for possible savings.

One HMO administrator was asked to identify methods of saving money on hospitalizations. He reviewed the various monthly and quarterly reports relating to this subject. After several weeks of review, he decided that it might be possible to save money by transferring patients from a hospital into a nursing home setting more quickly than was presently the case.

Hence, a new study is launched…

Under the present HMO contract with a large corporation, the HMO was responsible to pay for the first 31 days in a nursing facility.

A meeting took place between the administrator, an analyst, and a programmer. In order to best understand what analysis should be undertaken to get the best results, the analyst prepared a list of questions (the administrator's responses are shown in *italics*):

- What type of data should be studied?

- *Inpatient*

- As inpatient data are requested, can you specify exactly what type of facility should be studied? Inpatient facilities include acute care hospitals, rehabilitation centers, chronic hospitals, hospices, and other special settings.
  - *Acute Care Hospitals*
- Nursing homes are classified as three different types: ICF (Intermediate Care Facility), SNF (Skilled Nursing Facility), and ICF-MR (Intermediate Care Facility for the Mentally Retarded). Are you interested in all types or specific ones?
  - *ICF only*
- For acute care services, can we narrow the population studied by other criteria?
  - *Include only those patients that can be identified as still hospitalized for over 5 days.*
- Are these same criteria to be imposed on the ICF population?
  - *No*
- To continue, what type of information would be useful?
  - *The number of patients*
  - *The total dollar amount*
  - *The total number of days*
  - *The average cost per day*
- In addition, what information do you want from the ICF file?
  - *Same as that from the acute care file*
- Finally, is there a specific set of dates of services you are interested in?
  - *Whatever file is the most current and yet the most complete.*

The analyst identified calendar year 2000 data as the most complete and the most current. The first step proposed in the study design is to count the number of patients at the close of calendar year 2000 who remain in the hospital and the number of patients who were in an ICF during calendar year 2000.

The initial programming request includes the following:

- Obtain a frequency count on discharge status in the UB-92 year 2000 acute care summary file
- Select those patients who are still hospitalized
- Select those patients who have been hospitalized over 5 days
- Calculate their cost per day
- Calculate the cost per day for patients who are in an ICF
- Calculate the difference in costs
- Produce tabular reports on the demographic (age and gender) identifiers of selected populations

These reports are to be delivered within one week of the initial proposal.

Overview of the Data

UB-92 data are both complicated and extensive. These data files are not comparable to a hospital medical record, which contains notations on every drug, laboratory test, physician visit, procedure, etc. that is incurred during a patient stay. Rather, these files contain billing data. Services are collapsed into revenue codes with units of service and charges attached. For example, multiple laboratory services may be grouped under the revenue code 300, 'Laboratory, General Classification'. In addition, one inpatient hospital stay may in fact be billed across several UB-92 records, some with charges and others with adjustments, some across a particular date range and others across the date range from admission to discharge.

This HMO's IS department, in order to better utilize UB-92 files, had written SAS programs to create discharge summaries; where all possible records associated with a unique patient stay were stored in one large record. Charges, units, and days are totaled under this file structure design. These data sets have been validated for quality and are sorted by the recipient identifier (RECIPID) to allow for ease in merging processes. In this situation, the discharge summary data set for acute care inpatient discharges (ACUTE00) and a separate discharge summary data set for nursing home facilities (LTC00) will be used. Both of these data sets also contain two important fields for analysis: the length of stay (LOS) field containing the total number of days of the stay and the payment (PAYMENT) field containing the total cost of the stay.

The demographic information on each patient (AGE, GENDER) is stored in an annual enrollment file that contains one record per patient.

Preliminary Analysis

In order to satisfy the first request, the programmer ran a frequency analysis of the discharge status (DISCHSTATUS) in the acute care discharge summary file (ACUTE00). There exists a format for the discharge status

(STATUS.) that is used to produce more readable results:

```
proc freq data = acute00;
   tables dischstatus;
   format dischstatus status.;
run;
```

The results are shown in Table A. This frequency analysis demonstrates that 1,021 patients were transferred to an ICF in calendar year 2000 while 844 patients were still in the hospital at the end of the year. Code 30 is defined as 'still in the hospital' and code 5, 'transferred to ICF'.

The analyst, stepping beyond the exact specifications, request frequency counts on the length of stay field found in the acute care file. Rather than producing a report that showed the count of each length of stay, the programmer grouped the days to provide a more concise report:

```
proc format;
   value days
          0-5     = '0-5'
          6-high  = '> 5'
run;
proc freq data = acute00;
   tables los;
   format los days.;
   where dischstatus = 30;
run;
```

The output of this report is shown in Table B. It clearly shows that of the 844 patients still in the hospital, *only 32.7% of the patients (276 people) had stays of over 5 days*. This table serves several purposes, the most important being a checkpoint for the files to be created in the next steps of the study.

The programmer then proceeds to create the requested data set containing only those patients still in the hospital and with a stay greater than 5 days:

```
data stillin;
   set acute00 (where = (los gt 5
        and dischstatus = 30));
run;

The LOG reports:

NOTE: The data set WORK.STILLIN has 276
      observations and 59 variables.
```

This data step allows the analyst a double check that 276 patients fell into this category, as was reported on the discharge status frequency analysis. In addition, this data set can now be used for further studies.

The average cost per day of these 276 patients is then calculated using PROC SUMMARY. (Note that PROC MEANS also provides similar output.)

```
proc summary data = stillin;
     var payments los;
     output out= inptcost
            sum= inptdol inptdays;
run;
data results;
     set inptcost;
     costperday = inptdol/inptdays;
run;
proc print;
     var _freq_ inptdol inptdays
         costperday;
run;
```

Output of this routine (Table C) demonstrates the following:

- The number of patients (_FREQ_) is 276
- A total cost (INPTDOL) of $15,779,690
- A total number of days (INPTDAYS) of 6,053
- Thus, the average cost per day is $2,606 (INPTDOL/INPTDAYS)

A similar process is then used to calculate the number of people in ICFs. Since there are several types of long term care facilities contained in this file and the request specified that only ICFs were to be included, a provider type (PROVTYPE) code of 57 is used to identify ICFs:

```
data nursinghome;
    set ltc00  (where= (provtype = 57));
run;

NOTE: The data set WORK.NURSINGHOME has
      5983 observations and 43 variables.
```

The total costs are calculated thus:

```
proc summary data = nursinghome;
     var payments los;
     output out = nhcost
             sum = nhdol nhdays;
run;
data results;
     set nhcost;
     costperday = nhdol/nhdays;
run;
proc print;
```

```
    var nhdol nhdays _freq_
       costperday;
run;
```

Table D.1, containing the output of this procedure, shows:

- The number of patients (_FREQ_) is 5,983
- A total cost (NHDOL) of $876,121,178
- A total number of days (NHDAYS) of 1,143,242
- Thus an average cost per day is $767 (NHDOL/NHDAYS)

Thus, the difference of cost per day between the inpatient hospital and the nursing home is $1,839 ($2,606 - $767).

Warning – Inconsistency Detected

Although the programmer had followed the specifications, certain inaccuracies were apparent when this information was presented to the analyst:

- There are 1,021 patients identified in Table A as transferred from an inpatient hospital to an ICF. The results of the ICF study showed 5,983 patients in nursing homes. What caused this discrepancy?
- The HMO administrator had stated that the HMO is required to pay only up to 31 days in a nursing home facility. Should this be included in the design?

Before completing the final tasks listed in the initial study design, updates were required to the study.

Revised Study Design

The study design is now revised to include additional subset criteria:

- Include only patients who can be identified as having transferred from a hospital to an ICF (discharge status of 5)
- Of those patients, select only those with a length of stay of 31 days or less

Additional Analysis

The first step to be completed is the selection of those patients who were transferred from a hospital to an ICF:

```
data icf;
     merge acute00 (in = hosp keep=
                    recipid dischstatus
                   where=(dischstatus=5))
           nursinghome (in = ltc
                   where=(los le 31));
     by recipid;
     if hosp and ltc;
run;
```

```
NOTE: The data set WORK.ICF has 1021
      observations and 44 variables.
```

This process identifies the desired 1,021 patients who have been transferred from a hospital to a nursing home. The next step is to summarize the payments for this group of patients:

```
proc summary data = icf;
     var payments los;
     output out = icfcost
            sum = icfdol icfdays;
run;
data results;
     set icfcost;
     costperday = icfdol/icfdays;
run;
proc print;
     var _freq_ icfdol icfdays
         costperday;
run
```

The output (Table D.2) contains this information:

- The number of patients (_FREQ_) is 1,021
- A total cost (ICFDOL) of $8,592,300
- A total number of days (ICFDAYS) of 5,242
- Thus, an average cost per day of $1,639 (ICFDOL/ICFDAYS).

Consequently, the difference in cost per day between the inpatient hospital and the nursing home is $967 ($2,606 - $1,639). Note that the number of patients now matches the number identified in the original frequency analysis (276 patients still in the hospital and 1,021 patients transferred from a hospital to an ICF).

With the populations of both facilities correctly identified, the programmer needs to add the demographic information to the newly created data sets:

```
data stillin;
     merge stillin (in = hosp)
           enrollment00 (keep= recipid
                        age gender);
     by recipid;
     if hosp;
run;
```

Once the new variables have been added to the data set, the programmer produces the demographic tables requested using this simple code:

```
proc format;
     value ages
            0  - 18   = 'Children'
            19 - high = 'Adults';
run;
proc freq data = stillin;
     tables age * gender / list;
     format age ages. gender $sex.;
     title 'Demographic Identifiers
            for Still in Hospital';
run;
```

The demographic studies can be found in Tables E and F.

Review of the Materials Produced

A meeting took place with the original team and a clinical specialist to provide further insight. Each report was studied carefully. Although there were clearly differences between the per day cost of a hospital and a nursing home, the clinician did not believe there was enough information to implement policy changes. The demographic tables (E and F) did not provide any significant decision-making information.

Without further studies to assess the diagnoses involved or the clinical appropriateness of transfers from a hospital to an ICF, no action plan could be set in place. Was it possible there were other areas of study that might provide clearer results?

After discussion, additional studies were requested that included:

- Produce a report of the 15 top primary diagnoses associated with those patients still hospitalized
- Produce a report of the 15 top primary diagnoses associated with those patients in ICFs
- Create a list of the top five costliest hospitals for those patients still in the hospital
- Create a list of the top five costliest nursing homes to which hospital patients were transferred

Additional Coding

In order to complete the final programming request, the programmer created various SAS programs. The first task was to create a report showing the top 15 diagnoses for patients still in the hospital as defined by the study design:

```
proc freq data = stillin order = freq;
     tables primarydx / noprint
                        out = acutedxs;
run;
proc print data = acutedxs (obs = 15);
     var primarydx count;
     title 'Top 15 Diagnoses for
            Still in Hospital';
     format primarydx $icdcode.;
run;
```

The results of these studies are displayed in Table G. This code can be edited to create the same report (Table H) on patients in an ICF.

The next step is to produce a listing of the top 5 hospitals and nursing homes. This code was written:

```
proc summary data = stillin nway;
     class hospital;
     var payment;
     output out = hospcost
            sum = totcost;
run;
proc sort data = hospcost;
     by descending totcost;
run;
proc print data = hospcost (obs=5);
     var hospital totcost;
     format totcost dollar15.;
     title 'Top 5 Hospitals';
run;
```

The results of this analysis are shown in Table I, with the corresponding information on ICF costs in Table J.

Final Study Design

The final study design is written to include all the steps necessary to produce the required results:

- Obtain a frequency count on discharge status in the UB-92 year 2000 acute care summary file
- Select those patients who are still hospitalized
- Select those patients who have been hospitalized over 5 days
- Calculate their cost per day
- Identify patients who have been transferred from a hospital to an ICF (discharge status of 5)

- Of those patients include only those with a length of stay of 31 days or less
- Calculate the cost per day for patients who are in an ICF
- Calculate of the difference in per day costs between hospital and ICF facilities
- Produce tabular reports on the demographic (age and gender) identifiers of the selected populations
- Produce a report of the 15 top primary diagnoses associated with those patients still hospitalized
- Produce a report of the 15 top primary diagnoses associated with those patients in ICFs
- Create a list of the top five costliest hospitals for those patients still in the hospital
- Create a list of the top five costliest nursing homes to which hospital patients were transferred

Conclusion - Outcome of Study

As shown, this study was implemented as simple frequencies and iteratively enhanced as each resulting table was available. Since this analysis might have direct impact on patients' treatments, it was important that clinical input was requested.

During the wrap-up meeting of the study team, one surprising and unexpected result was identified. Although there could be no final decision to move patients after 5 days from a hospital to a nursing home, it was clear from Table I (Top 5 Hospitals), that one hospital, University Center, accrued the largest dollar amount for patients hospitalized for over 5 days.

The clinician was aware of the variations of specialties across the hospitals, but could not explain the wide variation in total costs. It was determined that a meeting between the HMO and hospital administrators was needed. At that time, discussions as to the length of stay of patients in the hospital were discussed. The possibility of medical record review was proposed to assess this issue.

In addition, there were still several questions pending. Is this truly the final study or is this all that is available in the time alloted? Can any requirements be placed on physicians concerning length of stay that are clinically sound?

Should additional analysis take place? For example:

- Does the type of condition and the status of the patient control the outcome?
- What other factors may affect overall length of stay?
- Do patients recover more quickly in hospitals?

Complete study designs are imperative to valid analyses. They are created over time and may in fact present unexpected results. Programmers and analysts who can work within the variation of needs and personalities fill an invaluable role in any organization.

For more information on study designs, check the case study discussed in *Health Care Data and the SAS System*.

References

Scerbo, M., Dickstein, C., and Wilson, A. (2001). Health Care Data and the SAS System, Cary, NC: SAS Institute, Inc.

Contact Information

For more information contact:

Marge Scerbo
CHPDM/UMBC
1000 Hilltop Circle
Social Science Room 309
Baltimore, MD 21250
Email: scerbo@chpdm.umbc.edu

Craig Dickstein
Intellicisions Data Inc.
P.O. Box 502
Weare, NH 03281
Email: cdickstein@att.net

Table A - Frequency of Patient Status, CY2000

Status	Frequency	Percent	Cum. Frequency	Cum. Percent
Disch/Trans to Home or Self Care	87,219	85.4	87,219	85.4
Disch/Trans to Other Hospital	1,688	1.6	88,907	87.0
Disch/Trans to SNF	5,619	5.5	94,526	92.5
Disch/Trans to ICF	1,021	1.0	95,547	93.5
Disch/Trans to Other Institution	2,420	2.3	97,967	95.9
Left Against Medical Advice	1,397	1.3	99,364	97.2
Patient Died	1,901	1.8	101,265	99.1
Still in the Hospital	844	0.8	102,109	100.0

Table B - Formatted Frequency of Total Days - Still in Hospital

Totaldays	Frequency	Percent	Cum. Frequency	Cum. Percent
0-5	568	67.3	568	67.4
> 5	276	32.7	844	100.0

Table C - Output of Summary on Acute Care Patients

# of Patients (_FREQ_)	Total Dollars	Total Days	Average Cost per Day
276	$15,779,690	6,053	$2,606

Table D.1- Output of Summary on ICF Patients

# of Patients (_FREQ_)	Total Dollars	Total Days	Average Cost per Day
5,983	$876,121,178	1,143,242	$767

Table D.2 - Output of Summary on ICF Patients - Amended

# of Patients (_FREQ_)	Total Dollars	Total Days	Average Cost per Day
1,021	$8,592,300	5,242	$1,639

Table E - Demographic Identifiers for Still in Hospital

Female		Male	
Children	**Adult**	**Children**	**Adult**
26	**99**	**43**	**108**

Table F - Demographic Identifiers of Transferred to ICF

Female		**Male**	
Children	**Adult**	**Children**	**Adult**
56	452	65	448

Table G – Top 15 Diagnoses for Still in Hospital

Primary Diagnosis	Count
Schizoaffective Disorder	32
Congestive Heart Failure	28
HIV Aids	23
Pneumonia	23
Respiratory Distress	23
Hypovolemia	16
Septicemia	16
Rehabilitation Procedure	15
Depress Psychosis	12
Extreme Immaturity	12
Food/Vomiting	12
Paranoid Schizophrenia	12
Staphyloccal Pneumonia	10
Bipolar Affective Disorder	9
Decubitus Ulcers	9

Table H – Top 15 Diagnoses for ICF

Primary Diagnosis	Count
Senile Dementia	240
CVA	198
Cardiovascular Disease	99
Alzheimers	76
Cerebrovascular Disorder	32
Hip Replacement	26
Hypertension	24
Psychosis	18
Paralysis Agitans	17
Multiple Sclerosis	14
Diabetes	14
Decubitus Ulcers	9
Presenile Dementia	7
Depressive Disorder	7
Neoplasm	5

Table I - Top 5 Hospitals

Hospital	Total Payments
University Center	$2,187,634
Ben Franklin Hospital	$1,506,123
Union Square Hospital	$414,168
Childrens Center	$361,421
St. Johns	$310,448

Table J - Top 5 ICFs

Hospital	Total Payments
Freetown Rehabilitation Center	$146,940
Morristown Center	$121,461
Main Eldercare	$98,821
Northeast Convalescent Home	$96,503
St. Michaels Nursing Center	$87,606

Introduction to the SAS® Macro Language

Thomas J. Winn, Jr.
Texas State Auditor's Office, Austin, Texas

Why bother learning about macros?

Have you ever wanted to "package" blocks of SAS code into components that could be invoked with a single command? This would be desirable, for instance, in repetitive coding situations involving the processing of numerous, similar data sets in identical ways.

Have you ever wanted to write a SAS program that modifies itself while it is running? Perhaps you wanted to write flexible SAS code with the ability to accommodate itself to certain changeable details -- for instance, to check for data requiring special processing, to dynamically generate data-dependent lines in a report, or to communicate particular information between the steps in a program.

Have you ever wanted to be able to build and utilize a customized "toolbox" of frequently-used programming modules, without having to explicitly list them in each program in which they are used?

For these situations, and for many others, use of the SAS macro language can produce the desired results. The SAS macro language is a powerful tool for producing flexible or repetitive code by using a modularized approach. It is particularly useful in applications development.

What are Macros?

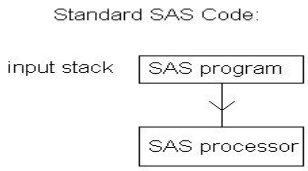
Macros are stored text which contain entire blocks of SAS code, and which are identified by a name. The stored text can include SAS statements, literals, numbers, macro variables, macro functions, macro expressions, or calls to other macros.
Macro variables are used to facilitate symbolic substitution of strings of text, whereas macros can be used to manipulate SAS source statements.

Macro information can be inserted at any point in a SAS program simply by referring to the macro entity by name, preceded by a special character, which distinguishes macro statements from ordinary SAS code. Macro variables are identified using the ampersand (&), and macros are identified using the percent sign (%). These special characters (& and %) are "macro triggers." If either of these triggers are followed by a non-blank character, the macro facility takes control over processing.

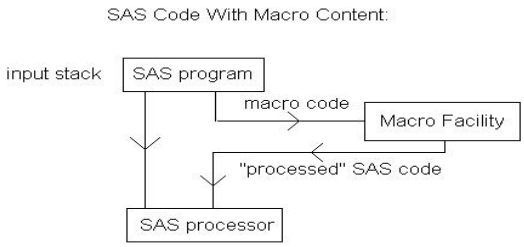
The macro facility constructs and edits SAS source statements by substituting the currently-defined values of macro variables, and also by replacing macro names with the stored text which is associated with each of them.

So, what does "macro" do?

Standard SAS code (that is, SAS code which does not contain any macro content) is passed directly from the input stack to the SAS processor.



On the other hand, pieces of SAS code that contain any macro information are intercepted, and are passed to the macro facility for interpretation before they are executed.



The SAS Macro Language is a programming language.

The macro facility follows instructions that are written using a special language. The SAS macro language has variables, statements, functions, expressions, and syntax. It works in conjunction with the SAS programming language.

Macro Variables

Macro variables are used to facilitate symbolic substitution of strings of text. Macro variables are named following the usual rules for SAS names. A macro variable can be referenced by placing an ampersand immediately before the macro variable name and, optionally, by placing a period after the macro variable name. A macro variable reference causes the macro facility to substitute the current value of the macro variable for the reference.

There are two types of macro variables: automatic macro variables, which are defined by the SAS Supervisor, and those, which are defined by the programmer. Once it has been appropriately defined, a macro variable may be used repeatedly in the SAS job. Automatic macro variables are created by the SAS Supervisor when the SAS System is invoked. They include information such as the time, day, and date on which the job began executing, and the name of the most recently created SAS data set. Automatic

macro variables are available for the duration of a SAS job and can be referenced anywhere in any SAS program.

The %LET statement is the most common way to create user-defined macro variables, and to assign values to those variables.

The value assigned to a macro variable is treated as a character string by the macro facility, but it isn't necessary to enclose the string in quotes. If quotes are used, the quotes become part of the string.

There is no such thing as a numeric macro variable. In the macro language, everything is stored text.
Here is the general syntax for %LET:

```
%LET macrovariablename = value ;
```

The name of a macro variable must conform to the customary rules for names in SAS.

Here is a particular example:

```
%LET OPT = N OBS UNIFORM;
```

Then, the statement

```
PROC PRINT &OPT;
```

would resolve as (that is, would be replaced by)

```
PROC PRINT N OBS UNIFORM;
```

Macros

Macros are used to manipulate SAS source statements. A macro is defined by enclosing its text between a %MACRO statement and a %MEND statement. The %MACRO statement may include a parameter list (either positional or keyword type) to define special macro variables that can be referenced within the macro.

Here is the general syntax which is most commonly used:

```
%MACRO macroname(parameters);
   . . . macrotext . . .
%MEND macroname;
```

Macro variables that are defined within a macro generally cannot be resolved outside of the macro.
It is possible to invoke a macro from within another macro. When this happens, the macros are said to be "nested." In general, a macro variable can be resolved in the macro within which it is defined, and in all other macros that are nested within that macro.

Macro Programming Statements and Macro Functions

Most macro programming statements appear similarly, and are used in the same way, as corresponding ordinary-SAS programming statements, except that they begin with a percent sign (%). For example, the %IF-%THEN-%ELSE, %DO, %DO %WHILE, %DO %UNTIL, %END, and %GOTO statements are used in precisely the same manner as the familiar IF-THEN-ELSE, DO, DO WHILE, DO UNTIL, END, and GOTO statements. These statements facilitate conditional code generation, iterative processing, and branching within a macro.

The SAS macro language includes several functions for manipulating strings that are similar to character string functions which might be used in a DATA step in ordinary SAS. %LENGTH, %SUBSTR, %INDEX, and %SCAN work just as you might suppose.

Single quotes (' ') and double quotes (" ") don't have the same effect in the SAS macro language as in ordinary SAS.
If a macro reference is placed within single quotes, it will not resolve (that is, it will be treated as constant text, and no symbolic substitution will occur).
However, macro variable references enclosed in double quotes will resolve.
The SAS language uses quotes to indicate character constants. However, since quotes are considered part of the text in the SAS macro language, quoting functions take the place of quotes in macro expressions.
Quoting functions are very useful for removing the customary syntactical meanings of special characters (like semicolons, unmatched parentheses, apostrophes, operators, and mnemonics) in character strings.
Consult the SAS documentation for further information.

A Few Examples of the Use of Macros

Here is an example of a macro, named REPET, which uses keyword-type parameters in an iterative loop to generate multiple DATA and PROC steps:

```
%MACRO REPET(FIRST=, LAST=);
   %LOCAL I;
   %DO I=&FIRST %TO &LAST;
      DATA NET&I;
         INFILE IN&I;
            INPUT DEPT $ INCOME EXPENSE;
            PROFIT = INCOME - EXPENSE;
      PROC PRINT;
  TITLE "REPORT OF INCOME & EXPENSE FOR &I";
   %END;
   RUN;
%MEND REPET;
```

Simply defining a macro does not cause it to execute. Here is how to invoke the previously-defined macro:

```
%REPET(FIRST=1990, LAST=2000)
```

Notice that the null default values of the parameters are replaced with desired values. Also notice that the statement does not end with a semicolon.

Following is an example in which macro variables are used for the start date of the time period under consideration in a batch reporting job which normally would be run each Monday morning. Before submitting the program, the appropriate date elements are inserted as arguments in the MDY function for a variable named START. If the program is run on a Monday, and if the desired time period is the preceding week, then it isn't necessary to do anything to START. This program is easy to maintain and use, since the start date is only entered once at the beginning of the program, and the macro facility automatically substitutes values for two different forms of the date as often as they are required.

```
DATA _NULL_;
   START = MDY(10,20,2000);
   %GLOBAL STRTDA1 STRTDA2;
    IF &SYSDAY=Monday THEN DO;
```

```
      START2 = &SYSDATE - 7;
      CALL SYMPUT("STRTDA1",
                  LEFT(PUT(START2,WORDDATE.)));
      CALL SYMPUT("STRTDA2",
                  LEFT(PUT(START2,DATE.)));
   END;
  ELSE DO;
     CALL SYMPUT("STRTDA1",
                 LEFT(PUT(START,WORDDATE.)));
     CALL SYMPUT("STRTDA2",
                 LEFT(PUT(START,DATE.)));
   END;

DATA A;
    INFILE IN1;
    INPUT  @1   VAR1        PD6.
           @7   VAR2          $2.
           @9   VAR3          $4.
           @13  DATEVAR  MMDDYY10.;
    IF VAR2 = '02;
    IF DATEVAR  GE "&STRTDA2"D;
PROC PRINT;
    VAR   VAR1  VAR3;
    TITLE "Records Processed Since &STRTDA1";
```

The preceding program makes use of two automatic macro variables, SYSDAY and SYSDATE, which are created when the SAS System is invoked. The SYMPUT routine is used to transfer information from the DATA step to macro variables, during the execution phase of the DATA step.

Sometimes, I use SAS to perform the same kind of statistical analysis and/or reporting on data files which share the same general structure, but which differ according to agency number, industry grouping, or economic region. Macros are ideal for this situation! Following is an outline of a typical case.

```
%MACRO ANLYS(REGN);
    DATA &REGN.2;
       SET &REGN;
       BY INDGRP;
       . . . (SAS programming statements)
    PROC . . . . ;
       . . . (SAS programming statements)
       TITLE "Analysis for &REGN Region";
    DATA . . . . ;
       . . . (SAS programming statements)
    PROC . . . . ;
    RUN;
%MEND ANLYS;
```

Notice the use of the parameter REGN. Macro parameters are special macro variables which are defined with the macro, and which are assigned a value when the macro is invoked. Also notice the use of a period as a macro variable delimiter in the first DATA step. Here is how the preceding macro would be called -- notice that each invocation includes a value for the parameter REGN:

```
    %ANLYS(PLAINS)
     %ANLYS(METROPL)
     %ANLYS(EASTEX)
     %ANLYS(GULFC)
     %ANLYS(CENTRAL)
     %ANLYS(BORDER)
     %ANLYS(OUTSTA)
```

Autocall Libraries

Macros can be stored in, and accessed from, macro libraries. The autocall facility is a way of making one or more libraries of commonly-used SAS macros available, without having to explicitly include them in each SAS program in which they are used. This allows SAS programmers to create macros for customized functions, and to have access to them whenever they are needed. There could be personal macro libraries, or departmental macro libraries for macros to be shared by many persons.

On MVS systems, autocall libraries are partitioned data sets. On a CMS system, an autocall library is a maclib. On a PC, an autocall library is a directory consisting of files with a 'sas' extension. The macros are stored as members in the library, where the name of the macro is the same as the name of the member. The libraries are concatenated, using the fileref SASAUTOS. Then, the autocall facility will automatically define and execute a stored macro whenever it is called in the SAS session, provided that the MAUTOSOURCE system option is in effect.

Suggestions for Further Reading

Art Carpenter, Carpenter's Complete Guide to the SAS Macro Language, Cary, NC: SAS Institute Inc., 1998.

SAS Institute Inc., SAS Guide to Macro Processing, Version 6, Second Edition, Cary, NC: SAS Institute Inc., 1990.

SAS Institute Inc., SAS Macro Facility Tips and Techniques, Version 6, First Edition, Cary, NC: SAS Institute Inc., 1994.

SAS Institute Inc., SAS Macro Language: Reference, First Edition, Cary, NC: SAS Institute Inc., 1997.

Thomas J. Winn Jr., "The SAS Macro Language: An Overview," Proceedings of SCSUG '91, pp. 267-272.

Thomas J. Winn Jr., "Introduction to the SAS Macro Language," Proceedings of SCSUG 2000, pp. 377-395.

Thomas J. Winn Jr., "Debugging SAS Macros," Proceedings of SCSUG '94, pp. 212-229; and SUGI-20 Conference Proceedings,1995, pp. 346-352.

Thomas J. Winn Jr., "Debugging SAS Macros, Revised," Proceedings of SCSUG '99, pp. 249-255.

Conclusion

The SAS Macro Language is a powerful tool for simplifying repetitive coding, for communicating information between program steps, for generating data-dependent SAS statements, for permitting conditional execution of SAS code, and for dynamically importing certain information from the SAS Supervisor. In macro programming, a modularized approach is used. In this presentation, we have seen, or referred to, a few of the basic uses. And we also have listed several references for further study.

Author Information

Tom Winn
Texas State Auditor's Office
P.O. Box 12067
Austin, TX 78711-2067

Telephone: 512 / 936-9735
E-Mail: twinn@sao.state.tx.us

Paper P809

A Beginner's Tour of a Project using SAS® Macros
Led by SAS-L's Macro Maven

Ronald Fehd, Centers for Disease Control and Prevention, Atlanta, GA, USA

ABSTRACT

SAS® Macros and the SAS®Macro Language are simple yet powerful tools. Once you understand how to use and write macros, your next task is to understand how to you use them consistently and more importantly, intelligently, from a programmer's point of view. This tutorial presents the basics of project design with an overview of macro usage across programs. Topics include communicating between programs, using system variables, use of the config.sas, autoexec.sas, %includes, etc. Expected audience are beginner and intermediate SAS programmers.

INTRODUCTION

This paper explains how I began using macros and have progressed over the years. I think it is important to illustrate the thought progression of beginning macro usage and progressing to more complicated and then back to more simple usage. The program examples illustrate the differing styles of using macros. Most have been generalized for this paper in order to highlight the concepts and the differences between user-written and data-supplied macro usage.

TABLE OF CONTENTS

8 D's OF SCIENTIFIC INVESTIGATION

Documentation:
Determine Goals
Study Design: data collection instrument
Data Collection: data dictionary
Data Entry
Data Management: SAS programs
Data Analysis: number-crunching, bean counting,
Disseminate Results: publish

Where do you work? Hopefully you're consulted on the design of the data collection instrument. Certainly you want to be familiar with the data dictionary, as it contains what you need to know to program: format definitions, variable type, length, and label.

PROGRAMMING CONSIDERATIONS

These programs are examples from a typical project. The main reason I learned to use macro variables was to be able to reuse programs developed in one project in the others. This exercise, in turn, lead me to consider the organizational programming issues of naming conventions and directory structure.

NAMING CONVENTIONS

I manage projects for two Research Groups:
* PEP: Performance Evaluation Project
* TB : Tuberculosis

Each of those groups has several projects
PEP: HIV: Human Immunodeficiency Virus
HTL: Human T-Lymphocyte Virus
TB: NAA: Nucleic Acid Amplification
NTM: Non-Tuberculosis Mycobacteria

These projects have biannual shipments, e.g.:
NAA-2000-Jan NAA-2000-Aug

Over the years I developed these data set naming conventions:

```
col name range explanation
1   ID:  P:    Panel
         S:    Sample
2:5 yymm       Year+Month: 2000-Jan
6   data A:Z   Study: A=Any-Other, E=EIA, ...
7   DT   D,T   data, text
8   ver  0:3   version: 0=final, 1=master,
                2=clean, 3=with corrections

example:
P0001AD0  Panel data for Any-Other
P0001AD1  \
P0001AD2   > versions one thru three
P0001AD3  /
P0001AT0  text data set
S0001AD0  Samples, transposed from Panel
```

See the macro TITLES below and the global macro variable DATA_SET which implements this data set naming convention.

DIRECTORY STRUCTURE

A directory for an individual shipment includes these sub-directories under MS-Windows:

```
c:\...\NAA\p0008 parent directory
..\doc  data collection form, data dictionary
..\htm  HyperText Mark-Up Language from SAS ODS
..\pgm  SAS programs
..\ssd  SAS data sets
..\xls  Excel, written from summary data sets
```

Note that each directory contains a specific set of file types. See the use of the macro-variable PATH below in pointing to each directory.

DATA DICTIONARY

Early in my career I depended on a print-out of proc CONTENTS as my data dictionary. Developing a good workable data dictionary is very important. Getting the people whose data I manage to understand how important it is to them and that it should therefore be, not only their priority but also their responsibility occupied me in many meetings over several years. There are two very important items in a good data dictionary: format definitions and data definitions.

Here is an abbreviated example from one of my projects:

```
Data Dictionary: PEP HIV-1 Results Panel 2001-01
CDC: programmer/analyst:    Ronald Fehd
Shipment ID  : 9
Shipment date: January 29, 2001

Contents:
I.   Sample Identification look-up table
I.   Deliverables Description
II.  Editing Instructions
III. Abbreviations
IV.  Formats
V.   Variable Descriptions
. . .
IV.  Formats

value HIV9SN /*Sample-Nmbr to Sample-Name */
.,0 = "&BLANK."
 1 = '9-1'
 ...[snip]
 5 = '9-5'
other = "&INVALID.";
. . .
V.  Variable Descriptions
          Char/
          Num
Variable  :len  Format    Label/Description
--------  ----  -------   -----------------
IDNmbr    C:3   $char3.   ID range:001--999
SmplNmbr  N:4   HIV9SN.   Sample Number
```

Note that the format name HIV9SN has a prefix which is the project identifier: HIV and an infix of the shipment ID: 9. I use this convention to separate the format definitions that are unique to a shipment. This is helpful when concatenating many shipments.

Refer to the program Autoexec-Basic, lines 11 and 12, and note the global macro variables BLANK and INVALID. Note that when the macro variables are referenced in the data dictionary and programs that they have a dot as suffix delimiter.

The use of BLANK and INVALID in format descriptions is a key feature to understand when reviewing the FREQ program examples because they are used to exclude those values from reports.

A Venn diagram of the values in any variable would contain first the set of valid values indicating no answer was provided. Next would be the set of expected and therefore valid values, which have been correctly and completely defined in the formats sections of the data dictionary. The remainder is the set of invalid values. What do you do with these meaningless values? Include and explain? Or exclude and reduce the number of observations reported as responding?

BLANK: labeling dot=missing and zero as BLANK is a convenient way of identifying acceptable values outside of the expected range that are valid. For a character format the statement identifying space and dot would be: ' ','.' = "&BLANK."

INVALID: labeling unspecified values and ranges with the value statement option <other = "&INVALID."> is the prerequisite to identifying and excluding these values both in data review and in preparing summary data sets. See the FREQ programs for usage.

NOTES ON A STYLE SHEET

These programming examples illustrate and expand Fehd (2000) *Writing for Reading SAS® Style Sheet* (W4R) which addressed conventions used to differentiate SAS statements from macro statements. Here is a short summary:

;/*	precede a slash-asterisk block at column one with a semicolon to avoid problems when copying to mainframe
MACRO	statements in UPPER CASE
sas®	statement in lower case
Variable	names in Mixed Case
%THEN	always use %DO;+%END; block to avoid confusing macro semicolon with SAS semicolon
&J.	use dot as suffix delimiter for macro variables
"&J."	always quote macro variables in conditions to avoid testing a macro variable with the value of the logical comparison operator 'OR', the two-letter state abbreviation for Oregon e.g.: %IF "&STATE." eq "OR" %THEN
dot	use as suffix delimiter in macro variables
dot-dot	infix in two-step data set names &LIBRARY..DATANAME infix in filename.ext: &FILENAME..SAS suffix for formats: $char&WIDTH..
DATA	placement of data step statements on page left: information, unconditionally executed: attrib array drop format keep, etc. control statements: if, do, middle: conditionally executed right: closure: end
proc	use hanging indent center: keywords right: user-supplied options or parameters

MACRO VARIABLE USAGE

There are two ways of creating and initializing macro variables: the first is a user-defined macro variable and the second is using data to supply a value.

The simple and direct way to create a macro variable is:
%LET MACVAR = blank for now;

Note that the keyword and name are in upper case; this is a reminder that this statement operates in the global programming environment, the same as title, footnote, and option statements.

In addition, it is important to realize that every macro variable is a character string, even when the value contains only digits and may conform to the appearance of a real number:
%LET PI = 3.1416; %LET SUM = &PI. + 1;%PUT SUM<&SUM>;
yields this note in the log: SUM<3.1416 + 1>;
%LET SUM =%eval(&PI. + 1);%PUT SUM<&SUM>;
we're getting closer: SUM<4>; because the %eval function does only integer arithmetic!

Remember: macro variable values are always strings!

The indirect way to create a macro variable from data is to use a pair of statements:
1. call symput('MACVAR',<character expression>);
2. RUN;

The RUN statement is a step boundary, and only after a step boundary is the macro variable available to the next step. See usage in program INOBS, lines 8 and 9, below.

THREE STEPS IN WRITING MACROS

1. Recognize a pattern: In developing macros I notice when I have written two similar SAS paragraphs: two similar sets of statements. Once I have two different paragraphs working correctly, then I begin to review and identify the commonalities and differences, which are the parameters of the first version of a potentially reusable program.

2. Prepare a parameterized %include file. I use global macro variables for the parameters. The major benefit of this intermediate step, especially for beginners, is that you still have line numbers available in your SAS log. Those line numbers are so valuable in debugging because they tell you where your SAS errors are located. This point is important! Line numbers are no longer available from within macros!

3. Define a macro and convert the list of global macro variables used to macro parameters. Kiss the line numbers in the log goodby; now you have to guess where the errors are occurring: somewhere inside your macro! These steps are illustrated in the examples below .

Project Set-up: three examples of autoexec.sas

```
   12345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012
01 ;/* AUTOEXEC  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */ 01
02 libname LIBRARY 'C:\SAS\PEP\HIV\P200101\SSD';                                               02
03 TITLE1 'HIV-1 shipment A: 2001-Jan-31';                                                     03
```

This is as simple as it gets: where is the data stored and what is the main title to be used in reports.

```
01 ;/* AUTOEXEC basic  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */ 01
02 %LET PROJECT  = HIV;          %*(HIV,HTL,NAA,NTM): directory PATH-name ;                    02
03 %LET PROJNAME = HIV-1;        %*PROJECT expanded: TITLEs               ;                    03
04 %LET SHIP_ID  = A;            %*(0:9,A:Z)       : infix in format names;                   04
05 %LET SHIPYYMM = 200101;       %*ccYYMM          : names: data set, PATH;                   05
06 %LET SHIPDATE = Jan 29, 2001; %*Mon dd, ccYY    : report TITLEs        ;                   06
07 ;/* AUTOEXEC begin execution - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/ 07
08 TITLE "&PROJNAME. Shipment: &SHIP_ID: &SHIPYYMM. &SHIPDATE.";                              08
09 %LET LIBRARY  = LIBRARY;      %*default library;                                           09
10 %LET DATA_SET = A&SHIPYYMM.D0;%*default data set, reset by macro TITLES;                    10
11 %LET BLANK    = BLANK;        %*numeric: missing and character: blank;                     11
12 %LET INVALID  = INVALID;      %*out of expected range: see formats;                        12
13 %LET PATH     = C:\SAS\PEP\&PROJECT.\P&SHIPYYMM.\;                                         13
14 %LET PATH_HTM = &PATH.HTM;    %*HyperText Markup;                                          14
15 %PUT _USER_;                  %*checking;                                                  15
16                                                                                            16
17 libname LIBRARY "&PATH.SSD";                                                              17
18 %INCLUDE SASAUTOS(Titles);    %*load fan-in macro(s);                                      18
19 options details noCenter source2;                                                        19
20                                                                                            20
21 run;%* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . AUTOEXEC end;          21
```

Now we're using macro variables, perhaps a little too much!

This is a basic autoexec with global macro variables. It is useful when programming with parameterized %include files.

Refer to the data dictionary and note the project description macro variables defined in lines 02:06 are used in the title in line 08. Lines 11:12 are first used in a proc FORMAT program, – which would be cut and pasted from the data dictionary example above – later in data review, and finally in excluding observations before creating summary data sets. PATH is defined in line 13 and concatenated in references in line 14 and 17. Any other directory could be pointed to using this style; see reference to PATH_HTM in the FREQ1VAR ODS example and PATH_XLS in the macro SAS2XLS. The statement %PUT USER in line 15 shows the user-defined macro variables in the session. The option source2 in line 19 shows all lines from %included files, and is an essential aid in debugging.

```
01 ;/* AUTOEXEC intermediate   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */ 01
02 %LET PROJECT  = HIV;          %*(HIV,HTL,TB)    : directory PATH-name;                     02
03 %LET PROJNAME = HIV-1;        %*PROJECT expanded: TITLEs;                                  03
04 %LET SHIP_ID  = A;            %*(0:9,A:Z)       : infix in format names;                   04
05 %LET SHIPYYMM = 200101;       %*ccYYMM          : names: data set, PATH;                   05
06 %LET SHIPDATE = Jan. 24, 2001;%*Mon dd, ccYY    : report TITLEs;                           06
07 /* AUTOEXEC begin execution - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */ 07
08 TITLE "&PROJNAME.: Shipment: &SHIP_ID: &SHIPYYMM. &SHIPDATE.";                            08
09                                                                                          09
10 %LET PATH     = C:\SAS\PEP\&PROJECT.\P&SHIPYYMM.\;                                       10
11 %LET LIBRARY  = LIBRARY;     ;%*default libname;                                         11
12 %LET DATA_SET = A&SHIPYYMM.D0;%*default data set, reset by macro TITLES;                  12
13 %PUT _USER_;                  %*checking;                                                13
14                                                                                          14
15 filename SASAUTOS ("%sysfunc(getoption(SASUSER))"                                       15
16                   );%*add program directory to search-path for macros;                  16
17 filename SASAUTOS list;       %*checking;                                               17
18                                                                                          18
19 %TITLES();                    %*checking;                                               19
20                                                                                          20
21 libname LIBRARY "&PATH.SSD";                                                            21
22                                                                                          22
23 options details noCenter MautoSource;                                                  23
24                                                                                          24
25 dm log 'awsmaximize on;log;zoom on;up max;down max' log;%*view log;                     25
26                                                                                          26
27 run;%*. . . . . . . . . . . . . . . . . . . . . . . . .  AUTOEXEC end;                 27
```

In this intermediate autoexec global macro variables have been moved to small files. See LetFrmt and LetPath below.

General purpose macros are stored in a separate file with the name of the macro as the file name, e.g.: the macro TITLES is stored in file TITLES.SAS. In line 15 the fileref SASAUTOS points to the program directory: SASUSER, which is defined in your SAS invocation. A review of your proc OPTIONS will show sasautos=sasautos. The fileref SASAUTOS is the autocall library, that is it contains macros within same-named files; these macros are called autocall macros. The option MautoSource enables searching of the SASAUTOS fileref for the autocall macros which are then %included. See the macros Nobs and Freq1Var below.

```
01 ;/*letFrmt %GLOBAL mac-vars used in formats, exception reports, summary*/01
02 %LET BLANK = BLANK;          %LET INVALID = INVALID;                     02

01 ;/*letPath %GLOBAL mac-vars used in writing reports to other files */    01
02 %LET PATH_HTM = &PATH.HTM;   %LET PATH_XLS = &PATH.XLS;                  02
```

Lest we forget: having identified BLANK, INVALID, and PATH_HTM as global macro variables used in few programs, remove them to their own files and%include them only when needed.

See usage in the Freq programs.

```
      +    1    +    2    +    3    +    4    +    5    +    6    +    7
   12345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012
01 ;/*TITLES Basic . . . . . . . . . . . . . . . . . . . . . . . . . . . */ 01
02 %MACRO TITLES(PFX  /*PreFiX                                            */ 02
03              ,NFX  /* InFiX                                            */ 03
04              ,SFX=0/*SufFiX in 0:4                                      */ 04
05              );                                                           05
07 %GLOBAL DATA_SET TITLEN;                                                 07
08 %LET TITLEN   = 2;                                                       08
10 %LET DATA_SET = &PFX.&SHIPYYMM.&NFX.&SFX.;             %*concatenation; 10
11                                                                          11
12 TITLE&TITLEN.                                                           12
13 %IF       "&PFX."  = "P" %THEN %DO; 'Panel '                    %END; 13
14 %ELSE %IF "&PFX."  = "S" %THEN %DO; 'Samples '                          14
15  %IF       "&NFX." = "A" %THEN %DO; 'Abbot'                     %END; 15
16  %ELSE %IF "&NFX." = "I" %THEN %DO; 'In-house'                  %END; 16
17  %ELSE                          %DO; "invalid NFX &NFX."        %END; 17
18                                                  %*IF PFX=S; %END; 18
19 %ELSE                          %DO; "invalid PFX &PFX."        %END; 19
20                                                    %*titleN end;; 20
21 %put @@TITLES: DATA_SET<&DATA_SET>;                                     21
22 footnote;options pageno=1; %* . . . . . . . . . . . . .  *TITLES; %MEND; 22
23 ;/************************** to enable end this line with a slash (/) * 23
24 %TITLES(Z);%*see error msg in TITLE2;                                   24
25 %TITLES(P);                                                             25
26 %TITLES(S,A);                                                          26
27 %TITLES(S,I);                                                          27
28 DATA _NULL_;FILE PRINT;PUT 'XXXXXX';STOP;RUN;                          28
29 ;/*************************************/                               29
```

The TITLES macro is a fan-in routine: it is called by many other programs. It serves two main purposes:

1. returns %global macro variable DATA_SET, declared in line 07 and set in line 10; it is concatenated from its various parts according to either your site's or the individual project's data set naming conventions, discussed above.

2. sets global environment variable title2. Note carefully that the statement begins on line 12 and ends with the second semicolon on line 20 in column 72. Note also that the macro control statements %IF and %ELSE in lines 13:19 do not generate any SAS semicolons; their only result is string literals for the title statement.

The secondary effects – line 22 – are footnotes are cleared and page number is reset to one.

Further titles in a program may be placed after calling the TITLES macro using this example statement:

```
Title%eval(&TITLEN.+1) 'More information';
```

After using this convention, extra titles throughout the entire project may be adjusted by adding title2 to the autoexec and changing the value of TITLEN to 3 in line 08.

W4R: Note the macro control statements in lines 13:20:
1. the one-character indent showing the NFX tests are subordinate to the PFX=S test.
2. alignment of all %THEN %DO in columns 26:35
3. columns 37:54 optional literals for TITLE statement
4. columns 67:72

closure of %DO block:	%END;
closure of TITLE:	semicolon
closure of macro:	%MEND;

When copying this macro to another project to use, it is easy to modify because the values to change – PFX, NFX, and title literals are aligned vertically.

NOBS: number of observations of a data set, two utilities used in the FREQ examples:

```
      +    1    +    2    +    3    +    4    +    5    +    6    +    7
   12345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012
01 ;/* iNobs: Include Nobs                                                                    01
02 parameters: %GLOBAL LIBRARY DATA_SET                                                      02
03 from: SAS Guide to Macro Processing, V6, 2nd ed., pg 263                                  03
04 note: call symput + RUN; NOBS has no value until loaded at step boundary                  04
05 returns %GLOBAL macro-var NOBS: number of observations;/* . . . . . */                   05
06 DATA _NULL_;                                                                              06
07 if 0 then set &LIBRARY..&DATA_SET. nobs = Count;                                          07
08 call symput('NOBS',compress(put(Count,32.)));                                 stop;       08
09 run;%PUT @@iNobs: &DATA_SET. has <&NOBS.> observations;                                   09
```

Remember what I said about using the pair of statements: call symput+run?. Examine this program closely and understand not only what but also when NOBS is happening. As a test insert this line between line 08 and 09:

```
%put NOBS<&NOBS.>;
```

You'll receive this message:
WARNING: Apparent symbolic reference NOBS not resolved.

because NOBS is uninitialized until the second statement of the pair in line 09.

Note snake eyes – double dots – in line 07. Remember that macro variables are delimited with a dot as suffix; the second dot is the infix in a two-level data set name.

```
01 ;/*macro NOBS - - - - - - - - - - - - - - - - - - - - - - - -  1999Jun03 01
02 NOBS returns macro-var with number of observations of data set            02
03 from SAS Macro Language Reference, 1e, pg 242                             03
04 note: test data shows calling macro must have RUN;                        04
05       and declare _MAC_VAR %local                                         05
06 note: compare arguments to attrn: NOBS, NLOBS, NLOBSF /*  . . . . . */   06
07 %MACRO NOBS(_MAC_VAR  /* macro-var name              default=NOBS     */ 07
08            ,DATA    =./* data set name               default=&SYSLAST.*/ 08
09            ,_GLOBAL_=0/* return %GLOBAL mac-var?     default=%LOCAL   */ 09
10            );%LOCAL DSN;                                          run; 10
11 %IF "&_MAC_VAR" = ""  %THEN      %LET _MAC_VAR = NOBS;                 11
12 %IF  &_GLOBAL_        %THEN %DO; %GLOBAL &_MAC_VAR.;               %END; 12
13 %IF "&DATA."    = "." %THEN      %LET DSN = &SYSLAST.;%*resolve mac-var; 13
14 %ELSE                            %LET DSN = &DATA.;                    14
15 %LET DSID = %sysfunc(open(&DSN.));                                     15
16 %IF &DSID %THEN %DO;    %LET &_MAC_VAR. = %sysfunc(attrn(&DSID.,NLOBS)); 16
17                         %LET RC         = %sysfunc(close(&DSID.)); %END; 17
18 %ELSE                    %PUT Open for &DATA. failed:%sysfunc(sysmsg());  18
19 %PUT NOBS: "&_MAC_VAR."=<&&&_MAC_VAR.> data=&DSN.; %* . . .*NOBS; %MEND; 19
20 ;/************************** to enable end this line with a slash (/) * 20
21 data X;do I = 1 to 10;output;stop;run;                                   21
22 %NOBS();%PUT _USER_;%PUT NOBS=<&NOBS.>;                                 22
23 %MACRO TESTING(DATA);        DATA &DATA.;do I = 1 to 12;output;stop;run; 23
24 %local NOBS_Y;                                                          24
25 %NOBS(NOBS_Y);RUN;%PUT _LOCAL_;%*see DATA and NOBS_Y;                   25
26 %MEND;                                                                  26
27 %TESTING(DATA1);                                                        27
28 %PUT NOBS_Y=<&NOBS_Y.>;%*see error message: NOBS_Y does not exist;      28
29 ;/************************************************TEST DATA closure*/ 29
```

The macro NOBS differs essentially from iNOBS in that it uses the set of system functions and does not end with a run statement. Since this is a fan-in macro, the calling macros declare a local macro variable in their macro symbol table, that is: in their scope. Each calling macro has a run statement after the call to NOBS, which allocates the macro variable within its scope. Examine the test data provided and note the call in FREQ1VAR, line 16.

W4R: NOBS is useful in showing the style sheet in use: note how easy it is to scan the control statements on the left – LINES 11:18 --, then the conditionally executed statements in the center; and finally all closures on the right side, where they are not distracting.

```
   12345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012
01 ;/* FREQ program */                                                                          01
02                                                                                             02
03 proc FREQ data = LIBRARY.P200101AD0;                                                        03
04           format Var_A Fmt_A.;                                                              04
05           tables Var_A;                                                                     05
06           TITLE2 'FREQ of p2001-Jan VAR_A LabType';                                         06
07                                                                                             07
08 proc FREQ data = LIBRARY.P200101AD0;                                                        08
09           format Var_B Fmt_B.;                                                              09
10           tables Var_B;                                                                     10
11           TITLE2 'FREQ of p2001-Jan VAR_B NumSpecimens';                                    11
12 RUN;                                                                                        12
```

Review this first program example and notice the two similar paragraphs. SAS-L's Ian Whitlock refers to this type of program as wallpaper: lots of repetition with little variation. Recognize the same library, same data set, same identifier, different variables, different formats and different titles.

W4R: Note the hanging indent of the options after the proc PRINT. Columns 11:16 contain the control statements for the procedure and its options. User-supplied information is placed in columns 18:72. This placement on the page is the key of the W4R style sheet: vertical alignment of keywords enables quick scanning to locate phrases or statements to examine and change.

With two examples in view, what is a template into which you can insert the common elements?

```
01 ;/* iFreqPrnt include FREQ Print output w/title                          01
02 parameters: LIBRARY : libname in (LIBRARY,WORK)                         02
03             DATA_SET: set in TITLES                                     03
04             VAR     : variable                                          04
05             FORMAT  : format name including dot as suffix;/* . . . .  */ 05
06 %INCLUDE SASAUTOS(INOBS);                                               06
07                                                                         07
08 proc FREQ data = &LIBRARY..&DATA_SET.;                                  08
09           format &VAR. &FORMAT.;                                        09
10           tables &VAR.;                                                 10
11           TITLE2 "FREQ of &DATA_SET. obs:&NOBS. &VAR format: &FORMAT."; 11
```

This template is derived from the FREQ program above. Lines 08:11 are the essential statements from lines 03:06 and 08:11 in the FREQ program. Note the documentation in lines 02:05; that is an important reminder of what you are doing.

Next we need a driver: a calling program to execute this as an %included file.

```
01 ;/* call FREQ Basic demo program */                                     01
02 %TITLES(P,A);             %*returns %GLOBAL DATA_SET;                  02
03                                                                         03
04 %LET VAR    = Var_A;                                                    04
05 %LET FORMAT = Fmt_A.;                                                   05
06                           %INCLUDE SASAUTOS(IFREQPRNT);                 06
07 %LET VAR    = Var_B;                                                    07
08 %LET FORMAT = Fmt_B.;                                                   08
09                           %INCLUDE SASAUTOS(IFREQPRNT);                 09
10 RUN;                                                                    10
```

Call FREQ Basic is the driver that calls the simple FREQ program iFreqPrnt.

Review Autoexec Basic, line 09 to see the global macro variable LIBRARY. Review line 19, and see the option source2; this option enables the log to show line numbers of the %included files which is very important for debugging your statements!

Note the TITLES macro – called in line 02 – defines the global macro variable DATA_SET.

W4R: Control statements on the left:
1. TITLES which sets DATA_SET
2. VAR and FORMAT declarations.
Conditionally-executed %include statements in the center. When copying to a new project, the items to change – parameters to TITLES and VAR and FORMAT declarations – are aligned.

```
01 ;/* iFreqSmry Include FREQ Summary                                      01
02 parameters: LIBRARY : LIBRARY, WORK                                     02
03                       note output data set written to same LIBRARY      03
04                       note snake-eyes in &LIBRARY..&DATA_SET            04
05                            &LIBRARY. has suffix delimiter of dot        05
06                            + second dot is two-level name delimiter     06
07             DATA_SET: set in TITLES                                     07
08             VAR     : variable                                          08
09             FORMAT  : format name including dot as suffix               09
10             BLANK+INVALID: see LETFRMT,                                 10
11 %INCLUDE SASAUTOS(LETFRMT);%*calling program must load fan-in module;   11
12 output object data set structure:  Label Value Count Percent       /*.*/ 12
13 %INCLUDE SASAUTOS(INOBS);                                               13
14                                                                         14
15 proc FREQ data   = &LIBRARY..&DATA_SET.                                 15
```

```
16           (where = (put(&VAR.,&FORMAT.)                                  16
17                     not in ("&BLANK.","&INVALID.")));                    17
18           format   &VAR. &FORMAT.;                                       18
19           tables   &VAR.                                                 19
20           /        noprint                                               20
21           out    = FREQ1VAR;                                             21
22                                                                          22
23 DATA    &LIBRARY..&VAR.       (label  = "N=&NOBS. &LIBRARY..&DATA_SET"   23
24                                 rename = (&VAR. = Value));               24
25  attrib Label length = $ 40                                              25
26         Count length =    4    label = "N=&NOBS.";                      26
27  retain N_Obs &NOBS.;                                                   27
28  set    FREQ1VAR;                                                       28
29  Label = put(&VAR.,&FORMAT.);                                           29
```

Change happens! And new requirements specifications need to be implemented. IFreqPrnt was a print routine; module iFreqSmry saves a summary data set with information in a standard data set structure – a summary object.

Note that the NOBS macro variables is the number of observations of the whole data set and that the where clause may exclude observations that are blank or invalid. The result could be that the sum of Count might not equal the number of observations.

W4R: Note the where clause in lines 16:17, which is a data step option; it is placed in the central control column because it is a keyword. Likewise with the slash and out= options of the tables statement in lines 19:21, keywords in the center, user-supplied optional tokens to the right.

```
01 ;/* call FREQ Intermediate demo program */                              01
02 %LET WHICH = PRNT;                                                      02
03 %LET WHICH = SMRY;                                                      03
04 %INCLUDE SASAUTOS(LETFRMT);  %*mac-vars used by IFREQSMRY;              04
05                                                                         05
06 %TITLES(P,A);                %*returns %GLOBAL DATA_SET;               06
07                                                                         07
08 %LET VAR    = VAR_A;                                                   08
09 %LET FORMAT = LabType.;                                                 09
10                              %INCLUDE SASAUTOS(IFREQ&WHICH.);           10
11 %LET VAR    = VAR_B;                                                   11
12 %LET FORMAT = NumSpec.;                                                12
13                              %INCLUDE SASAUTOS(IFREQ&WHICH.);           13
14 RUN;                                                                    14
```

Change has happened! In this driver, we want to be able to switch between the old program iFreqPrnt, which produce only a print, and the new iFreqSmry which produces a summary data set.

Conditional execution of %include files: To enable the use of the print routine, disable line 03 with an asterisk:
*LET WHICH = SMRY;
This leaves the macro variable WHICH with the value of PRNT, which yields an include filename of (IFREQPRNT).

```
01 %MACRO SAS2XLS(DSN                                                      01
02               ,LIBRARY = LIBRARY                                        02
03               ,OUT     = .                                              03
04               ,PATH    = &PATH_XLS.);                                   04
05 %IF "&OUT." = "." %THEN %LET OUT = &DSN.;                               05
06 filename           FILE2DEL  "&PATH.\&OUT..XLS";                       06
07 data      _NULL_;                                                       07
08  Rc     = fdelete('FILE2DEL');                                          08
09  SysMsg = sysmsg();                                                     09
10  put Rc = SysMsg =;                                                     10
11 PROC DBLOAD dbms   = EXCEL                                              11
12             data   = &LIBRARY..&DSN.;                                   12
13             path   = "&PATH.\&OUT..XLS";                                13
14             putnames yes;                                               14
15             limit  = 0;                                                 15
16             load;                                                       16
17 filename    FILE2DEL clear;                                             17
18 run;%* . . . . . . . . . . . . . . . . . . . . . . . . . . . .; %MEND; 18
19 %*SAS2XLS(TBL8F);                                                       19
```

SAS2XLS is a short fan-in utility macro. Note in line 04 that PATH defaults to a global macro variable PATH_XLS, which is defined in a the LETPATH include file. This routine will not overwrite an existing file; lines 09:11delete a previous file.

Converting %includes to macros:
Ok, so you've been writing for two years or have written ten thousand statements, whichever milestone came first for you. You are comfortable writing flawless paragraphs, even chapters, of SAS statements. You figure you're ready to lose the line numbers, write macro statements in ALL CAPS, in order to remind yourself that you are switching gears between the number-crunching paradigm of SAS and the really simple string-handling of the macro language. Let's look at a conversion:

```
01 ;/* macro FREQ1VAR FREQ of One Variable                                    01
02                              NOTE: need %GLOBAL mac-vars: BLANK INVALID 02
03 usage:                                                                    03
04 %FREQ1VAR(VAR1,FORMAT1);                                                 04
05 %FREQ1VAR(VAR2,FORMAT2,DATA=P0001AD3);                                    05
06 %FREQ1VAR(VAR3,FORMAT3,LIBRARY=WORK);                                     06
07 output data set structure: N_Obs Label Value* Count Percent;/*. . . . */ 07
08 %macro FREQ1VAR(VAR                  /* variable name                */ 08
09                ,FORMAT               /* format with dot as suffix    */ 09
10                ,LIBRARY  = LIBRARY   /* or LIBRARY=WORK              */ 10
11                ,DATA     = &DATA_SET./* NOTE: %GLOBAL mac-var        */ 11
12                ,OUT      = .         /* default is &VAR              */ 12
13                ,HTM_PATH = &PATH_HTM./* default to %GLOBAL mac-var    */ 13
14                );%local NOBS;                                            14
15 %IF "&OUT." eq "." %THEN %LET OUT = &VARIABLE.;                          15
16 %NOBS(data=&LIBRARY..&DATA.);run;                                        16
17                                                                          17
18 proc FREQ data   = &LIBRARY..&DATA.                                      18
19           (where = (put(&VAR.,&FORMAT.)                                  19
20                     not in ("&BLANK.","&INVALID."));                     20
21           format   &VAR. &FORMAT.;                                       21
22           tables   &VAR.                                                 22
23           /        noprint                                               23
24           out    = FREQ1VAR;                                             24
25                                                                          25
26 DATA    &LIBRARY..&OUT.(label  = "&LIBRARY..&OUT."                       26
27                        rename = (&VAR. =                                 27
28         %IF "%substr(&FORMAT.,1,1)" eq "$" %THEN %DO; ValueChr     %END; 28
29         %ELSE                                     %DO; ValueNum     %END; 29
30                                                   %*   DATA closure;  )); 30
31  attrib N_Obs length =  4                                                31
32         Label length =$40                                                32
33         Count length =  4 label = "N=&NOBS.";                            33
34  retain N_Obs &NOBS.;                                                    34
35  set    FREQ1VAR;                                                        35
36  Label = put(&VAR.,&FORMAT.);                                            36
37                                                                          37
38 %SAS2XLS(&OUT.);                                                        38
39                                                                          39
40 ods  listing close;                                    %*SAS.list OFF; 40
41 ods  html    body = "&HTM_PATH.\&OUT..HTM";                              41
42 proc PRINT   data = &LIBRARY..&OUT.;                                     42
43 ods  html    close;                                                      43
44 ods  listing;                                         %*SAS.list  ON; 44
45 run;%* . . . . . . . . . . . . . . . . . . . . . . . . .FREQ1VAR; %MEND; 45
```

The FREQ1VAR macro illustrates several conversion issues:
1. Parameter list: VAR and FORMAT are positional parameters because they follow the order of macro variable declarations in the previous calling programs.
Named parameters: LIBRARY is hard coded to default to the name LIBRARY, but DATA defaults to the global macro variable DATA_SET, which is set by the fan-in macro TITLES.

Avoid this error: setting a local variable equal to a global variable of the same name: DATA_SET = &DATA_SET. This generates a circular reference, which leaves your session in an infinite loop!

2. local macro variables: The macro variable NOBS was global when generated by the include file iNOBS. In line 13 it is declared local in order to reduce the number of global macro variables. Note the run statement after the call of the macro NOBS; this step boundary in FREQ1VAR enables macro variable NOBS to be local to FREQ1VAR. If the step boundary were in the macro NOBS, the macro variable NOBS would be in the global symbol table.

Similarly OUT has a default value of blank – line 12 – but is immediately tested and reset to the value of VAR when blank. Note the quotes around the values being compared. This is a mnemonic that reminds you the macro language is a string-processing language. Why use a default value of dot? Because you can see it! You want to avoid this test:
%IF &OUT = %THEN ...
This expands to: %IF &OUT. eq blank %THEN ...
I recommend: %IF "&OUT." eq "dot" %THEN ... because it's more readable.

3. %THEN %DO lines 28:29 ValueChr and ValueNum are the new names of the FREQ tables variable. This construct shows the macro language inserting the appropriate token in the data statement rename phrase which starts on line 27 and ends on line 30. Without the %DO ... %END bracket, there would be a semicolon after each of the two tokens; these are the closure of the %IF and %ELSE macro statements, not to be confused as closure of the data statement.

4. use of global macro variables as default values of parameters: DATA and PATH_HTM. Note that the macro SAS2XLS uses the global macro variable PATH_XLS.

This macro is a shortened version of Fehd (1999) %FREQ1VAR.

```
01 ;/* call FREQ macro demo program */                                    01
02 %INCLUDE SASAUTOS(LETFRMT);  %*mac-vars used by IFREQSMRY;             02
03                                                                        03
04 %TITLES(P,A);                %*returns %GLOBAL DATA_SET;              04
05                                                                        05
06 %FREQ1VA4(Var_A,Fmt_A.);                                              06
07 %FREQ1VA4(Var_B,Fmt_B.);                                              07
08 RUN;                                                                  08
```

Change! Notice that the calling programs are getting shorter. And now, let us have SAS eliminate our typing of variable names and formats.

```
01 ;/* call FREQ macro demo program */                                    01
02 %INCLUDE SASAUTOS(LETFRMT);                                           02
03                                                                        03
04 %TITLES(P,A);                %*returns %GLOBAL DATA_SET;              04
05                                                                        05
06 proc CONTENTS data  = &LIBRARY..&DATA_SET                             06
07               out   = CONTENTS                                        07
08               (keep = Name Format);                                   08
09                                                                        09
10 filename TEMPTEXT catalog 'sasuser.profile.sasinp.source';            10
11                                                                        11
12 data _NULL_;                                                          12
13  Set CONTENTS;                                                        13
14  put '%FREQ1VAR(' Name ',' trim(Format) ');';                         14
15 run;                                                                  15
16 %include TEMPTEXT /source2;                                          16
17 run;                                                                  17
18 file     TEMPTEXT clear;run;                                         18
```

Where is a list of variable names and formats? In a correctly-written data structure, which was thoroughly and complete described in a data dictionary. Proc CONTENTS provides an object with variable name and format, which can be used to write a series of macro calls as in Call-FREQ-macro, above.

CONCLUSION

There are several important issues to consider when using macros:
* Maintenance: what is the skill level of the maintenance programmer?
* Global macro variable usage: where do changes occur? As an example, changing the value of the macro variable LIBRARY from LIBRARY to WORK, would cause all subsequent programs to abend with a 'data set not found' error message.
* Ensuring local macro variable usage
* Programming style: ad hoc, planned, or revised and revisited?

Here ends the tour of a decade-long process of converting programs with many repetitive paragraphs, to the simpler style of using %ncludes. The simple modules developed are reusable. The %nclude style is expensive in its use of global macro variables.

Moving from %ncludes to macros is expensive in terms of development because of the moderate learning curve for the macro language. In the long run, macro usage is simpler, and less expensive in terms of global macro variable usage. In addition new project development is quicker because of reusable modules and the minimal changes necessary to customize copied programs.

REFERENCES

Fehd, Ronald (1999),"%FREQ1VAR: Frequency of one variable with format: a macro to standardize proc FREQ output data sets," *Proceedings of the Twenty-Fourth Annual SAS Users Group International,* 24: paper 234, pg 1373.

Fehd, Ronald (2000), "The Writing for Reading SAS® Style Sheet: Tricks, Traps & Tips from SAS-L's Macro Maven," *Proceedings of the Twenty-Fifth Annual SAS Users Group International,* 25: paper 38-25, pg 217.

SAS Institute Inc. (1990), *SAS® Guide to Macro Processing, Version 6, Second Edition,* Cary, NC: SAS Institute Inc.

SAS Institute Inc. (1997), *SAS® Macro Language Reference, First Edition,* Cary, NC: SAS Institute Inc.

ACKNOWLEDGMENTS

I thank my colleague and comrade on SAS-L, Dianne L. Rhodes for her critique and encouragement. My thanks to my greatest asset: SAS-L contributors who regularly contribute good program ideas.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Author Name	Ronald Fehd
Company	Centers for Disease Control MS:G25
Address	4770 Buford Hwy NE
City, State ZIP	Atlanta GA 30341-3724
Work Phone:	770/488-8102
Fax:	770/488-8282
Email:	RJF2@cdc.gov

Are Strings Tying You in Knots?

Deb Cassidy
Cardinal Distribution, Dublin, OH

ABSTRACT

If the answer to the title questions is "YE", then this presentation is for you. You will learn how to make sure you get the entire string "YES" when creating a new variable. But that is just the beginning of some of the fun you can have with character data. This presentation will walk through examples of cleaning up your character data before you can use procedures such as PROC FREQ. Some of the functions to be covered are SCAN, TRIM, LEFT, INDEX, COMPRESS, COMPBL, DEQUOTE and TRANWRD. This presentation uses BASE SAS® and is aimed at beginning users although all users may learn a new, helpful function.

PROBLEM

You have been given a dataset with information about some of the SSU attendees. You have been asked to supply the following summary information.

- # of people in each region
- # of people interested in each subject area
- list of people sorted by last name
- verify ZIPCODE is a valid United States code

The requirements seem easy and you think you will just need the following PROCs: FREQ, SORT, and PRINT. Since you were not given the variable names, you first do a PROC CONTENTS and a PROC PRINT on the first 5 observations. You discover a few complications.

EXAMPLE DATA

The results of PROC CONTENTS and PROC PRINT are shown at the end of the paper. PROC CONTENTS reveals that the data does not even have the fields ZIPCODE, region or last name. At least it does have a variable for interest. When you look at a few records as shown in the PROC PRINT, you see the missing information is really there but not in the form you need. (Please note that the data are fictitious and any resemblance to real attendees is entirely a figment of the author's imagination.)

THE REAL PROBLEM

Now that you have seen the data, you know you really need to do several things before you can run your PROCs.

- Create region from state after state is extracted from ADD2
- Separate INTEREST into only one interest per record
- Separate first and last names from NAME
- Separate ZIPCODE from ADD2

CREATING NEW VARIABLES

The easiest task is to create region from state. (In real life, you have to extract state from ADD2 first but in a tutorial the presenter gets to rearrange things.) There are several ways to do this. In fact, you could actually use PROC FORMAT and avoid the creation of the new variable region. However, we are going to assume you have a reason for adding the new variable. There are also cases where the logic is more complicated and an IF/THEN or CASE logic solution is the only one feasible. The code would be similar to the following sample section.

```
DATA STATE_REG;
 SET ORIGINAL;
IF STATE IN ('CA', 'OR',...) THEN
   REGION='WEST';
ELSE IF STATE IN ('FL','TN','NC'...) THEN
  REGION='SOUTH';
ELSE IF STATE IN ('OH'...) THEN
  REGION='MIDWEST';
... REST OF STATES...
ELSE REGION='INVALID';
RUN;

PROC FREQ DATA=STATE_REG;
 TABLES REGION / NOCUM NOPERCENT;
RUN;
```

The results give you:

REGION	FREQUENCY
INVA	1
MIDW	1
SOUT	2
WEST	1

While the counts are right, you notice the regions are truncated at 4 characters. You have heard that PROC FREQ had a limitation but upon further investigation you discover that it used to be limited to the first 16 characters but this limitation no longer exists. So what happened? Run another PROC CONTENTS and you will see the REGION variable was created as character variable with a length of 4. SAS determines the length based on the first time it sees the variable. Since your first value was "WEST", the variable was created with a length of 4. One solution would be to put the longest value first. Unfortunately the longest value is "INVALID" but it always needs to be last. Another choice would be to pad your first value with blanks - "WEST " would solve the problem. This works but is very susceptible to future problems if you need to add values. The better alternative is to use a length statement.

```
LENGTH REGION $7.;
...put IF statements AFTER length statement
```

Your new results are:

REGION	FREQUENCY
INVALID	1
MIDWEST	1
SOUTH	2
WEST	1

Of course, someone will want "INVALID" to appear at the end. They might also want the other values to appear in a specific order. PROC FREQ does have several options that will control order but unfortunately there is not one for "how I typed them." One alternative is to use lead blanks since blanks will sort first. However, that could get quite confusing and would look rather strange. This time PROC FORMAT is probably the best way to solve the problem. Instead of creating the region as a character variable you will want to create it as a numeric variable with the regions in the order that you want to see in your output. PROC FORMAT is used to show the complete region name for the numeric variable.

```
PROC FORMAT;
 VALUE REGNAME
 1='WEST'
 2='SOUTH'
 3='MIDWEST'
 4='NORTH'
 5='INVALID';
PROC FREQ;
 TABLES REGION/NOCUM NOPERCENT;
 FORMAT REGION REGNAME.;
RUN;
```

The results are:

REGION	FREQUENCY
WEST	1
SOUTH	2
MIDWEST	1
INVALID	1

EXTRACTING VALUES FROM EXISTING VARIABLES

The rest of the presentation shows many of the functions that SAS has included for working with character data. How do we get STATE and ZIPCODE from ADD2? Fortunately, when you looked at your observations you see that all ADD2 lines were entered in the form "CITY, STATE ZIPCODE". The comma and blank between CITY and STATE and the blank between STATE and ZIPCODE are important. STATE is a 2-character postal code. Some ZIPCODEs were entered as 5 characters while others are in the ZIP+4 FORMAT. This difference will not cause a problem but other differences might. If some observations had been entered in a different form, you might still be able to do the processing but it would be more complicated. There may be times when you have no choice except to edit each observation.

The first observation is:
Raleigh, NC 23232-2222

There are actually several ways to extract STATE and ZIPCODE. The method that will be shown later for last name could also be used here. The SCAN function will break a string into "words" based on the delimiters.

```
WORD1=SCAN(ADD2,1,',');
WORD2=SCAN(ADD2,2,',');
```

The above code will break ADD2 into words using a comma as the delimiter. The first word will be characters up to the first comma but not including the comma. The second word will be everything after the first comma up to the second comma. In this case, there is no second comma in ADD2 so it will be to the end of the string. If you asked for a third word, you would get a blank.

You can specify one or more delimiters or use the defaults. You must put the delimiter(s) in quotes or SAS will think you are using a variable name (, is not a valid variable name so you will really have an error). Do not put any variable names in quotes or SAS with think you want the string "ADD2" scanned rather than the value of the variable.

For the first observation the results are:

WORD1=Raleigh
WORD2= NC, 23232-2222

PROC CONTENTS will reveal a characteristic of the SCAN function that will often cause problems although it does not in this case. Variables created with a SCAN function will always have a length of 200 unless you specify a length before you use the function.

To separate WORD2 into STATE and ZIPCODE, we will use the SUBSTR function. It extracts a "subset" of a string based on the positions you specify.

```
STATE=SUBSTR(WORD2,1,2);
ZIPCODE=SUBSTR(WORD2,4);
```

STATE is created by extracting 2 characters from WORD2 starting at position 1 in the string. ZIPCODE is created by starting at position 4. Since the number of characters was not specified, all characters through the end of the string will be extracted. The starting position is required but the number of characters to extract is not. The results are:

STATE= N ZIPCODE=23232-2222

Why did ZIPCODE work but not STATE? Actually neither one is what you might think. The blank between the comma and the state code caused the problem. The first and fourth characters of WORD2 are both blanks and SUBSTR counted them. ZIPCODE actually starts with a blank. To solve this problem you can either change your SUBSTR function or use another function first to eliminate any lead blanks. The second method is safer because you never know when you will have 2 or more lead blanks.

```
WORD2=LEFT(WORD2);
```

This code will left-justify the string and eliminate any lead blanks. You would then follow this code with the SUBSTR function statements. The following code will give you correct results.

```
WORD1=SCAN(ADD2,1,',');
WORD2=LEFT(SCAN(ADD2,2,','));
STATE=SUBSTR(WORD2,1,2);
ZIPCODE=SUBSTR(WORD2,4);
```

Someone might ask why we didn't use a comma and blank as the delimiters in the SCAN function. ZIPCODE would then be the third word and the blanks would not be a problem because delimiters do not appear in the words from SCAN. This would work for our first example but it would not work for

San Francisco, CA 99494

In this case, WORD1 would be "San", WORD2 would be "Francisco" and WORD3 would be "CA" with ZIPCODE being the fourth word. There are even some cities with three words in the name.

One other thing that you noticed in reviewing your data was that "San Francisco" was incorrectly typed with two blanks instead of one. Fortunately there is now a function to fix that problem.

```
CITY=COMPBL(CITY);
```

The COMPBL (compress blanks) function will compress multiple blanks into a single blank. It you wanted to remove all the blanks you would use the COMPRESS function.

```
CITY=COMPRESS(CITY);
```

Too bad we can not use the same logic for extracting last name because there is not a pattern – or is there? Did you notice that the last name occurs after the last blank? But how do you find the last blank? There is not a SCAN from the right function but there is a REVERSE function.

```
REVNAME=REVERSE(NAME));
```

The reversed names are:

```
nosleN yenraB
taogiluK ydnA
 kcuB naV beD
tebbiR hpesoJ .S
  HTROW NEROL
```

It is obvious there are lead blanks since the names are right-aligned but SAS prints character variables as left-aligned. NAME was created with a length of 30

so any name with less than 30 characters will have blanks added to make it 30. The trailing blanks are now lead blanks when the REVERSE function is used. PROC PRINT will actually drop off the lead blanks that are common to all the values but you will see them if you use FSVIEW.

Now that you have REVERSEd the string, you can use SCAN to break REVNAME into first and last names. When there are several delimiters together, they will be treated as one. If the first character is a delimiter, it is ignored. You will need to remember to type in an actual blank in the SCAN function to specify the delimiter. The length is specified so you do not end up with 200 characters. Although the length could be shorter, I used the same as the original variable to ensure there were not any people with a single name which happened to be 30-characters (we have Cher, Madonna and John-Boy so why not David-Joseph-Andrew-GregoryBob).

```
LENGTH REVNAME LASTNAME REST_NAME
$30;

REVNAME=REVERSE(NAME);
LASTNAME1=SCAN(REVNAME,1,' ');
LEFTNAME1=LEFT(REVNAME);
Z=INDEX(LEFTNAME1,' ');
REST_NAME1=SUBSTR(LEFTNAME1,Z);
```

The name is reversed as shown above. SCAN is used to get the first word from the reversed name so you will end up with the last name. Getting the rest of the name is a little trickier because you need to account for people like "S. Joseph Ribbet". You can simply select the second word because that would only give you "Joseph". The LEFT function will get rid of the leading blanks. The INDEX function returns the location of the specified character. In the examples of

```
nosleN yenraB
taogiluK ydnA
```

the first blanks are the 7th character and 9th character, respectively. You then use these results as part of the parameters for your SUBSTR function. Earlier we specified a specific number for the starting point of the substring function and it applied to all observations. In this example, the starting point will be specific to each observation. You will end up with:

NAME	LASTNAME1	REST_NAME1
Andy Kuligoat	taogiluK	ydnA
Deb Van Buck	kcuB	naV beD
S. Joseph Ribbet	tebbiR	hpesoJ .S
LOREN WORTH	HTROW	NEROL

Now you just need to reverse them back. Do not forget to left-justify them, too. So you have

```
LASTNAME=LEFT(REVERSE(LASTNAME1));
FIRSTNAME=LEFT(REVERSE(REST_NAME1));
```

Although it wasn't explicitly request, you are also going to assume the names need to be in the form "last name, first name". But first, you noticed that Loren Worth was all capital letters and you really want mixed case. There are UPCASE and LOWCASE functions. You just need to separate out the first letter and UPCASE it while using LOWCASE on the rest of the letters. The following code should accomplish this and put everything back together again in the desired form.

```
LENGTH FIRSTLTR LASTLTR $1;
FIRSTLTR=
   UPCASE(SUBSTR(FIRSTNAME,1,1));
FIRST_REM=
   LOWCASE(SUBSTR(FIRSTNAME,2));
LASTLTR=
   UPCASE(SUBSTR(LASTNAME,1,1));
LAST_REM=
   LOWCASE(SUBSTR(LASTNAME,2));
NEWNAME=
   LASTLTR || TRIM (LAST_REM)||
   ', ' ||
   FIRSTLTR || FIRST_REM;
```

The || is the concatenate symbol. Sometimes finding this on your keyboard is the hardest part about writing the code. You may need to check your keyboard mapping to find the right keys. Your final results are:

```
Nelson, Barney
Kuligoat, Andy
Buck, Deb van
Ribbet, S. joseph
Worth, Loren
```

You will notice that everything looks fine except for the two people that really had three part names. You have a couple options and which is most appropriate will depend upon your situation. You could write additional code similar to the above code that would check for 3 or even 4-part names. But what if you have a name like "Dr. John Jacob Wizehimer III"? The possibilities are really endless. You need to decide if having a few wrong cases is worth the effort to try to identify everything. Another option is to simply identify the observations that did not have the

standard 2-part name and edit these by hand. This can be done easily by using SCAN to see if the third word is not blank. In my case, this has been the best solution because I only had 4 or 5 special cases out of 30,000 observations. One other thing to consider is the case of Deb Van Buck. Is "Van" really her middle name or is her last name "Van Buck". How do you know which of the following is the correct result?

Buck, Deb Van
Van Buck, Deb

The next step is to process the INTEREST variable. The requester wanted to know how many people were interested in each subject area. A quick PROC FREQ would seem to give you the answer. However, people have multiple interests so you need to account for each one. Just like you have before, you can use SCAN to break the variable into individual words. The difference this time is that the new variable will have the same variable name for the different words and you will use an OUTPUT statement after you create each word. The OUTPUT statement will result in one record per word per original record.

```
NEW_INTEREST=SCAN(INTEREST,1,',');
OUTPUT;
NEW_INTEREST=SCAN(INTEREST,2,',');
OUTPUT;
... continue until you have accounted for the
maximum number of words...
```

How do you know the maximum number of words? You do not want the people with fewer intereststo have blank records. The easiest way to do this is to count the number of commas. You will need to add 1 to the count since the last word will not be followed by a comma.

The LENGTH function is used to determine the actual length of a value excluding any trailing blanks. Do not confuse this with the LENGTH statement which sets the length for the variable for all observations. The COMPRESS function is used to eliminate certain characters from a string. To count the occurrences of a character, find the length of the original value and subtract the length after you have eliminated that character.

```
INTEREST_COUNT=LENGTH(INTEREST) -
LENGTH(COMPRESS(INTEREST,',')) + 1;
```

You now have an accurate count of the interests for each observation. You can now use a DO LOOP to write your statements.

```
DO I=1 TO INTEREST_COUNT;
NEW_INTEREST=
LEFT(UPCASE(SCAN(INTEREST,I,',')));
  OUTPUT;
END;
```

You will notice I also put in the LEFT and UPCASE functions. LEFT will eliminate any leading blanks that might have been between the comma and the interest value while UPCASE will ensure the case is consistent across observations. However, you still have the problem where people used differently terminology for the same thing. You could use IF/THEN statements , CASE logic or PROC FORMAT to change the values. Another method is to use the TRANWRD function.

```
LENGTH NEW_INTEREST2 $30;
NEW_INTEREST2=
   TRANWRD
   (NEW_INTEREST,'AUDIO/VISUAL','A/V');
```

This will change the string "AUDIO/VISUAL" to the shorter string "A/V". The LENGTH statement is needed or you end up with a 200-byte field. Do not confuse this function with TRANSLATE which has been available longer. TRANSLATE changes individual letters. It also has you specify the outcome before the original value. If you attempted to use

```
TEST= TRANSLATE
   (NEW_INTEREST,'AUDIO/VISUAL','A/V');
```

you would get "AUDIOUDISUAL" when the original value was "AUDIO/VISUAL". This is because all "A"s translated to "A", "/" translated to "U" and "V" translated to "D".

Just when you think you are finished, you also notice the observation which had quotes around the person's interests. You can now easily eliminate the quotes

```
INTEREST=DEQUOTE(INTEREST);
```

You can also do the opposite with the QUOTE function. The latter is helpful if you will be writing your data to a file so it can be read with other software. A primary use is when you need to write a comma-separated file but your data happens to have commas in it. By using quotes around the entire value, the embedded commas will be ignored.

The requester also wanted to know if everyone provided a legitimate ZIPCODE and how many

provided the ZIP+4 version. To determine if the ZIPCODE is a numeric or a numeric with a dash, you can use the verify function:

CHECK=VERIFY(ZIPCODE,'0123456789- ');

If the only characters are the ones specified, then the variable CHECK will have a value of 0. Otherwise, it will have the position of the first character which was not specified. You must include the blank because observations with a 5-digit code rather than ZIP+4 will have a value of 6 since the 6^{th} –10^{th} characters are blank.

OTHER FUNCTIONS

Believe it or not, there are even more character functions. The above examples used INDEX. There is also INDEXC and INDEXW functions. INDEXC looks for any character while INDEXW looks for the entire word. For example, the data has observations with interests for

Data warehousing, a/v, food

logistic regression, data mining

If you look for "housing", INDEX will find it at position 10 in the first record and will not find it in the second. INDEXW will not find it in either record because it looks for an entire word rather than a partial word. You also need to note that a comma immediately after the desired string will prevent INDEXW from finding it. INDEXC will return a 10 for the first record a 2 for the second record. You might wonder how this is possible since 'housing' is not in the second record. While the entire word is not there, the letter 'o' is indeed in the second position.

There is also a RIGHT function which right-justifies data just like the LEFT function. In fact, we could have used it in the above code when we were reversing name to avoid getting the lead blanks.

TRIMN is similar to TRIM but it returns a null string if the expression is missing.

There are also several functions which I have not yet had the need to use. These are listed at the end of the paper following the code.

SUMMARY

There is a lot of information here but you should be able to handle almost any character data that you might encounter. I said almost because I did not cover things like non-printable characters or fields that are too wide to be printed or viewed. When you work with character data, you should remember to:

- Use PROC CONTENTS to verify lengths
- Specify lengths for new variables
- Look for patterns to use to break up strings
- Watch out for leading and trailing blanks
- Remember there can be more than one way to accomplish the same results
- Sometimes you have to be satistified with incorrect results or edit the data by hand

The final code and output are shown at the end of the paper. In some places, the code is different that in the body of the paper because I combined steps.

CONTACT INFO
Deb Cassidy
Cardinal Distribution
7000 Cardinal Place
Dublin, OH 43017
deb.cassidy@cardinal.com
614-757-7136

PART OF PROC CONTENTS

-----Alphabetic List of Variables and Attributes-----

#	Variable	Type	Len	Pos
2	add1	Char	30	38
3	add2	Char	30	68
4	interest	Char	60	98
1	name	Char	30	8
5	num_papers	Num	8	0

FIRST 5 OBSERVATIONS

Obs	name	add1	add2
1	Barney Nelson	953 Probe Way	Raleigh, NC 23232-2222
2	Andy Kuligoat	23134 Indianpaint Road	Chattanooga, TN 37204
3	Deb Van Buck	1524 Patty Lane	Franklinton, OH 43222-6028
4	S. Joseph Ribbet	Georgia Street, PO Box 4321	Sierra, CA 95474
5	LOREN WORTH	Nucleic Way	San Francisco, CR 99494

Obs	interest	num_ papers
1	"Emerging Technologies, WEB pages"	51
2	Data warehousing, a/v, food	17
3	Training	1
4	logistic regression, data mining	8
5	data presentation, Audio/Visual, logistic regression	.

FINAL OUTPUT

REGIONS AND ZIPCODE CHECK

The FREQ Procedure

region	Frequency
WEST	1
SOUTH	2
MIDWEST	1
INVALID	1

CHECK	Frequency
0	5

LIST OF PARTICIPANTS

Obs	NAME
1	Kuligoat, Andy
2	Nelson, Barney
3	Ribbet, S. Joseph
4	Van Buck, Deb
5	Worth, Loren

Interests After Combining for Spelling Differences

The FREQ Procedure

interest2	Frequency
A/V	2
DATA MINING	1
DATA PRESENTATION	1
DATA WAREHOUSING	1
EMERGING TECHNOLOGIES	1
FOOD	1
LOGISTIC REGRESSION	2
TRAINING	1
WEB PAGES	1

FINAL CODE

```
PROC CONTENTS DATA=ORIGINAL;
TITLE 'ORIGINAL DATA';
RUN;

PROC PRINT DATA=ORIGINAL;
TITLE 'ORIGINAL DATA';


DATA FINAL;
 SET ORIGINAL;
 LENGTH REVNAME $30 FIRSTLTR LASTLTR $1  NEWNAME $31 ;
 REVNAME=LEFT(REVERSE(NAME));
 LASTNAME1=SCAN(REVNAME,1,' ');
 THIRD_NAME=SCAN(REVNAME,3,' ');
 FIRST_SPACE=INDEX(REVNAME,' ');
 REST_NAME=SUBSTR(REVNAME,FIRST_SPACE);

 LASTNAME=LEFT(REVERSE(LASTNAME1));
 FIRSTNAME=LEFT(REVERSE(REST_NAME));

 FIRSTLTR=UPCASE(SUBSTR(FIRSTNAME,1,1));
 LASTLTR=UPCASE(SUBSTR(LASTNAME,1,1));
```

```
 FIRST_REM=LOWCASE(SUBSTR(FIRSTNAME,2));
 LAST_REM=LOWCASE(SUBSTR(LASTNAME,2));

 NEWNAME=LASTLTR || TRIM(LAST_REM) || ', ' || FIRSTLTR || FIRST_REM;

CITY=COMPBL(SCAN(ADD2,1,','));
STATE_ZIP=LEFT(SCAN(ADD2,2,','));
STATE=SUBSTR(STATE_ZIP,1,2);
ZIPCODE=SUBSTR(STATE_ZIP,4);

          IF STATE IN ('CA', 'OR') THEN REGION=1;
  ELSE IF STATE IN ('TN', 'NC') THEN REGION=2;
  ELSE IF STATE IN ('OH') THEN REGION=3;
  ELSE REGION=5;
   **** INCLUDE REST OF STATES HERE. ;


CHECK=VERIFY(ZIPCODE,'0123456789- ');

RUN;

*** NOTE THAT ANY NAMES WITH 3 OR MORE PARTS WILL BE EDITED BY HAND AT THIS POINT;


PROC FORMAT;
 VALUE REGNAME
 1='WEST'
 2='SOUTH'
 3='MIDWEST'
 4='NORTH'
 5='INVALID';
 RUN;

PROC FREQ DATA=FINAL;
  TABLES REGION CHECK/NOCUM NOPERCENT;
  FORMAT REGION REGNAME.;
  TITLE 'REGIONS AND ZIPCODE CHECK';
  RUN;

PROC SORT DATA=FINAL;
 BY NEWNAME;
RUN;

PROC PRINT DATA=FINAL SPLIT='*';
 VAR NEWNAME;
 LABEL NEWNAME='NAME';
 TITLE 'LIST OF PARTICIPANTS';
 RUN;


 DATA ALL_INTEREST;
   SET FINAL (KEEP=INTEREST);
   INTEREST=DEQUOTE(UPCASE(INTEREST));
   NUM_COMMAS=LENGTH(INTEREST) - LENGTH(COMPRESS(INTEREST,',')) + 1;
   DO I=1 TO NUM_COMMAS;
```

```
    INTEREST2=LEFT(SCAN(INTEREST,I,','));
    OUTPUT;
    END;
   RUN;

PROC FREQ DATA=ALL_INTEREST;
 TABLES INTEREST2/NOCUM NOPERCENT;
 TITLE 'FIRST PASS FORINTERESTS';
 RUN;

  *** COMBINE INTERESTS WHICH WERE SPELLED DIFFERENTLY;
DATA ALL_INTEREST2;
    SET ALL_INTEREST;
    INTEREST2=TRANWRD(INTEREST2,'AUDIO/VISUAL','A/V');
RUN;

PROC FREQ DATA=ALL_INTEREST2;
 TABLES INTEREST2/NOCUM NOPERCENT;
 TITLE 'INTERESTS AFTER COMBINING FOR SPELLING DIFFERENCES';
 RUN;
```

OTHER FUNCTIONS FOR CHARACTER DATA

Byte(n)	Returns one character in the ASCII or EBCDIC sequence where n is an integer representing a specific ASCII or EBCDIC character
COLLATE(start-position<,end-position>) | (start-position<,,length>)	Returns and ASCII or EBCDIC collating sequence character string
RANK(x)	Returns the position of a character in the ASCII or EBCDIC collating sequence
REPEAT(argument,n)	Repeats a character expression
SOUNDEX(argument)	Encodes a string to facilitate searching
SUBSTR(argument, position<,n>)=characters-to-replace	Replaces character value contents

Paper P811

INVALID: a Data Review Macro
Using PROC FORMAT Option OTHER=INVALID to Identify and List Outliers

Ronald Fehd, Centers for Disease Control and Prevention, Atlanta GA

ABSTRACT

Data cleansing, or as it is more euphemistically known: data review, often occupies too much of a programmer's time and energy. With a properly written data dictionary, a data set will contain appropriate formats for each variable; one can then cut and paste the format definitions into proc FORMAT value statements and label all outliers with the value statement option: other="INVALID". This routine combines a FORMATS catalogue and CONTENTS data set, then uses that information to write data steps which select all outliers. Reports are written, by identifier, for print review and to file, for later use as includes for updating purposes. Expected audience is intermediate and advanced programmers and macro users.

INTRODUCTION

Data review consists of several steps:
1. Determining acceptable values
2. Finding unacceptable values
3. Comparing unacceptable values with data collection form
4. Deciding whether to change the data
5. Recording changes
6. Updating the data

This macro is a method for a data set. Its necessary parameters are a data set name, and a list of identifiers. It is assumed that variables have formats assigned to them, and that the formats have the option other="&INVALID.". Output from the macro includes both summary and detail reports. The primary detail report is written to a file. This file can be edited to record changes to the data and then used as a %include file to update the data.

This macro facilitates step 2: finding all unacceptable and invalid values. However these errant values must be identified in their respective proc FORMAT values statements with the option: other="&INVALID.".

Decisions as to whether to change data – step 4 – are eased by macro INVALID, with its summary reports. Frequencies of both variables and identifiers provide an overview of difficulties in the data set. In addition, a percentage-of-invalid report is attached to the detail, by identifier, report.

Editing the detail, by identifier, report, which is written to a file, addresses the difficulties inherent in step 5. where variable names may be spelled incorrectly, etc.

Finally, updating the data – step 6 – is accomplished easily, using the edited detail report file as a %include file.

DISCUSSION

There must be 50 ways to review variables. Both Fehd (1998) DemoXrpt and Cody(1999) demonstrate using the proc FORMAT value option other='INVALID'. Fehd ensures standardization of the label by using a global macro variable INVALID. Cody and Handsfield (1998) provide ways to mathematically find outliers of numerical variables. McQuown (2000) offers his experience in writing a data review of a large survey.

These authors provide ideas and tools to write customized data review. This macro automates half of that task: it does intra-variablechecking, but not inter-variable checking: Doing logic checks between pairs of variables, is a task left for another paper.

The report writer section of INVALID follows that developed in Fehd (1998) DemoXrpt and Fehd (1998) COMPARWS.

The relation of data dictionary and the proc FORMAT program is discussed in Fehd (2001) Beginner's Tour.

How it works

The pseudo-code for macro %INVALID is:
1. For all variables of a data set
2. Choose all observations with invalid values
3. Print summary and detail reports

1. For all variables of a data set

Proc CONTENTS output data set provides variable names, formats and type. This information is also available from SASHELP.VCOLUMN and proc SQL: DICTIONARY.COLUMNS. Refer to lines 164:167.

2. Choose all observations with invalid values

Invalid values are defined in each value statement of a proc FORMAT program. Refer to the Test Data beginning at line 425. This routine depends on having a global macro variable defined as:

```
%LET INVALID = INVALID;
```

This is used to standardize all the value statement options:

```
other = ''&INVALID.''
```

Macro INVALID identifies all format values with that label in the Control-Out output data set of proc FORMAT. This list of format names is used to create a format in the WORK library named $INVALID. Lines 192:220.

The CONTENTS data set is used to write a series of macro calls to the data review macro: CHK4NVLD in lines 224:243. Before these calls are executed, the report data set INVALID structure is created at lines 249:262.

This CHK4NVLD macro implements a choose with the phrase:

```
put(Var,Format.) eq ''&INVALID.''
```

at line 413. If any invalid observations are output to the DETAILS data se, then the DETAILS data set is appended to the report data set INVALID. See line 421:423.

3. Print summary and detail reports

Four reports are provided, with parameters to enable each:

Report	at lines:	parameter
1. Summary: Identifiers	340:344	SMRYIDS
2. Summary: Variables	346:350	SMRYVARS
3. Details: by Variable	352:362	SMRYNAME
4. Details: by Identifiers	292:337	DETAILS

Examples of the parameters and reports are given in the Test Data.

Assumptions

INVALID provides named parameters for the location of the data set and the format library. These are both assumed to be in
`LIBNAME LIBRARY '<libref>';`
The Test Data section – lines 450:467 – illustrates accessing a data set and formats in the WORK library.

Gotcha! Step 0: Checking assumptions

As I began testing on my production data I found that reports from INVALID did not match my custom-written exception reports. Imagine my surprise when I examined my data sets and found that variables had formats that were not in the data dictionary and there were formats present in the proc FORMAT program that were not attributed to any variable. In order to compare the data set formats with the format in the format library I have added a special report at lines 106:155. Refer to the Test Data section for an example.

Notes

The macro is written according to Fehd (2000) Writing for Reading SAS Style Sheet. Fehd (1997) %ARRAY contains the ARRAY macro. Fehd (2001) Beginner's Tour contains the NOBS macro.

Summary

The routine depends on the value of a global macro variable INVALID. This macro variable is appropriately set in autoexec.sas. See Fehd (2001) Macro Tour for further discussion. Two separate programs use this macro variable: one containing proc FORMAT value statements, and another which calls %INVALID.

CONCLUSION

A properly written data dictionary contains the other='INVALID' option on all format values. This facilitates data review.

Date review was and is time consuming. This routine provides a comprehensive report of invalid values in a data set. The summary reports provide a valuable overview of which variables and which identifiers have invalid values. If data cleansing is necessary, the detail report can be edited and %included to perform an update. This file then contains a record of the updates for later referral.

REFERENCES

Cody, Ron, *Cody's Data Cleaning Techniques Using SAS® Software,* Cary, NC: SAS Institute Inc., 1999. 226 pp.

Fehd, Ronald, *%ARRAY: construction and usage of arrays of macro variables.* Proceedings of the 22nd Annual SAS® Users Group International Conference, Cary, NC: SAS Institute Inc., 1997.
http://www2.sas.com/proceedings/sugi22/CODERS/PAPER80.PDF

Fehd, Ronald, *%COMPARWS: Compare with summary: a macro using proc COMPARE to write a file of differences to edit and use for updates.* Proceedings of the 23rd Annual SAS® Users Group International Conference, Cary, NC: SAS Institute Inc., 1998.
http://www2.sas.com/proceedings/sugi23/Posters/p170.pdf

Fehd, Ronald, *DEMOXRPT: macros for writing Exception Reports: perform range and logic checks on a data set; write file of exceptions to edit and use for updates.* Proceedings of the 23rd Annual SAS® Users Group International Conference, Cary, NC: SAS Institute Inc., 1998.
http://www2.sas.com/proceedings/sugi23/Appdevel/p7.pdf

Fehd, Ronald, *A Beginner's Tour of a Project using SAS® Macros Led by SAS-L's Macro Maven,* Proceedings of the 26th Annual SAS® Users Group International Conference, Cary, NC: SAS Institute Inc., 2001.
http://www2.sas.com/proceedings/sugi26/p066-26.pdf

Handsfield, James, *CHEKOUT: A SAS® Program to Screen for Outliers.* Proceedings of the 23rd Annual SAS® Users Group International Conference, Cary, NC: SAS Institute Inc., 1998.
http://www2.sas.com/proceedings/sugi23/Posters/p197.pdf

McQuown, Gary, *SAS® Macros Are the Cure for Quality Control Pains.* Proceedings of the 13th Annual Northeast SAS® Users Group Conference Cary, NC: SAS Institute Inc., 2000.

Author: Ronald Fehd e-mail: RJF2@cdc.gov
Centers for Disease Control MS-G25
4770 Buford Hwy NE
Atlanta GA 30341-3724 voice: 770/488-8102

ACKNOWLEDGMENTS

This macro is the result of a decade of data cleaning efforts. I'd like to thank my colleagues at CDC for all their dirty data. I couldn't have done it without you.

```
001 ;/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *2001May19 001
002 MACRO: INVALID                          NOTE: uses macros ARRAY, NOBS     002
003                                         NOTE: uses global mac-var INVALID 003
004          method for data set + format library                              004
005          where user-written formats have OTHER="&INVALID."                 005
006                                                                            006
007  USAGE: 1.1 %INVALID();    %*list formats: &LIBRARY.._ALL_ and &FMTLIB;  007
008         1.2 %INVALID(DATA);%*list formats: &LIBRARY..DATA  and &FMTLIB;  008
009         2.1 %INVALID(DATA,ID0);                  one ID                   009
010         2.2 %INVALID(DATA,ID1 ID2);              two IDs                  010
011         2.3 %INVALID(DATA,ID0,PRNTFILE=<fileref>);write report to file    011
012         2.4 %INVALID(DATA,ID0,TESTING=1);        testing                  012
013         2.5 TITLE1 'title for this project';                              013
014             %INVALID(DATA,ID0,TITLEN=2);                                  014
015                                                                           015
016  DESCRIPTION: read   all formats in FMTLIB                                016
017               choose formats with OTHER="&INVALID."                       017
018               make   format $INVALID                                      018
019               review all variables/columns in data set                    019
020               choose invalid values                                       020
021               write  file of invalid values                               021
022                      to be used later as an %INCLUDE file for updates     022
023               read   and print file of invalid values                     023
024               print  summary report(s) of invalid values                  024
025  search for <%**>                                                         025
026  PROCESS: 1. if IDLIST blank, print list of formats . . . . . . . . exit 026
027           2. else: make %ARRAY of ID(s)                                   027
028           3. make %ARRAY of ID(s)'s attributes                            028
029           4. if not exist WORK.FORMATS.$INVALID then create:              029
030           4.1. get all formats with Label=INVALID                         030
031           4.2. if none w/Label=INVALID, print exit-msg  . . . . . . exit 031
032           4.3. prepare proc FORMAT CNTLIN data set for value $INVALID     032
033           4.4. make proc FORMAT value $INVALID                            033
034           5. read CONTENTS, write macro calls                             034
035           6. prepare empty data set INVALID for proc APPEND               035
036           7. execute Check-for-Invalid: build data set INVALID            036
037           8. if INVALID empty, print exit-msg . . . . . . . . . . . exit 037
038           9. make values used in summary                                  038
039          10. write corex to PRNTFILE                                      039
040          11. if wanted, print summary report(s)                           040
041          12. if wanted read and print PRNTFILE                            041
042          13. macro: Check-for-Invalid:                                    042
043              if obs w/invalid values, append to data set INVALID          043
044                                                                           044
045  KEYWORDS: %ARRAY %NOBS                                                   045
046            "invalid values" "data review" "data checking"                 046
047            SESUG 2001: invalid outlier FORMAT fmtlib other=               047
048                                                                           048
049 NOTE: expected number of ID vars is 2, see %LOCAL DIM_IDS, need as parm? 049
050 NOTE: this warning is acceptable                                         050
051 WARNING: Variable VALUENUM has different lengths on BASE and DATA files  051
052                                                    (BASE 8 DATA 4).     052
053                                                                         053
054 Author: Ronald Fehd, B.S. C.Sc                 e-mail: RJF2@cdc.gov      054
055         Centers for Disease Control  MS-G25                             055
056         4770 Buford Hwy  NE                                             056
057         Atlanta  GA  30341-3724                voice: 770/488-8102      057
058 RJF2 99Feb25 begun                  NOTE: lines 059:080 change notes deleted
081 ;/* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . */ 081
082 %MACRO INVALID(/* - - - - - - - - - - - - - - - - - - - - - - - - - - */ 082
083  DATA            /*data set name                                      */ 083
```

```
084 ,IDLIST  =.      /* list of primary-key(s) == by-vars                */ 084
085 ,LIBRARY =LIBRARY/*libname of DATA                                   */ 085
086 ,FMTLIB  =LIBRARY/*libname of catalog FORMATS                         */ 086
087 ,PRNTFILE=PRINT  /*output destination, default = PRINT               ** 087
088                  /*may be 'external-file' -- note quotes -- or fileref*/ 088
089 ,PRNTLIST=PRINT  /*turn off detail listing w/PRNTLIST=NULL           */ 089
090 ,DETAILS =1/*?print details? used when only summary wanted           */ 090
091 ,SUMMARY =1/*?print any of SUMMARY report(s)?                        */ 091
092 ,SMRYIDS =1/*?print FREQ of IDS?                                     */ 092
093 ,SMRYNAME=1/*?print detail  of variables, by name?                   */ 093
094 ,SMRYVARS=1/*?print FREQ of variables with invalid?                  */ 094
095 ,TESTING =0/*?enable test msg and prints?                            */ 095
096 ,TITLEN  =2/*TITLE line #                                           */ 096
097 )/*store des = 'method to list INVALID values in data set'/*.........*/ 097
098 ;%LOCAL LENLABEL;             %LET LENLABEL=60;%*data INVALID & DETAILS; 098
099 %LOCAL  DIM_IDS;              %LET DIM_IDS = 2;%*how many IDs?;          099
100 %LOCAL  VALIDVARNAME;                          %*reset at exit;         100
101 %LET    VALIDVARNAME=%sysfunc(getoption(validvarname,keyword));         101
102 OPTIONS ValidVarName = UPCASE;%*align CONTENTS & user-supplied IDLIST;  102
103 %IF &TESTING %THEN %DO;  OPTIONS   mprint;                        %END; 103
104 %ELSE              %DO;  OPTIONS nomprint;                        %END; 104
105                                                                        105
106 %**1. if IDLIST blank, print list of formats w/INVALID, exit;          106
107 %IF "&IDLIST." eq "." %THEN %DO;%*-----------------------------------; 107
108 %IF "&DATA."   eq ""  %THEN %DO;  %LET DATA = _ALL_;              %END; 108
109 proc CONTENTS data    = &LIBRARY..&DATA.                               109
110               memtype = data                                           110
111                         noprint                                        111
112               out     = CONTENTS                                       112
113               (keep   = Format                                         113
114                where  = (Format not in (' ','$CHAR')));                114
115 proc SORT     data    = CONTENTS                                       115
116                         nodupkey;                                      116
117               by        Format;                                        117
118 proc FORMAT   library = &FMTLIB.                                      118
119               cntlout = FMTLIB                                         119
120               (keep   = FmtName Label Type                             120
121                where  = (Label   = "&INVALID.")                        121
122                rename = (FmtName = Format     ) );                     122
123 %IF &TESTING %THEN %DO;*PROC PRINT;%END;                               123
124 proc SORT     data    = FMTLIB                                         124
125                         nodupkey;                                      125
126               by        Format;                                        126
127                                                                        127
128 data    FMTLIB;                                                       128
129  drop   Label Type;                                                    129
130 *length Format $ 9;                                                   130
131  set    FMTLIB;                                                       131
132  if Type = 'C' then Format = '$' !! Format;                            132
133                                                                        133
134 proc SORT data = FMTLIB;                                             134
135           by     Format;                                             135
136                                                                        136
137 DATA    FMTLIB;                                                       137
138  do until(EndoFile);                                                   138
139   merge CONTENTS(in = haveC)                                           139
140         FMTLIB  (in = haveF)                                           140
141    end = EndoFile;                                                     141
142  by Format;                                                            142
143  Data   = HaveC;                                                       143
144  FmtLib = HaveF;                                                       144
```

```
 if      haveC and     haveF then  Ok = 'OK';                                 
 else if haveC and not haveF then  Ok = '??';                                 
 else                              Ok = '..';                                 
                                   output;                             end;   
                                                                      stop;   
proc PRINT    data = FMTLIB noobs;                                           
              TITLE&TITLEN. "INVALID: compare formats in DATA:"              
                            "&LIBRARY..&DATA. and FMTLIB:&FMTLIB.";          
proc DATASETS library  = WORK nolist;                                        
              delete     CONTENTS FMTLIB (memtype = data);          quit;   
%GOTO ENDOMACR;%*.............................. %IF IDLIST EQ "."; %END;     
%*ELSE *DO: IDLIST not BLANK;                                                
%**2 make %ARRAY of ID(s);        %local I;                                  
%DO I = 1 %TO &DIM_IDS;           %local IDS&I.;                    %END;    
%LET IDLIST = %upcase(&IDLIST.);  %*proc CONTENTS.Name is upcase *;          
%ARRAY(IDS,&IDLIST);run;          %*ARRAY mac-vars are local;               
%DO I = 1 %TO &DIM_IDS;           %local TYPE&I. Q&I. LEN&I.;       %END;    
                                                                             
%**3 make %ARRAY of ID(s) attributes;                                        
proc CONTENTS data  = &LIBRARY..&DATA.                                       
                      noprint                                                
              out   = CONTENTS                                               
              (keep = Format Formatd Formatl Length Name Type VarNum);       
                                                                             
DATA    _NULL_;                                                              
 length Q $ 1;                                                               
 do until(EndoFile);                                                         
  set CONTENTS (where=(Name in (                                             
   %DO I = 1 %TO &DIM_IDS.;     "&&IDS&I"                                    
    %IF &I lt &DIM_IDS %THEN             ,;                        %END;    
   ))) end = EndoFile;                                                       
  %* Type::(1:numeric, 2:character) from proc CONTENTS.Type*;                
  if Type = 1 then do; C_N = ' '; Q = ' ';                            end;   
  else             do; C_N = '$'; Q = "'";                            end;   
  %DO I = 1 %TO &DIM_IDS.;                                                   
   if Name = "&&IDS&I." then do; call symput("TYPE&I.",C_N);                 
                                 call symput(   "Q&I.",Q  );                 
                                 call symput( "LEN&I.",                      
                                        compress(put(Length,3.))); end;      
                                                      %*%DO I=1; %END;       
 %*..................................... do until(EndoFile)*; end;          
                                                                      stop;   
%**4 does format $INVALID? exist?, because of previous use of macro?;        
%*func catalog-exist(LibName.MemName.ObjName.ObjType);                       
%IF not %sysfunc(cexist(WORK.FORMATS.INVALID.FORMATC)) %THEN %DO;%*---*;     
                                                                             
%**4.1 get all formats which have Label=INVALID;                             
proc FORMAT library =&FMTLIB                                                 
            cntlout = FORMAT_CNTL                                            
            (keep   = FmtName Label Type                                     
            where   = (Label = "&INVALID."));                                
                 %*note: global &INVALID;                                    
%**4.2 if none w/Label=INVALID, exit;                                        
%local   ANY_FRMT;           %NOBS(ANY_FRMT);run;                            
%IF not &ANY_FRMT %THEN %DO;  %PUT no formats with label=<&INVALID.>;        
                              %GOTO ENDOMACR;                        %END;   
                                                                             
%**4.3 make proc FORMAT CntlIn data set for value $INVALID;                  
DATA    FORMAT_CNTL           (drop=Type);                                   
 retain Label   '1'           %*one==True;                                   
        FmtName '$INVALID'                                                   
```

```
206         HLO     ' ';                                                    206
207  do until(EndoFile);                                                    207
208   set FORMAT_CNTL (keep   =  FmtName   Type                             208
209                    rename = (FmtName = Start))                          209
210                    end    =  EndoFile;                                  210
211   if Type = 'C' then Start = '$' !! Start;                              211
212   output;                                          %*do until(EoF);end; 212
213  HLO   = 'O';                 %*Oh==Other;                             213
214  Label = 'O';                 %*zero==False;                           214
215  Start = '**OTHER**';                                                  215
216  output;                                                      stop;    216
217                                                                        217
218 %**4.4 make FORMAT value $INVALID;                                     218
219 proc FORMAT cntlin  = FORMAT_CNTL                                     219
220             library = WORK;%* . . . . . . . . . .%IF not cexist; %END; 220
221                                                                        221
222 filename TEMPTEXT catalog 'sasuser.profile.sasinp.source';            222
223                                                                        223
224 %**5 read CONTENTS, write macro calls;                                 224
225 %local ANY_FRMT;             %LET ANY_FRMT=0;%*flag for exit;         225
226 %local CHARVLEN;             %LET CHARVLEN=0;%*max char var length;   226
227 data  _NULL_;                                                         227
228  file %IF &TESTING %THEN %DO; PRINT                          %END;    228
229       %ELSE              %DO; TEMPTEXT                       %END;    229
230                                                  %*file closure;;      230
231  retain CharVLen  1                                                    231
232         Any_Frmt 'O';                                                  232
233  do until(EndoFile);                                                   233
234   set CONTENTS                                                         234
235    (where=(input(put(Format,$INVALID.),1.)))                           235
236    end = EndoFile;                                                     236
237   Any_Frmt = '1';                                                      237
238   if Type = 2 then CharVLen = max(CharVLen,Length);                    238
239   put '%CHK4NVLD(' "&DATA." ',' Name ',' Type ',' Format ',' VarNum ')'; 239
240  %*..................................... do until(EndoFile); end;    240
241  call symput('ANY_FRMT',             Any_Frmt     );                   241
242  call symput('CHARVLEN',compress(put(CharVLen,3.)));          stop;    242
243  %*note       CHARVLEN used by CHK4NVLD;                       run;    243
244 %IF &TESTING %THEN %put ANY_FRMT<&ANY_FRMT.> CHARVLEN<&CHARVLEN.>;     244
245                                                                        245
246 %IF not &ANY_FRMT %THEN %DO;%PUT data &DATA has no formats w/&INVALID.; 246
247                          %GOTO ENDOMACR;                   %END;       247
248                                                                        248
249 %**6 prepare empty data set for proc APPEND;                          249
250 DATA INVALID;                                                         250
251  length                                                                251
252   %DO I = 1 %TO &DIM_IDS.; &&IDS&I &&TYPE&I &&LEN&I                   %END; 252
253   Var      $ 32                                                        253
254   Var_Info $ 60                                                        254
255   Type     $  1 %*note: CONTENTS.Type is numeric;                      255
256   ValueChr $  &CHARVLEN.                                               256
257   ValueNum    8 %*WARNING different lengths on BASE and DATA;          257
258   VarNum      4                                                        258
259   Label    $  &LENLABEL                                                259
260   N           4 %*use: report counter;                                 260
261   Format   $  8                                                        261
262   ;                                                           stop;    262
263                                                                        263
264 %**7;%include TEMPTEXT %IF &TESTING %THEN %DO; /source2          %END;  264
265                                                %*include closure;;     265
266 filename TEMPTEXT clear;                                              266
```

```
                                                                                                                    
%local DATANOBS; %NOBS(DATANOBS,data = &LIBRARY..&DATA.);                                                           
%local NVLDCELS; %NOBS(NVLDCELS,data = INVALID);run;                                                                
                                                                                                                    
TITLE&TITLEN. "invalid values in &DATA. Obs:&DATANOBS.";                                                            
                                                                                                                    
%**8;%IF not &NVLDCELS %THEN %DO; DATA _NULL_;                                                                      
                                  file &PRNTFILE;                                                                   
                                  put '***no invalid values;';                                                      
                                  %GOTO ENDOMACR;                  %END;                                            
%**9 make values used in summary;                                                                                   
proc FREQ data =  INVALID;    %*do cross-tab of IDs;                                                                
          tables  &IDS1                                                                                             
          %IF &DIM_IDS ge 2 %THEN                                                                                   
          %DO I = 2 %TO &DIM_IDS;  * &&IDS&I.                    %END;                                            
          /       list noprint                                                                                      
          out   = INVALID_IDS;                                                                                      
          tables  Var                                                                                               
          /       list noprint                                                                                      
          out   = INVALID_VARS;                                                                                     
                                                                                                                    
%local NVLDIDS ; %NOBS(NVLDIDS ,data = INVALID_IDS );                                                               
%local NVLDVARS; %NOBS(NVLDVARS,data = INVALID_VARS);                                                               
%local DATAVARS; %NOBS(DATAVARS,data = CONTENTS);run;                                                               
                                                                                                                    
%IF &DETAILS. %THEN %DO;%*-------------------------------------------; 22
%**10 write corex to PRNTFILE; options pageno=1;                                                                    
TITLE&TITLEN. "invalid values in &DATA. Obs:&DATANOBS.";                                                            
                                                                                                                    
PROC SORT data = INVALID;                                                                                           
          by     &IDLIST.;                                                                                          
                                                                                                                    
DATA    _NULL_;                                                                                                     
 file   &PRNTFILE;                                                                                                  
 retain B -1                                                                                                        
        Q "'";                %*squote;                                                                           
 do until(EndoFile);                                                                                                
  set   INVALID  end = EndoFile;                                                                                    
  by    &IDLIST;                                                                                                    
  if first.&&IDS&DIM_IDS then do;                                                                                   
                             put "*;if &IDS1 = &Q1"   &IDS1  +B  "&Q1."                                             
   %IF &DIM_IDS. ge 2 %THEN                                                                                         
    %DO I = 2 %TO &DIM_IDS.; " and &&IDS&I = &&Q&I." &&IDS&I +B "&&Q&I."                                            
                                                          %END;                                                     
                             " then do;";             %*if first.*; end;                                            
                                                                                                                    
  if Type eq '1' then put '* ' Var @13 '=' @15      ValueNum       ';';                                             
  if Type eq '2' then put '* ' Var @13 '=' @15 Q +B ValueChr  +B Q ';';                                             
                                                                                                                    
  if last.&&IDS&DIM_IDS then  put '*;end;';                                                                         
 %*..................................... do until(EndoFile)*; end;                                                 
%**;%macro MAKEPCNT(VAR,NUM,DENOM);%*create % trim value to &D decimals;                                            
%local D I TRIMLEN; %LET D = 3; %LET TRIMLEN = 0;                                                                   
%LET &VAR = %sysevalf(100 * &&&NUM  / &&&DENOM);                                                                    
%LET I    =  %index(&&&VAR,.);%*if dot in value trim number of decimals;                                           
%IF &I %THEN %DO; %LET TRIMLEN = %eval(%length(&&&VAR) - &I);                                                       
                  %IF &TRIMLEN gt &D %THEN %LET TRIMLEN = &D;                                                       
                  %LET &VAR = %substr(&&&VAR,1,&I + &TRIMLEN);     %END;                                            
%*PUT &VAR = &&&VAR;%*.............................MAKEPCNT*; %MEND;                                                
%** print summary percentages at end of file;                                                                       
%local DATACELS;%LET DATACELS = %eval(&DATANOBS * &DATAVARS);                                                       
```

```
%local PCNTNOBS;%MAKEPCNT(PCNTNOBS,NVLDIDS ,DATANOBS);
%local PCNTVARS;%MAKEPCNT(PCNTVARS,NVLDVARS,DATAVARS);
%local PCNTCELS;%MAKEPCNT(PCNTCELS,NVLDCELS,DATACELS);
retain           C1 12             C2 22            C3 32;
put "***smry :" @C1 "Ids"         @C2 "Vars"       @C3 "Cells;";
put "*** Base:" @C1 "&DATANOBS"   @C2"&DATAVARS"   @C3 "&DATACELS;";
put "*Invalid:" @C1 "&NVLDIDS"    @C2"&NVLDVARS"   @C3 "&NVLDCELS;";
put "*** Pcnt:" @C1 "&PCNTNOBS%"  @C2"&PCNTVARS%"  @C3 "&PCNTCELS%;";
                                                                stop;
%*.............................................. %IF &DETAILS *; %END;
%**11 print desired summary report(s);
%IF &SUMMARY. %THEN %DO;options pageno=1;%*----------------------------;
%IF &SMRYIDS %THEN %DO;
 proc PRINT data = INVALID_IDS;
             sum    Count;
             TITLE%eval(&TITLEN.+1) "summary: IDs listed "
             "invalid<&NVLDIDS.> / data<&DATANOBS.> = &PCNTNOBS.%"; %END;

%IF &SMRYVARS %THEN %DO;
 proc PRINT data = INVALID_VARS;
             sum    Count;
             TITLE%eval(&TITLEN.+1) "summary: Variables listed "
             "invalid<&NVLDVARS.> / data<&DATAVARS.> = &PCNTVARS.%";%END;

%IF &SMRYNAME %THEN %DO;
 proc SORT  data = INVALID;
             by     Var &IDLIST.;
 proc PRINT data = INVALID  label ;*    (drop = Type VarNum Label);
             var    &IDLIST. Value:;
             sum    N;
             by     Var_Info;
             id     Var_Info;
             label  Var_Info = 'Var Format Label';
             TITLE%eval(&TITLEN.+1) "detail: by Var-Name "
             "invalid<&NVLDCELS.> / data<&DATACELS.> = &PCNTCELS.%";%END;
%**12;%IF "&PRNTLIST" eq "PRINT"
      and "&PRNTFILE" ne "PRINT" %THEN %DO;%*-------------------------;
data _NULL_; %*print detail list*; options pageno=1;
 TITLE%eval(&TITLEN.+1);
 file PRINT;
 do until(EndoFile);%local LRECL;%LET LRECL = 72;
  infile &PRNTFILE end = EndoFile pad lrecl = &LRECL.;
  input @1 Line $char&LRECL..;
  put   @1 Line $char&LRECL..;             %*do until(EndoFile)*;  end;
                                                                stop;
%*................... %IF PRNTLIST eq PRINT & PRNTFILE ne PRINT*; %END;
%*.............................................. %IF &SUMMARY *; %END;
%ENDOMACR: run;TITLE%eval(&TITLEN.);options &VALIDVARNAME.;
%IF not &TESTING %THEN %DO;
proc DATASETS library  = WORK     nolist;
              delete     CONTENTS DETAILS     FORMAT_CNTL
                         INVALID  INVALID_IDS INVALID_VARS
              (memtype = data);                                 quit;
%*.......................................... %IF not &TESTING*; %END;
OPTIONS nomprint; run;%*.............................INVALID*; %MEND;
%**13;%macro CHK4NVLD(DATA,VAR,TYPE,FORMAT,VARNUM);%* - - - - - - - - - -;
%*note CHARVLEN created by calling macro INVALID;
%local HAVEDATA;%LET HAVEDATA = 0;%*flag for append;

data DETAILS(drop   = HaveData
             rename = (&VAR =
```

```
389             %IF "&TYPE" = "2" %THEN ValueChr;                          389
390             %ELSE                   ValueNum;                    )); 390
391  length Var      $ 32                                                  391
392         Var_Info $ 60                                                  392
393         Type     $  1  %*NOTE: CONTENTS.Type is numeric;               393
394         %IF "&TYPE" = "2" %THEN %DO; ValueNum 8                 %END; 394
395         %ELSE                   %DO; ValueChr $ &CHARVLEN.      %END; 395
396         VarNum      4                                                  396
397         Label    $  &LENLABEL                                          397
398         N           4  %*use: report counter;                         398
399         Format   $  8;                                                 399
400  retain Var      "&VAR."                                               400
401         Var_Info "&VAR."                                               401
402         HaveData '0'                                                   402
403         Type     "&TYPE."                                              403
404         %IF "&TYPE"="2" %THEN %DO; ValueNum .                   %END; 404
405         %ELSE                  %DO; ValueChr '.'                %END; 405
406         VarNum    &VARNUM.                                             406
407         Label    "."                                                   407
408         N         1                                                    408
409         Format   "&FORMAT.";                                           409
410  do until(EndoFile);                                                   410
411   set &LIBRARY..&DATA.                                                 411
412       (keep  = &IDLIST &VAR.                                           412
413        where = (put(&VAR.,&FORMAT..) eq "&INVALID.")                   413
414       ) end  = EndoFile;                                               414
415   if Label eq '.' then do;   call label(&VAR.,Label); %*load only once;  415
416                              Var_Info = "&VAR. &FORMAT. " !! Label; end; 416
417   HaveData='1';          %*HaveData updated when 1 obs meets where cond; 417
418   output;                                        %*do until(EoF);end; 418
419  call symput('HAVEDATA',HaveData);                                  stop; 419
420                                                                      run; 420
421 %IF &HAVEDATA %THEN %DO;                                              421
422  proc APPEND base = INVALID                                            422
423              data = DETAILS;                                    %END; 423
424 run;%*.....................................................; %MEND; 424
425 ;/*TEST DATA *************** to enable, end this line with slash (/) **  425
426 *PROC CATALOG catalog=WORK.FORMATS kill;quit;%*zap previous;             426
427 %LET INVALID=INVALID;%*in either AUTOEXEC or LETFRMT;                   427
428 proc FORMAT;                                                           428
429  value one_3_    1,2,3         ='OK' other="&INVALID.";                429
430  value two_4_      2,3 , 4     ='OK' other="&INVALID.";                430
431  value $thre_5_     '3','4','5'='OK' other="&INVALID.";                431
432  value no_chk    1,2,3 , 4 , 5 ='OK';                                  432
433  value $notused             '5'='OK' other="&INVALID.";                433
434                                                                        434
435 DATA TESTNVLD;                                                        435
436 attrib IDChr    length= $ 3                 label = 'ID1Chr'          436
437        IDNmbr   length=   4                 label = 'ID1Num'          437
438        FormNmbr length= $ 2                 label = 'ID2'             438
439        A        length=   4 format=one_3_.  label = 'A: number 1'     439
440        B        length=   4 format=two_4_.  label = 'B: number 2'     440
441        C        length= $ 1 format=$thre_5_. label = 'C: char var'    441
442        I        length=   4 format=no_chk.  label = 'loop counter'    442
443        ;                                                               443
444 retain FormNmbr '01';                                                 444
445 do I = 1 to 5;  IdChr  = put(I,z3.);  IdNmbr =     I;                 445
446                 A      =     I;       B      = sum(I,2);             446
447                 C      = put(I,1.);   output;                    end; 447
448                                                                  stop; 448
449 %*Comparison report of formats in LIBRARY and FMTLIB;                  449
```

```
450 %INVALID(,LIBRARY = WORK                                                    450
451          ,FMTLIB  = WORK);                                                  451
452 %*EXPECTED REPORT:                                                         452
453 INVALID: compare formats in DATA:WORK._ALL_ and FMTLIB:WORK                 453
454 FORMAT      DATA    FMTLIB    OK   Comment                                  454
455 $NOTUSED      0        1      ..   no problem                              455
456 $THRE_5_      1        1      OK                                           456
457 NO_CHK        1        0      ??   no <other=INVALID>, is problem?         457
458 ONE_3_        1        1      OK                                           458
459 TWO_4_        1        1      OK                                           459
460 NOTE: only formats with CHK=OK will be used by this routine;              460
461                                                                            461
462 %INVALID(TESTNVLD                                                         462
463         ,IDLIST  =IdChr FormNmbr                                           463
464         ,LIBRARY =WORK                                                     464
465         ,FMTLIB  =WORK                                                     465
466         ,PRNTFILE="ZIReport.SAS"                                           466
467         );                                                                 467
468 %*one detail line from the detail, by identifier, report:                  468
469 %*EXPECTED OUTPUT:* if IDCHR = '00001' and FORMNMBR = '01' then do;        469
470 %*                     char    ^     ^ quoted                              470
471 %* summary written at end of detail report:                                471
472 ***smry :  Ids       Vars      Cell                                        472
473 *** Base:  5         7         35                                          473
474 *Invalid:  5         3         7                                           474
475 *** Pcnt:  100%      42.857%   20%                                         475
476 NOTE: re Cell Pcnt range: <1%==good, 1:2%==OK, >2%==trouble!;              476
477                                                                            477
478 %INVALID(TESTNVLD                                                         478
479         ,IDLIST  =IDNMBR FORMNMBR                                          479
480         ,LIBRARY =WORK                                                     480
481         ,FMTLIB  =WORK                                                     481
482         ,PRNTFILE='C:\TEMP\ZINVALID.SAS'                                   482
483         );%*NOTE hard-coded output fileref;                                483
484 %*EXPECTED OUTPUT:*if IDNMBR =  1  and FORMNMBR = '01' then do;            484
485 %*                    numeric  ^ ^ no quotes;                              485
486                                                                            486
487 %* Four Summary Reports:;                                                  487
488                                                                            488
489 %* ,SMRYIDS =1 *?print FREQ of IDS? *                                      489
490                                                                            490
491 invalid values in TESTNVLD Obs:5                                          491
492 summary: IDs listed invalid<5> / data<5> = 100%                           492
493                                                                            493
494 Obs    IDNMBR    FORMNMBR    COUNT    PERCENT                             494
495                                                                            495
496  1        1         01         1      14.2857                             496
497  2        2         01         1      14.2857                             497
498  3        3         01         1      14.2857                             498
499  4        4         01         2      28.5714                             499
500  5        5         01         2      28.5714                             500
501                              =====                                         501
502                                7;                                          502
503                                                                            503
504 %*,SMRYVARS=1 *?print FREQ of variables with invalid? *                     504
505                                                                            505
506 invalid values in TESTNVLD Obs:5                                          506
507 summary: Variables listed invalid<3> / data<7> = 42.857%                  507
508                                                                            508
509 Obs    VAR    COUNT    PERCENT                                            509
510                                                                            510
```

```
511  1      A       2      28.5714                                          511
512  2      B       3      42.8571                                          512
513  3      C       2      28.5714                                          513
514                =====                                                   514
515                  7;                                                    515
516                                                                        516
517 %* ,SMRYNAME=1 *?print detail  of variables, by name? *                 517
518 invalid values in TESTNVLD Obs:5                                        518
519 detail: by Var-Name invalid<7> / data<35> = 20%                         519
520                                                                        520
521    Var Format Label     IDNMBR  FORMNMBR  VALUECHR  VALUENUM  N       521
522                                                                        522
523 A ONE_3_ A: number 1       4       01        .          4     1         523
524                           5       01        .          5     1         524
525 ----------------------                                     -         525
526 A ONE_3_ A: number 1                                       2;        526
527                                                                        527
528 run;/*****************************************************************/ 528
```

Advanced Macro Topics

Steven First, Systems Seminar Consultants, Madison, WI

Abstract

The SAS macro language continues to be an integral part of the SAS system, and can be a wonderful tool or an overcomplicated solution. This paper will be of interest to macro users that would like to more fully understand the macro system.

This paper will discuss the following topics:

1. Macro data structures and timing
2. Local and Global referencing environments
3. Quoting
4. Macro and data step functions and call routines
5. Debugging and tracing techniques
6. Autocall facilities
7. Design considerations
8. Alternatives to SAS macros
9. Changes and enhancements with Version 7 and 8

Introduction

The SAS macro facility gives the SAS programmer some very interesting power and at the same time requires more understanding than the SAS data or PROC step. While it can make SAS jobs more robust and easier to code, it can also be easily overdone with the result being a more complex system than is sometimes necessary. This paper will explore some of the more advanced features of the macro facility, but at the same time strive to keep things as well documented and simple as possible.

Macro Data Structures And Timing

The SAS Macro facility is a second language that can be intermixed with normal SAS statements. Unlike normal SAS statements, the function of macro statements is to generate SAS statements or perhaps just part of SAS statements during the word scan and compile process.

Without macros, SAS programs are DATA and PROC steps. The program is scanned one statement at a time looking for the beginning of step (step boundary). When the beginning of step is found, all statements in the step are compiled one at a time until end of step is detected. When the end of step is found (the next step boundary), the previous step executes. This interleaving of compiling and execution can be confusing to many users. Other languages also have compile and execute phases, but typically those programs are similar to a single SAS step. Because in SAS a step is compiled, then executed, before the next step is compiled and executed, it is easy to see how the timing can be difficult to comprehend.

Again, most of the work of the macro facility is performed during word scan and compile, with the DATA or PROC step doing the work at step execution time.

SAS Step Boundaries

If it is important for the user to understand the timing of compile and execution, it is critical to define the start and end of a SAS step.
The beginning of steps are obviously the DATA or PROC statements. The ends of steps are a little more difficult to detect. The end of a step for our purposes is whenever SAS encounters another step boundary or end of job.

The following keywords are step boundaries to the SAS system:

DATA	ENDSAS
PROC	LINES
CARDS	LINES4
CARDS4	PARMCARDS
DATALINES	QUIT
DATALINES4	RUN

In the following program, the word scanner can only detect end of step when the next step starts. The last step will start compile, but will only execute if run in batch because there is no explicit end of step give. Interactive sessions will disply the message "PROC MEANS is RUNNING", but it really is in the word scan phase. This is not a very good way of coding a SAS job even though it may work in some instances.

```
data saleexps;            <--Step, start compile
 infile rawin;
 input name $1-10 division $12
       years 15-16 sales 19-25
       expense 27-34;
proc print data=saleexps; <--Step end, exec prev
                             Step start, compile
proc means data=saleexps; <--Step end, exec prev
 var sales expense;          Step start, compile
```

The RUN statement does not actually end some PROCs, but it does tell the wordscanner to stop compiling and 'RUN' the step. It is highly recommended that the RUN statement be coded, as there is then no question as to when compile finishes and execution starts.

```
data saleexps;            <--Step, start compile
 infile rawin;
 input name $1-10 division $12
       years 15-16 sales 19-25
       expense 27-34;
 run;                     <--Step end, exec prev

proc print data=saleexps; <--Step start, start
   compile
 run;                     <--Step end, exec prev
proc means data=saleexps;
 var sales expense;
 run;                     <--Step end, exec prev
```

It shold be noted that global statements such as title, footnote, options, libname etc. are compiled, and then executed immediately. Again by explicitly coding RUN statements, the programmer has absolute control over which steps global statements affect.

Timing with Macro Variables

As %LET statements define macro variables, and those variable are referenced by name and ampersand what are the timing considerations?

%LET and & are processed during the word scan and compile phases, which remember always precede the execution phase. The %LET statement below defines a macro variable called dsname which is then referenced many times later in the program. This is a very simple substitution and it works very well. In fact, quite often an automatic variable is defined by SAS and we don't even need to code the %LET.

```
%let dsname=saleexp;
data &dsname;            <--Step, start compile
 infile rawin;
 input name $1-10 division $12
       years 15-16 sales 19-25
       expense 27-34;
 run;                    <--Step end, exec prev

proc print data=&dsname; <--Step start, comp
 run;                    <--Step end, exec prev
proc means data=&dsname;
 var sales expense;
 run;                    <--Step end, exec prev
```

A Timing Error

The program below attempted to set a macro variable to either Hardware or Software based on the values read in Division. In this case there were only "H" division records, but yet when the title prints, "Software division" is displayed. Why?

```
data saleexps;
 input name $1-10 division $12 ;
 if division='H' then
    %let mdiv=Hardware;
 if division='S' then
    %let mdiv=Software;
 datalines;
Steve      H
Bob        H
;
run;
proc print data=saleexps;
 title "&mdiv division";
run;
```

The program has timing confused. The %LET statements set the mdiv values at COMPILE time and the first %LET statement moves Hardware to the macro variable, but then the second %LET statement moves Software into the same macro variable. The resulting value is then from the last %LET statement processed during wordscan. It is much later that the data step executes the IF statements which now effectively do nothing since %LET has no effect at data step execution time. I have seen this error in several programs where the timing wasn't understood completely.

Execution Time Macro Components

To solve the problem above, the programmer really needed to use a statement to set the macro variable's value at data step execution. The most commonly used routine is CALL SYMPUT. This statement creates a macro variable at data step EXECUTION time.

```
data saleexps;
 input name $1-10 division $12 ;
 if division='H' then
    call symput('mdiv', 'Hardware');
 if division='S' then
    call symput('mdiv', 'Software');
 datalines;
Steve      H
Bob        H
;
run;
proc print data=saleexps;
 title "&mdiv division";
run;
```

Macro variables created via CALL SYMPUT cannot be referenced with an & until a later step again because of timing.

```
data saleexps;
 input name $1-10 division $12 ;
 if division='H' then
    call symput('mdiv', 'Hardware');
 if division='S' then
    call symput('mdiv', 'Software');
 title "&mdiv division";
 datalines;
Ssteve      H
Bob        H
;
run;
proc print data=saleexps;
 run;
```

The reference to &mdiv occurs at wordscan time which again is before execution time when SYMPUT executes. In order to make the &mdiv resolve correctly, it must be referenced in a later step. This again underscores the importance of step boundaries.

Another Timing Error

Suppose we want to retrieve a macro variable value that was created in the same step. This might be because the user is interacting with the step, or perhaps we are using a stored compiled step or an SCL program.

```
data saleexps;
 input name $1-10 division $12 ;
 if division='H' then
    call symput('mdiv', 'Hardware');
 if division='S' then
    call symput('mdiv', 'Software');
 newvar="&mdiv";
 datalines;
Steve      H
Bob        H
;
run;
```

Again the reference to &mdiv would be attempted before it was created by SYMPUT, even though it appears later in the program. The run time retrieval routine is typically SYMGET which works fine.

```
data saleexps;
 input name $1-10 division $12 ;
 if division='H' then
    call symput('mdiv', 'Hardware');
 if division='S' then
    call symput('mdiv', 'Software');
 newvar=symget('mdiv');
 datalines;
Steve      H
Bob        H
;
run;
```

Macro Structures and Conventions

The structures and storage used by the macro facility are also located in separate areas from the SAS DATA and PROC steps. SAS observations are normally stored on disk or tape, and when processed by the DATA step, values are normally stored in the program data vector. In the DATA step, variables are referenced by name, and they are not quoted. Numeric constants are not quoted, but single or double quotes are used around character constants to differentiate them from numeric constants and variables.

In the macro language since everything is a character value, no quotes are needed, and if quotes are included they are usually treated like any other character. One exception is that macro variables are not resolved within single quotes and they are resolved within double quotes. The semicolon is also not needed as much in the macro lanquage as it is in SAS.

Macro variable names and their corresponding values are stored in memory locations called symbol tables and they are not retained beyond the current job. Macro variables are not normally named with a & when defined with a %LET statement, but are normally prefixed with & when referenced later. In the example below, the macro variable name (dsname) is a constant and the value (saleexps) is also a constant.

Example:

```
%let dsname=saleexps;
data &dsname;
etc.
```

There is no reason the variable name itself can't be a macro variable reference. In the example below the macro variable name includes an & which means that the name itself varies and will need to be resolved before creating the new variable.

Example:

```
              /* name dsname, value saleexps */
%let dsname=saleexps;

              /* name saleexps, value abc */
%let &dsname=abc;

             /* name saleexps, value abc */
data &sallexps;
etc.
```

Multiple Ampersands

The double ampersand (&&) is a special reference that always resolved into a single ampersand.

Example:

```
%let c=hallo;
%put &c &&c;
```

The macro processor resolves the first reference into hallo. In the second reference, the macro processor resolves && into & and the scans the letter c. Next, because the macro processor always rescans an item resulting from a resolution, it scans the remaining &c giving the same result hallo and displaying:

```
hallo hallo
```

Using a double ampersand in this case gives the same result as a single ampersand. Why then would we ever need multiple ampersands? One example is when we have a series of similarly named variables then end in consecutive numbers that we may want to index through.

For example, we have a series of county datasets that we would like to print with a macro loop. A very common technique is to store the names in separate variables along with a final variable containing the number of other variables, then loop through them.

Solution 1: Incorrect

```
%let cnt1=ashland;
%let cnt2=bayfield;
%let cnt3=washington;
%let totcnt=3;
%macro cntprt;
 %do I=1 %to 3;
   proc print data=&cnt&I;
   run;
%end;
%cntprt
```

The above fails, because there is no variable called cnt, so we have to delay the resolution with multiple ampersands. Using &&cnt&I resolves &&cnt to &cnt and &I to 1 on pass 1, then rescans the result (&cnt1) to resolve to ashland etc. This indirect referencing technique is used in many macros that use looping.

Solution2: Correct

```
%let cnt1=ashland;
%let cnt2=bayfield;
%let cnt3=washington;
%let totcnt=3;
%macro cntprt;
 %do I=1 %to 3;
   proc print data=&&cnt&I;
   run;
%end;
%cntprt
```

Triple Ampersands

Three ampersands can be used in the case where the value of one variable is the name of a second variable whose value you would like to retrieve.

Example:

```
%let c=data;
%let data=year2001;
data &&&c;
 etc.
```

During scan one, && resolves to &, &c resolves to data. Scan 2 resolves the remaining &data to year2001.
More than three ampersands can be used though this is done rarely. In any case the macro processor resolves two ampersands into a single ampersand and then rescans all text generated by the previous scan.

Macros Versus Macro Variables

SAS macros (macro programs) allow much more than simple substitution of values, they introduce logic. This can also introduce unneeded complication to the program if only substitution is needed.

The macros themselves go through the word scan process and are normally stored in a work area until invoked. Another way of saying this is that macros themselves are compiled and invoked (executed). Remember that the macro may generate steps that are compiled and executed as well.

Referencing Environments

As macros define their own macro variables, the concept of referencing environments is necessary. Every SAS program has one or more referencing environments. A referencing environment is the area in which a macro variable is stored and later retrieved. Referencing environments are especially important when using multiple macros that could conceivably use the same variable names for different values, and thus contaminate or destroy results.

Referencing environments are large areas surrounding progressively smaller areas. The largest and outermost area is the global area. The automatic SAS variables, those variables created outside of macros, as well as most macro variables created with SYMPUT exist in the global environment. Variables stored in the global environment are available anywhere in the SAS job.

Each macro that you invoke creates its own local referencing environment. The local referencing environment is empty until the macro creates a macro variable. This environment exists only while the macro is executing at which time the storage is freed to the system. Also any nested calls to macros will result in

nested local environments.

The environment for the currently existing macro is called the current macro environment.

How the Macro Processor Uses Referencing Environments

When creating macro variables, the macro facility tries to change any existing macro variable rather than creating a new one. If no existing variable is found, the macro processor creates the new variable in the current environment. The %LOCAL and %GLOBAL statements can be used by the programmer, to control these actions.

While this seems complicated, in practice it is not usually very difficult to deal with. In most cases, if all macros are defined in the same job, each macro creates it's own variables as the macro executes, and then those variables are deleted. When macros call one another is the case that usually causes problems.

If a calling macro and the called macro accidentally use the same variable names for different values, there is a good chance that the inner macro may contaminate the value of the outer macro. To prevent this the %LOCAL statement can tell the macro facility to define it in the local environment, regardless of where else it may be defined. It is always good practice to define all variables in a macro as local unless there is some reason to not do so. Unfortunately, almost nobody uses %LOCAL even though they probably should. This is especially important to utility-type macros that could be called by other macros.

Using %LOCAL isn't always the answer to all problems, though. Suppose we want to write a utility macro that performs some task, and then reports the success to the caller through some sort of macro variable used as a return code. Without taking special action that variable would only be defined in the local environment, and as such the caller would have no access to it. In this case, the only option is to define the return code variable as global and hope that the caller or other macros are not using the same named variable.

Obviously well chosen names and good documentation can help tremendously when building a library of macros. Without good conventions and documentation, utility macros can be very difficult to use and understand, and some very strange results can occur.

Macro Quoting

All programming languages need some way to differentiate that certain characters are to be treated as a text value instead of some instruction or operator used in the language. In the data and PROC steps, and in most other programming languages, a single or double quote mark serves this role.

In the example below when var1 is not quoted, it refers to the name of an existing data step variable. When it is quoted it is simply four constant characters. Likewise the semi-colon marks the end of all SAS statements, but in addition when Z is set to the quoted semicolon, the quotes cause the system to treat the semicolon as it would any other character.

```
data temp;
  set ds1;
  x=var1;
  y='var1';
  z=";";
run;
```

Obviously the macro facility has special symbols that may need similar treatment. However the macro facility does not use the single or double quote character to mask these characters. Instead it uses a variety of functions to do what quotes do in other languages, thus the name 'macro quoting'. Another way of stating this is that the actions that you take to cause the macro facility to treat a certain character as text rather than part the macro language is called "macro quoting". Similarly when the literature mentions that the "result is quoted", it does not mean that actual quote marks are inserted, but rather the result is treated like text and the meaning of the special characters are removed.

Different Kinds of Quoting Functions

Compilation Functions

There are functions that cause the macro facility to treat items as text during macro compilation or when used in open code. %STR and %NRSTR are examples. %STR removes meaning of the following: ;'+-*/**~=<>,*LT LE EQ NE GT OR AND & trailing and leading blanks. %NRSTR (no rescan string) processes all of the above and also prevents rescanning (ignore meaning of & %).

Examples:

```
%let name=%str(proc print;run;);
%let name=%str(    &company);
%let desc=%nrstr(    %of total report);
```

Execution Functions

There are functions cause the macro facility to treat items as text during macro execution as well. %QUOTE and %NRQUOTE operate at execution time and remove meaning of most characters in the result of the macro call. %BQUOTE and %NRBQUOTE should be used if the value contains unmatched quotes or parentheses.

Examples:

```
%macro getst(state,employee);
%if quote(&state) = %str(NE) %then
   %do;
      %let longst=Nebraska;
  %end;
  %put hello &employee from &longst;
%mend;
```

%getstate(NE,O'brien)

Functions to Prevent Resolution

%SUPERQ is a function that will prevent starting any resolution of macro variables. This is most needed in windowing operations such as SAS/AF, %WINDOW, or SAS/INTRNET screens, or after CALL SYMPUT if there is any chance the input value could contain an ampersand or percent sign. Other functions do not work as well in this case, and would issue warnings and recursion messages.

Example:

```
data _null_;
   call symput('mv1','Bob&Fred %macro report');
   run;
%let mv2=%superq(mv1);
%put mv2=&mv2;
```

The Result:

```
mv2=Bob&Fred %macro report
```

Unquoting

The effects of quoting can be removed if desired by the %UNQUOTE function. In the example below var2 is not resolved but using %UNQUOTE later allows the resolution to take place.

Example:

```
%let mv1= hallo;
%let mv2=%nrstr(&mv1);
%let mv3=%unquote(&mv2);
%put mv1=&mv1 mv2=&mv2 mv3=&mv3;
```

The Result:

```
mv1=hallo mv2=&mv1 mv3=hallo;
```

Other Macro Functions

There are several other character functions that do various manipulations on macro values. Most of these functions are very similar to functions in the data step and perform the same way to substring data, left align, right align, parse words etc. In many cases there is a quoted version of the function where the name starts with a Q (ex. %QSUBSTR). These functions are sometimes more difficult to code and debug than counterparts in the DATA step mostly because the DATA step is a more comprehensive and robust language.

Using Data Step Functions Within Macros

The macro facility can have access to the DATA steps function library of over 300 functions via the %SYSFUNC macro function. This can be extremely handy to include any of the logic from existing functions. %SYSFUNC can also apply a format to the result from the function. There are some functions such as the LAG function that cannot be used because LAG needs to read prior records in a data step which might not exist in a macro application.

Example: Extract today's date, format it as a worddate, store the final result in a macro variable.

```
%let mydate=%sysfunc(date(),worddate.);
%put &mydate;
```

Displays:

```
January 15, 2001
```

Debugging and Tracing Techniques

There are essentially five debugging tools in the macro facility.

1. %PUT
2. SYMBOLGEN
3. MLOGIC
4. MPRINT
5. MFILE

The %PUT statement is probably the best and simplest method of displaying values at word scan time. %PUT can display text or macro variable references and calls. Like the PUT statement from the DATA step, %PUT can be placed at strategic spots in your program to display debugging information. Like other items in the macro facility, quotes are not needed to display text.

Example:

```
%let var1=Stevo;
%put ***** var1=&var1;
```

Displays:

```
**** var1=Stevo
```

A fairly recent addition is parameters to %PUT that not only display a single variable, but can show several.

ALL	Shows all variables and there respective symbol table
AUTOMATIC	shows only the system variables
USER	displays only user variables
LOCAL	displays only local variables
GLOBAL	displays only global variables

Using _ALL_ is really the only practical way to determine whether a variable is defined locally, globally or both. This was requested for many years, and it's now available.

The other four debugging tools are system options.
SYMBOLGEN shows macro variables as they are being resolved.
MLOGIC displays decisions and looping that the system makes with %DO, %IF etc.
MPRINT displays the generated code sent to the SAS compiler.
MFILE routes the MPRINT results to a separate file.

MFILE can be useful to give us an answer to a very difficult problem. Suppose the following macro is submitted with a semicolon missing after the VAR statement. The statement number displayed in the log is the line number of the statement that called the macro. This isn't overly useful, since the macro could contain or generate thousands of lines, and we have an error somewhere in it.

```
1    options mprint nomfile;
2     %macro printmac(msasds=invoice);
3      proc print data=&msasds;
4       by company;
5       id days;
6       var rate age    /* note missing ;*/
7       title "listing of &msasds";
8      run;
9     %mend printmac;
10
11    %printmac(msasds=invoice)
MPRINT(PRINTMAC):   proc print data=invoice;
MPRINT(PRINTMAC):   by company;
MPRINT(PRINTMAC):   id days;
NOTE: Line generated by the invoked macro
"PRINTMAC".
11      proc print data=&msasds;    by company;
id days;    var rate age             title
"listing of &msasds";   run;

-

200
ERROR 200-322: The symbol is not recognized and
will be ignored.

NOTE: Line generated by the macro variable
"MSASDS".
11   "listing of invoice
      --------------------
      22
MPRINT(PRINTMAC):   var rate age title "listing
of invoice" run;

ERROR 22-322: Syntax error, expecting one of the
following: a name, ;, -, :, _ALL_, _CHARACTER_,
_CHAR_, _NUMERIC_.
```

As a last resort, the generated code could be routed to a file, manually included and run. Since the code no longer contains any macro statements or conventions, statement numbers will be more meaningful.

```
filename mprint 'c:\temp\pmaccap.sas';
options mprint mfile;
 %macro printmac(msasds=invoice);
  proc print data=&msasds;
   by company;
   id days;
   var rate age     /* note missing ;*/
```

```
   title "listing of &msasds";
  run;
 %mend printmac;

 %printmac(msasds=invoice)

options nomfile;
```

The captured program looks like the next few lines from the SAS log, where it easier to find the offending statement.

```
53   proc print data=invoice;
54   by company;
55   id days;
56   var rate age title "listing of invoice"
run;
                         ------------------- ---
                         22                  202
ERROR 22-322: Syntax error, expecting one of the
following: a name, ;, -, :, _ALL_, _CHARACTER_,
_CHAR_, _NUMERIC_.
```

The SAS Autocall Facility

There are five methods of making a macro available to your SAS job:

1. Define a macro in the same program that uses it.
2. Use %INCLUDE to include macros stored in external files.
3. Use the INCLUDE display manager command to manually include the macro.
4. Use the autocall facility to search a predefined macro library.
5. Use the Stored Compiled Macro Facility.

In my opinion the first three methods work fine for many systems, and the compiled macro facility is probably not worth the bother. While it might save a slight amount of word scanning, the user now has to manage both compiled code and source code which can be very difficult to deal with especially with new releases.

A good option to handle production macros that will be used by many users is to store them in an autocall library. By making this library available to all users, they can call macros without including them first.

Using the autocall facility requires five steps:

1. Create a macros source library appropriate for your operating system.
2. Place each macro as a separate member in the library. The member name must be the same as the macro name
3. Associate the fileref SASAUTOS with the library.
4. Turn on the MAUTOSOURCE SAS system option.
5. Invoke any stored macro.

Invoking a macro that has NOT been defined in the current session causes the macro facility to:

1. Search the autocall library for the member named the same as the invoked macro.
2. Issue an error message if the macro is not found.
3. Automatically include the macro source if found.
4. Submit the macro.
5. Call the macro.

The autocall facility is a PDS under OS/390 and is a directory in the directory based systems. You can also concatenate several libraries and in fact several macros are shipped with the SAS system and the user only needs to call them as they would a macro function.

Examples of SAS-Institute Autocall Macros (partial list)

```
%COMPSTOR(arg)          compiles autocall members
%CMPRES(arg)            remove multiple blanks
%DATATYP(arg)           determine argument data
                        type
%LEFT(arg)              remove leading blanks
%LOWCASE(arg)           translate to lower case
%MDARRY                 simulate multi-dimension
                        arrays
%QCMPPRES(arg)          quoted form of %CMPRES
%QLEFT(arg)             quoted form of %LEFT
%QLOWCASE(arg)          quoted form of %LOWCASE
%QTRIM(arg)             quoted form of %TRIM
%SUPERQ(variable)       remove meaning of all
                        special characters
%SYSGET(arg)            returns oper system vars
%SYSRC(arg)             translates numerics to
                        mnemonics
%TRIM(arg)              trims trailing blanks from
                        arguments value
```

An example of using an autocall macro:

%TRIM can be used to trim trailing blanks from macro values.

```
%LET X=%STR(MADISON   );
%LET Y=%TRIM(&X);
%PUT ---&X--- ---&Y---;
```

Generates:

```
---MADISON   --- ---MADISON---
```

Design Considerations

Since the macro system has a unique purpose, it stands to reason that design may be unique.

It is my opinion that the best systems:

1. Minimize macros
2. Clearly document macros and their purpose
3. Display source of all modules at least while testing.
4. Keep things simple

Minimizing macros

The are many cases where macros have been written where a SAS option or DATA statement may work much better. Examples are the CNTLIN option for building formats, the #BYVAL variable for displaying by variables in titles. Macros are simply more difficult to work with if there is another reasonable alternative.

Below is a pair of macros that create titles. The first technique uses only macro statements and the second one does most of the work in the DATA step. Because of the robustness of the DATA step and the limitations of the macro language, the second approach may be simpler, though this approach obviously wont work for all cases.

In both cases the macro call below generates the title following the call.

```
%ssctitlm(1,
           3,'left edge',
          20,'sample center',
          40,"date run:&sysdate")
```

Generates

```
title1
"  left edge        sample center
            date     run:13DEC00";
```

Solution one done completely with macros:

```
%macro ssctitlm(mtitlno,mcol1, mtext1,
                mcol2, mtext2,
                mcol3, mtext3);
/*******************************************/
/* macro ssctitlm                          */
/*  purpose:  create sas title variables   */
/*            forprocs                     */
/*    input:  title number, starting col,  */
/*            text up to 3 segments of title. */
/*   output:  title vars that can be used by  */
/*            sas                          */
/*******************************************/
/* need data step logic to build data step    */
/* vars, put in macro variables at end.       */
/*******************************************/
%let noblank=%eval(&mcol1-1);  /* # blks left */
%let mtext=%str();             /* set to null */
%do i=1 %to &noblank;          /* do # blanks */
 %let mtext=%str(&mtext)%str( ); /* insert blk*/
%end;                          /* end of loop */

%let tlen=%eval(%length(&mtext1)-2); /* no 's */
                               /* strip out 's*/
%let mtext1=%substr(%str(&mtext1),2,&tlen);
%let mtext=%str(&mtext)%str(&mtext1); /* app  */
                               /* to mtext    */
                               /*# bls text2  */
%let noblank=%eval(&mcol2-1-%length(&mtext));
%do i=1 %to &noblank;          /* do # blanks */
 %let mtext=%str(&mtext)%str( );   /* insert  */
                               /*  blanks     */
%end;                          /* end blk loop*/

%let tlen=%eval(%length(&mtext2)-2); /* no 's */
                               /* strip 's    */
%let mtext2=%substr(%str(&mtext2),2,&tlen);
%let mtext=%str(&mtext)%str(&mtext2);  /* app */
                                  /* to text  */
                               /*# bls mtext3 */
%let noblank=%eval(&mcol3-1-%length(&mtext));
%do i=1 %to &noblank;          /* do # blks   */
 %let mtext=%str(&mtext)%str( );/* insert blk */
%end;                          /* end blk loop*/

%let tlen=%eval(%length(&mtext3)-2);
                              /* no 's       */
                               /* strip 's    */
%let mtext3=%substr(%str(&mtext3),2,&tlen);
%let mtext=%str(&mtext)%str(&mtext3);
                             /* app to mtext*/

 title&mtitlno "&mtext";       /* make title  */
/***********   end of macro ssctitlm
***************/
 %mend ssctitlm;
```

The following macro generated the same title statement, but with a lot less coding effort.

```
 %macro ssctitle(mtitlno,mcol1,mtext1,
                 mcol2,mtext2,mcol3,mtext3);
/*******************************************/
  /* macro ssctitle                        */
  /*  purpose:  create sas title variables */
  /*            for procs                  */
  /*    input:  title number, starting col, */
  /*            text for 3 segments of title. */
  /*   output:  title vars that can be used by*/
  /*            sas.                       */
/*******************************************/
  /* need data step logic to build data step  */
  /* vars put into macro variables at end.    */
/*******************************************/
data _null_;                   /* use data step*/
  length text $ 160;           /* ds var mx len*/
  text=' ';                    /* blank it out */
  substr(text,&mcol1)=&mtext1;/* first text   */
  substr(text,&mcol2)=&mtext2;/* second text  */
  substr(text,&mcol3)=&mtext3;/* third text   */
  call symput("mtitle&mtitlno",text);
                               /* ds vars->mac */
 run;                          /* end of step  */
 title&mtitlno "&&mtitle&mtitlno";
                               /* move to title*/
  /********** end of macro ssctitle ***********/
 %mend ssctitle;
```

Clearly Documenting and Keeping It Simple

A recent program that a student offered used all of the below techniques:

1. Autocalled macros
2. Generated Code to a file, then %included it back to run.
3. %included macros
4. Inline macros
5. Nested macros
6. All source display (2000 lines) turned off
7. No commenting.

While any of the above techniques are fine, having so many pieces and no documentation really makes the job much more difficult than a few, well-documented techniques.

Changes and Enhancements

The major changes to the macro system is 32 character names and the %PUT _All_ Statement mentioned earlier. Below are some of the other changes implemented.

The following automatic macro variables are available in all operating environments.

SYSCC — contains the current condition code that SAS returns to your operating environment (the operating environment condition code).

SYSCHARWIDTH — contains the character width value.

SYSDATE9 — contains a SAS date value in DATE9. format, which displays a 2-digit date, the first three letters of the month name, and a 4-digit year.

SYSDMG — contains a return code that reflects an action taken on a damaged data set.

SYSPROCESSID — contains the process ID of the current SAS process.

SYSPROCESSNAME — contains the process name of the current SAS process.

SYSSTARTID — contains the identification number that was generated by the last STARTSAS statement.

SYSSTARTNAME — contains the process name that was generated by the last STARTSAS statement.

SYSUSERID contains the user ID or login of the current SAS process.

%PUT In Version 8, the %PUT statement has been enhanced. It displays text in different colors to generate messages that look like SAS-generated ERROR, NOTE, and WARNING messages.

%SYSLPUT In Version 8, this new macro statement creates a new macro variable or modifies the value of an existing macro variable on a remote host or server.

The following new macro statement invokes a SAS CALL routine:

%SYSCALL

The following macro functions are new:

%SYSEVALF evaluates arithmetic and logical expressions using floating-point arithmetic.

%SYSFUNC and %QSYSFUNC
execute SAS functions or user-written functions

Conclusion

The SAS macro facility continues to have incremental improvements with each new SAS release, and with when used properly, it gives the tremendous power and capabilities.

Contact Information

Your comments and questions are valued and encouraged. Contact the author at:

Steven First
Systems Seminar Consultants
2997 Yarmouth Greenway Drive
Madison, WI 53716
608 278-9964 x 302 voice
608 278-0065 fax
sfirst@sys-seminar.com
www.sys-seminar.com

Paper P813

Top-Down Programming with SAS® Macros

Edward Heaton, Westat, Rockville, MD

ABSTRACT

Structured, top-down programming techniques are not intuitively obvious in the SAS language, but macros can be used to approximate a top-down structure. This can lead to programs that are more versatile, more robust, and generally easier to develop. This presentation will review a SAS macro that reads elements from a difficult-to-read ASCII text file that is typically used to control the processing for a COBOL job. Along the way, we will look at the program structure and review the engineering decisions that allow the code to be developed in an organized and robust manner.

This presentation will demonstrate methods for macro testing and debugging, as well as various features of the SAS Macro Facility.

INTRODUCTION

Top-down programming can make your programs much more easy to understand, develop, debug, and modify. This paper will demonstrate top-down programming while creating a utility to facilitate the creation of value-label formats.

One task that greatly benefits from an orderly, top-down structure is the reading of a complex external file. We will develop a SAS macro that reads a meta-data file to obtain variable and format information.

Westat developed a COBOL program to read a free-format text file and write files that facilitate many of the steps of survey processing. That text file (hereafter called a source file) is a useful first-step in creating value labels. In this presentation we will create an external SAS macro that can be called from a SAS autocall library. The macro will read the source file and will write a SAS data set. That data set needs to contain all the information necessary to create value-label formats. While we develop this macro, we want to write code that will be easily adaptable to other stages of our survey processing cycle. We will not address those other stages, but certain programming techniques help to assure that adaptability.

FILE FORMAT

Let's start by defining the source file. Perhaps the best approach is to look at the specifications for creating the source file.

1. The file is free-format; words do not need to begin and end in specific columns.
2. The statements shall be typed in lines of 100 characters or fewer, including spaces, but they may span several lines.
3. A virgule (*/*) marks the beginning of a line continuation. Line continuations are for the descriptive parts of the statement, such as the wording of a question or the description of a value code.
4. Each statement begins with a key letter that identifies the statement type:
   - P Parameter – We will not process this statement.
   - H Header – We will not process this statement.
   - V Variable – Variable statements define properties of the variable.
     `V name width type`
     E.g.: To define variable Q1 as two columns wide containing numeric data, the source file will have the following statement.
     `V Q1 02 N`
     For this task, we will want to read the name of the variable and its data type.
   - Q Question – If we were creating variable labels, we would want to read the wording of the question. We are not so we will not use this statement.
   - C Code – The code statement defines the values of the variable defined in the preceding **V** statement. A **+** indicates a "missing" value. You can link a **C** statement to an **S** statement via one or more asterisks (*).
     `C [********] value = text`
     E.g.:
     ```
     C ++ = inapplicable
     C 01-70 = number of years
     C 98 = don't know
     C 99 = not ascertained
     ```
   - R Remark – This is a comment statement and has no bearing on our task.
   - S Skip pattern – Skip patterns tell us which variables to skip based on the values of the variable defined in the preceding C statements. We are not going to deal with skip statements; they have no bearing on this task.
   - A Abbreviation – The A statement will not be processed by our macro.

Our paper will work with the variable and code statements only. The source file might look something like the following:

```
P RECLEN 200
H 1 U.S. Department of Education
H 2 National Center for Educational
  / Statistics
H 3 Internet Access in U.S. Public Schools,
  / Fall 2000
V QA 01 A
Q What is the title/position of the
/ Respondent?
C 1 = Technology Coordinator
C 2 = Library/Media Specialist
C 9 = Not Ascertained
V QB 01 A
Q Does the respondent have email?
C 1 = Yes
C 2 = No
C 9 = Not Ascertained
V Q1 04 N
Q What is the total number of instructional
/ rooms in your school?  (Include all rooms
/ used for any instructional purposes:
```

```
/ classrooms, computer labs and other labs,
/ library/media centers, etc.)
C 0001-0250 = Total number of Instructional
            / Rooms
C 9999 = Not Ascertained
V Q2A 04 N
Q How many computers are there in your
/ school?  (Count all computers, including
/ those used by administrators, teachers, and
/ students.)
C *    0000 = No computers in school (Skip to
            / Q13.)
C 0001-0550 = Total number of computers in
            / school
C      9999 = Not Ascertained
S * Skip Q2B - Q12DC and code as +
…
```

THE PLAN OF ATTACK

We will create a SAS data set that can be written to an MS Access® database. The **Label** field of that database will be edited to and then used for value labels – the wording of value labels is usually done by non-programmers at our site. When all the labels have been updated, we use PROC DATASETS to attach the labels to the variables. The code to write the SAS data to an existing MS Access database will look something like the following. (For this example, the **%CreateValueLabelsDataSet** macro is stored in the **Macros** sub-folder. The source file is in the same directory as this job, and it is called **codebook.dat**. We are creating a SAS data set called **ValueLabels** and an MS Access table called **ValueLabels**.)

```
Options sasAutos= ( "Macros" sasAutos ) ;

LibName labels odbc complete=
   "dsn=MS Access 97 Database;
    dbq=EasyLabels.mdb"
;

%CreateValueLabelsDataSet(
   inFile="codebook.dat"
 , cntlOut= ValueLabels
)

/* Write data to Access. */
Data labels.ValueLabels ;
   Set ValueLabels (
      rename= ( Label =LongLabel )
   ) ;
   Length Label $40 ;
   Label = LongLabel ;
Run ;

LibName labels clear ;
```

The **%CreateValueLabelsDataSet** macro read the source file and wrote the SAS data set. Then we wrote that data to MS Access, in effect shortening the length of the **Label** variable to 40 characters and creating a variable called **LongLabel**. **LongLabel** contains the entire text part of the code statement. When the MS Access table has been reviewed and updated we will use the data to create value label formats and attach those value label formats to our SAS data set with code similar to the following. By referring to SASAUTOS in the SASAUTOS= option, you make sure that all of the standard SAS autocall libraries are included.

```
LibName project "Library" ;

Options
   fmtSearch= ( project )
   sasAutos= ( "Macros" sasAutos )
;

LibName labels odbc complete=
   "dsn=MS Access 97 Database;
    dbq=EasyLabels.mdb"
;

Proc format
   library=project
   cntlIn=labels.ValueLabels( drop=
      VarName LongLabel
   )
   fmtLib
;
Run ;

/* Create a variable-format list as a macro
   variable. */
Proc sql noPrint ;
   Select
      trim(VarName) || " " ||
      trim(FmtName) || "."
      into :FormatString separated by " "
      from labels.ValueLabels
   ;
Quit ;

/* Apply the variable-format list to the data
   set. */
Proc dataSets ;
   Modify project.OurData ;
      Format &FormatString ;
   Run ;
Quit ;

LibName project clear ;
```

As you can see, these jobs are relatively short and simple. They could, however, be developed into macros to make the task even more simple.

EXECUTION OF THE PLAN

STANDARD DOCUMENTATION

Let's start with the standard job documentation. I use a template that prompts me for all the information that I typically want in my header.

```
/*****************************************
MACRO:

OBJECTIVE:

PROGRAMMER:
   Ed Heaton, Senior Systems Analyst,
   Westat,
   1550 Research Boulevard, Room 2018,
   Rockville, MD 20850-3159
   Voice: (301) 610-4818
   Fax:   (301) 294-3992
   mailto:EdwardHeaton@Westat.com
   http://www.Westat.com

DETAILS:

INPUT:

OUTPUT:

STORAGE:

AUDIT TRAIL:
   yyyymmdd EH : Developed macro.
*****************************************/
%Macro MName ( debugging=0 ) ;
```

```
   %If &debugging %then %do ;
      %TurnDebugOptions( ON )
   %End ;

/* SAS code and calls to other macros */

   %If &debugging %then %do ;
      %TurnDebugOptions( OFF )
   %End ;
 %Mend MName ;
/****************************************/
```

Maybe I need to say something about the **debugging=** parameter. I include it whenever I develop a macro. It is used for, as you guessed, debugging. The default value is zero. I pass **debugging=1** if I want to debug the macro. Then I can use this parameter to do things that I want to do only when I am in debugging mode. We will see examples of this feature in this paper. Even if I do not have immediate plans for the **debugging=** parameter, I include it. Then it is available for future use if I find a problem later on.

THE TURNDEBUGOPTIONS MACRO

When in debugging mode (called with **debugging=1**), a macro calls another macro called **%TurnDebugOptions**. That macro will either set or reset certain systems options that write copious amounts of information to the system log, depending on the value passed to the macro. The macro might be useful to you, so let's take a few minutes and look at it.

```
/*****************************************
MACRO: TurnDebugOptions

OBJECTIVE:
   This macro will turn macro debugging
   options ON or OFF.  To turn them on, call
   this macro passing in ON.  This macro will
   first retrieve original values of SAS
   system options MPRINT, SOURCE, NOTES,
   SYMBOLGEN, and MLOGIC.  It will then set
   the system options to MPRINT, MLOGIC,
   SOURCE, NOTES, and SYMBOLGEN so that we
   can follow the workings of macros.  To
   turn off the debugging options and restore
   the original SAS system option values,
   call this macro with the parameter set to
   OFF.

USAGE:
   The most common usage of this macro will
   be to add the DEBUGGING=0 parameter to
   your macro statements.  Then, if
   DEBUGGING=1 is passed in to your macro,
   call this macro passing "ON" and, at the
   end of your macro, call it again passing
   "OFF".  E.g.:
      %Macro Sample ( debugging=0 ) ;
         %If &debugging %then %do ;
            %TurnDebugOptions( ON )
         %End ;
         Macro Code
         %If &debugging %then %do ;
            %TurnDebugOptions( OFF )
         %End ;
      %Mend Sample ;

PROGRAMMER:
   …

STORAGE: …

AUDIT TRAIL:
   20000613 EH : Wrote code combining the …
   20000614 EH : Added code to allow …
*****************************************/
%Macro TurnDebugOptions ( switch ) ;
   %If ( %upCase(&switch) eq ON ) %then %do ;
      %Global
         _mPrint
         _mLogic
         _source
         _notes
         _symbolGen
      ;
      %Let _mPrint =
         %sysFunc( getOption( mPrint ) )
      ;
      %Let _mLogic =
         %sysFunc( getOption( mLogic ) )
      ;
      %Let _source =
         %sysFunc( getOption( source ) )
      ;
      %Let _notes =
         %sysFunc( getOption( notes ) )
      ;
      %Let _symbolGen =
         %sysFunc( getOption( symbolGen ) )
      ;
      Options
         mPrint
         mLogic
         source
         notes
         symbolGen
      ;
   %End ;
   %Else %if (
      %upCase( &switch ) eq OFF
   ) %then Options
      &_source
      &_notes
      &_mLogic
      &_mPrint
      &_symbolGen
   ;
   %Else %put
      TurnDebugOptions must pass ON or OFF.
   ;
%Mend TurnDebugOptions ;
/****************************************/
```

So you see, if a calling macro passes **ON**, the **%TurnDebugOptions** macro checks the current settings of five systems options using the GETOPTION function and then writes that value to macro variables using **%SYSFUNC**. If **OFF** is passed, the macro uses the systems options settings that were stored in these macro variables to restore the options to the original settings. If the **%TurnDebugOptions** macro is called twice passing **ON** without calling it passing **OFF**, the original systems settings will be lost. It is the responsibility of the programmer to avoid this situation.

TOP-DOWN PROGRAMMING – THE TOP

Let's define the steps to create a data set of value labels. The **%CreateValueLabelsDataSet** macro exists in a file called **CreateValueLabelsDataSet.sas**. Let's define a *top-level* macro as a macro that exists in a file of the same name.

```
%Macro CreateValueLabelsDataSet (
   inFile=
 , cntlOut=
 , debugging=0
) ;
   %If &debugging %then %do ;
      %TurnDebugOptions( ON )
   %End ;
```

```
   /* Create a SAS data set that has all the
      data that we need. */

   %If &debugging %then %do ;
      %TurnDebugOptions( OFF )
   %End ;
%Mend CreateValueLabelsDataSet ;
```

Whew, that was easy – just one step! Is that all there is to it? Well, yes; but the devil *is* in the details. Let's create the DATA step. Note the macro %IF statement embedded in the DATA statement below. I typically start all temporary variable names with an underscore. That way, I can drop all of them with DROP=_: (The colon is a wild card, like the asterisk in Windows.). If I am not in debugging mode, this will drop all variables that start with an underscore.

```
Data &cntlOut (
   %If not &debugging %then drop=_: ;
) ;

   %_DefineVariables( debugging=&debugging )

   Retain VarName FmtName Type ;

   InFile &inFile truncOver end=eof ;

   %_ReadOneStatement(
      cntlOut=&cntlOut
    , debugging=&debugging
   )

   If ( upCase( _StatementType_ ) eq "C" )
      then output &cntlOut
   ;

Run ;
```

We want to define our variables (the ones that we want to keep). I almost always find this a useful first step. We retained the **VarName**, **FmtName**, and **Type** variables because there are usually multiple **C** (case) statements per variable and we want to write an observation for each **C** statement. Each observation needs the variable name, the format name, and the data type. Of course, we need to read the source file. The output data set needs data for the **VarName**, **FmtName**, **Start**, **End**, **Label**, and **Type** variables.

After the DATA step, let's include code to print the data whenever we are in the debugging mode. We can print the whole data set; it shouldn't be too long. But let's print only the first 40 characters of **Label** because that value might be quite long.

```
%If &debugging %then %do ;

   Title4 "&cntlOut" ;

      Proc print data=&cntlOut ;
         Format Label $40. ;
      Run ;

   Title4 ;

%End ;
```

SUPPORTING MACROS – THE NEXT LEVEL

Now we need to create the two macros that we called. Let's define a *supporting* macro as one that exists in the same *.SAS file as a *top-level* macro and is called by another macro in that file. In keeping with the top-down concept, let's define *supporting* macros below the *top-level* macro. I always start a *supporting* macro with an underscore for two reasons: (1) the macro name is more unlikely to be used in the job that calls this *top-level* macro; and (2) it will be easier to explain to the users, when its name appears in the SAS log, that it is a *supporting* macro and that the users will not find a *.SAS file by that name. Let's create the macro that defines the variables. I have three criteria for deciding to create a macro rather than just including the code in the calling job.

1. Is the code used more than once? Whenever I have the same code in two different places, then I will surely, at some point, update it in only one of those places. Then my code will either not work or, worse, will work intermittently based on the data.
2. Will the code make the program block too long? I find it hard to visualize code when I can't see it from start to end on one page or screen. I.e.: I need to be able to see the start and end of a DO block at the same time. Ideally, I can see the entire DATA step, SQL statement, or PROC as a unit. I would rather see place-holders for code (i.e.; macro calls) than not be able to see both the start and the end. My rule of thumb is that I don't like any macro or program step to be over a half a page long if I can easily keep it shorter.
3. If I don't know how to write the code, I can simply insert a descriptive macro call and go on with the code that I do know how to write. (Often, coding tasks that I do know how to code will help me to see the algorithm for tasks that I didn't understand.) Then I can go back and create the macro that I called. (Sometimes, we need output from a macro but do not know how to get the values from the data. We can use the **debugging=1** parameter to hard-code values in the otherwise empty macro. Then we can continue with development of the job at hand and return to the undeveloped macro when we understand more.)

DEFINE VARIABLES

The **%_DefineVariables** macro will simply make the **DATA** step short enough to see in its entirety.

```
%Macro _DefineVariables ( debugging=0 ) ;

   Attrib
      VarName
         label= "Variable Name"
         length= $  6
      FmtName
         label= "Format Name"
         length= $  8
      Start
         label= "Starting Value for Format"
         length= $ 16
      End
         label= "Ending Value for Format"
         length= $ 16
      Label
         label= "Format Value Label"
         length= $200
      Type
         label= "Type of Format"
         length= $  1
   ;

%Mend _DefineVariables ;
```

Why did we set the length of the variable name to six characters? SAS allows 32 characters. The problem is with the format name; SAS still allows only 8 characters here. To keep our process algorithmic, I want to systematically create a format name with the variable name as its base. Variable names can end with a digit; format names cannot. So I want to append an eff (**F**) onto the end of the variable name. (E.g.: If the variable name is **Q59a12**, I want the format name to be **Q59a12F**.) What if the variable is type character? Then the format name needs to start with a dollar sign ($). (E.g.: **$Q59a12F**.) So, you see, a 6-character variable name can become an 8-character format name.

READ ONE STATEMENT

Now we will read the values for the needed variables and output the data set. However, given the nature of our input file, this seems like a big task. Let's break it down. We saw that the source file has seven different kinds of statements. We need to determine the type of statement and process it. Let's read the first non-blank character, call it the statement type, and hold the record. Since the source-file statements can span lines, we need to hold the record with a trailing @@. Then we will use the SELECT statement to set our course.

We need only process the **V** and **C** statements; a **V** statement will provide a variable's name and its data type; and a **C** statement will provide a value label. The other statements will be skipped. However, we will want to keep the select statement very general because we will want to expand our macro to get variable labels from the source file and , maybe, to collect skip-pattern information so that we can use the data to create a data dictionary. We don't want to significantly change this macro once it works correctly, so let's make sure it is easily expandable.

```
%Macro _ReadOneStatement ( debugging=0 ) ;

   Input @1 _StatementType_ : $1. @@ ;

   Select ( _StatementType_ ) ;
      When ( "P" ) do ;
         %_ReadPStatement(
            debugging=&debugging
         )
      End ;
      When ( "H" ) do ;
         %_ReadHStatement(
            debugging=&debugging
         )
      End ;
      When ( "V" ) do ;
         %_ReadVStatement(
            debugging=&debugging
         )
      End ;
      When ( "Q" ) do ;
         %_ReadQStatement(
            debugging=&debugging
         )
      End ;
      When ( "C" ) do ;
         %_ReadCStatement(
            debugging=&debugging
         )
      End ;
      When ( "R" ) do ;
         %_ReadRStatement(
            debugging=&debugging
         )
      End ;
      When ( "S" ) do ;
         %_ReadSStatement(
            debugging=&debugging
         )
      End ;
      When ( "A" ) do ;
         %_ReadAStatement(
            debugging=&debugging
         )
      End ;
      When ( "/" ) do ;
         %_VirguleErrMessage(
            debugging=&debugging
         )
      End ;
      When ( " " ) do ; /* Empty Row */
         %_ReadNextRecordCode(
            debugging=&debugging
         )
      End ;
      Otherwise  do ;
         %_GenErrMessage(
            debugging=&debugging
         )
      End ;
   End ;

%Mend _ReadOneStatement ;
```

This is long, but it does fit my half-page criterion when I have longer lines of code. (That is, each WHEN statement, including the DO block, will fit on one line of code.) This macro called a lot of other macros that we will have to define.

Three notes:

1. The colon (**:**) format modifier tells SAS to start reading at the next non-blank column. So the statement-type identifier does not have to be in the first column.
2. Our source code can have blank records. We do not want to output these, so our macro will need to explicitly tell SAS when to output an observation. If we come to an empty record, we will simply move to the next record.
3. Every time we read a statement, we need to read all the way through every line continuation of the statement. So the **%_ReadOneStatement** macro should never see a line continuation.

LEVEL 3: SUPPORTING THE SUPPORTING MACROS

Working on a SAS job
Going down, down, down.
Working on a SAS job;
Whoop, I wanna lie down.

Five o'clock in the morning,
My code is still not done.
Lord I'm so tired.
How long can this go on?

I'm a working on a SAS job …

Okay, the **%_ReadOneStatement** macro was easy because it only decided which other macros to call. Let's define these macros.

READ P STATEMENT

We don't need the **P** statement for this task. However, we

do want to address it and to write our code so that it is clear where and how to change things if we want to expand the scope of our macro.

```
%Macro _ReadPStatement ( debugging=0 ) ;

   %_GoToNextStatement(
      debugging=&debugging
   )

%Mend _ReadPStatement ;
```

So, this macro simply calls another macro. Does this seem a little trite? On the surface, yes. We could have called the **%_GoToNextStatement** macro directly. Remember, however, that a macro that works is best left alone. If we called **%_GoToNextStatement** directly from **%_ReadOneStatement**, we would have to change both the **%_ReadOneStatement** and **%_ReadPStatement** macros if we want to do something with the **P** statement. This way, we only have to change the **%_ReadPStatement** macro.

We need similar macros for each of the statements that we do not need for this task. We don't need the **H**, **Q**, **R**, **S**, and **A** statements.

READ V STATEMENT

This macro will read a variable statement. Variable statements start with a vee (**V**). Variable statements are in the form

V *name width type* [*possibly more stuff we don't need*]

The following statement
V Q1 02 N
defines variable **Q1** as two columns wide, containing numeric data. Since the source file is written in a free-format language, you can place the vee in a column other than column 1 and insert as many spaces between the elements as wanted. However, the elements must occur in the order as specified.

The variable name may be any valid SAS variable, except that we artificially restricted the length to six characters or fewer for this macro. COBOL keywords cannot be used because the standard job to read this file was written in COBOL.

```
%Macro _ReadVStatement ( debugging=0 ) ;

   Input VarName $ _Length_ $ Type $ @@ ;

   %_ReadNextRecordCode(
      debugging=&debugging
   )

   FmtName = trim( VarName ) || "F" ;

   Select ( Type ) ;
      When ( "G" ) delete ;
      When ( "N" ) ;
      When ( "A" ) do ;
         Type = "C" ;
         FmtName = "$" || FmtName ;
      End ;
   End ;

%Mend _ReadVStatement ;
```

Not too bad. Statement type **G** denotes a group of variables, and is not a part of the SAS data set. So we will drop that record.

As you can see, we had to do some coding here. But the macro is still quite easy to visualize. We will need to define the **%_ReadNextRecordCode** macro before we are done, but we need that macro for our **%_ReadOneStatement** macro anyway.

READ C STATEMENT

The code statement equates coded (actual or range of) values to a description. If a survey item (such as a verbatim answer or a comment) does not have codes, no **C** statement is required. You have the option to link the **C** statement to a skip pattern statement by one or more asterisks. The code statement consists of five elements:

C [******] *value* = *text*

You use the **+** (blank) code when the question is skipped.

```
%Macro _ReadCStatement ( debugging=0 ) ;

   %_ReadValueRange( debugging=&debugging )

   %_ReadValueLabel( debugging=&debugging )

%Mend _ReadCStatement ;
```

Okay, we have read the **V** statement to get the variable name and its type. From that, we have built the format name. Now we have read the starting and ending values and the label. We will need to define the **%_ReadValueRange** and **%_ReadValueLabel** macros.

READ NEXT RECORD CODE

The **%_ReadNextRecordCode** macro was called by the **%_ReadOneStatement** and **%_ReadVStatement** macros. This macro simply looks ahead to see what the statement identifier is for the next line of code.

If we are *on* the last record we cannot *advance to* the next record. Duh! So simply release the record with the INPUT statement (no trailing @@). Otherwise advance to the next record and read the first character.

```
%Macro _ReadNextRecordCode (debugging=0) ;

   If EOF
      then input ;
      Else input
         / @1 _NextRecordCode_ : $1. @@
      ;

%Mend _ReadNextRecordCode ;
```

Hey, this macro calls no others! So this is the end of this branch and the code was really quite simple.

VIRGULE ERR MESSAGE

The **%_VirguleErrMessage** macro is called by the **%_ReadOneStatement** macro, but it really should never be called at all. That is to say, all line continuations should be processed by one of the **%_Read**?**Statement** macros where ? is in {**P**,**H**,**V**,**Q**,**C**,**R**,**S**,**A**}. However, if the source file is badly written we might try to read a line continuation from the **%_ReadOneStatement** macro and we will need to know that something is wong.

```
%Macro _VirguleErrMessage ( debugging=0 ) ;

   Put
      "ERROR: / records should never be"
      " accessed from the"
      " _ReadOneStatement macro!"
    / _inFile_
   ;

   %_GoToNextStatement(
      debugging=&debugging
   )

%Mend _VirguleErrMessage ;
```

GEN ERR MESSAGE

Suppose a statement starts with a symbol other than those in { **P**,**H**,**V**,**Q**,**C**,**R**,**S**,**A**,**/**}. Then we have another error in the source file that we need to know about. So let's write this error and move on.

```
%Macro _GenErrMessage ( debugging=0 ) ;

   Put
      "WARNING: " _StatementType_=
      "is not coded."
    / _inFile_
   ;

   %_GoToNextStatement(
      debugging=&debugging
   )

 %Mend _GenErrMessage ;
```

We still have macros that have been called but not defined. We need to go deeper with our programming.

DEEPER AND DEEPER

GO TO NEXT STATEMENT

The **%_GoToNextStatement** macro was called by the **%_ReadPStatement**, **%_ReadHStatement**, … macros. So let's define it here.

Statements can span records, so we need to keep skipping records until the next record code is not a line continuation. Since we have already defined the **%_ReadNextRecordCode** macro, we will use it here.

```
%Macro _GoToNextStatement ( debugging=0 ) ;

   %_ReadNextRecordCode(
      debugging=&debugging
   )

   Do while ( _NextRecordCode_ eq "/" ) ;
      %_ReadNextRecordCode(
         debugging=&debugging
      )
   End ;

%Mend _GoToNextStatement ;
```

Again, this is very simple code. We have already defined the **%_ReadNextRecordCode** macro.

READ VALUE RANGE

The **%_ReadValueRange** macro was called by the **%_ReadCStatement** macro. Let's investigate how to build this macro.

The value that we want to read might be a single value or it might be a range of values of the form *lowest value – highest value*. It also might have a series of asterisks before the values that indicate a condition that prompts us to skip certain questions. About the only thing that all of the **C** statements have in common is an equals sign that separates the value label from the rest of the statement.

Let's define the a temporary variable, called **_Value_**, with plenty of space so that we can be sure to include all the source-file code for the value. How long does it need to be? Well, the limit for a record in the source file is 100 characters. The first character must be a statement identifier or a line continuation indicator. That leaves 99 characters. So let's make **_Value_** 99 characters long. Why not; it's a temporary variable. The value, or range of values, is the part before the equal sign, with any asterisks (skip indicators) removed. Since the source code is free format, and spaces have little meaning, let's compress out the spaces so that we have consistent code to process. Plus signs (**+**) indicate missing values; multiple plus signs simply reflect the length of the field. So we will need to convert the source file's missing value indicator to the appropriate SAS missing value indicator. For now, we will simply insert a macro call to indicate this task needs to be coded. We will probably want to pass the name of the variable that we want to set to missing, and the name of the variable that specifies the data type, in case this macro sees further use.

If we are working with a range of values rather than a single value, we will need to parse that range into the starting value and the ending value. If we are working with only one value we will simply copy the value of **Start** to **End**.

```
%Macro _ReadValueRange ( debugging=0 ) ;

   Input _Value_ $char99. @@ ;

   _Value_ = scan( _Value_ , 1 , "=" ) ;

   _Value_ = compress( _Value_ , " *" ) ;

   If index(_Value_,"+")
      then do ;
         %_SetPlusToMissing(
            var=_Value_
          , type=Type
          , debugging=&debugging
         )
      End ;
      Else do ;
         Start = scan( _Value_ , 1 , "-" ) ;
         End   = scan( _Value_ , 2 , "-" ) ;
         If ( End = " " ) then End = Start ;
      End ;

%Mend _ReadValueRange ;
```

The SCAN function looks for words, and allows us to specify word partitions. So **SCAN(_Value_ , 1 , "=")** tells SAS to partition the character string called **_Value_** into smaller character strings using the equal sign (**=**) as the partition delimiter. The "**1**" tells SAS to return the first partition. So this gives us all the characters between the statement identifier (**C**) which is not part of **_Value_** and the equal sign.

The COMPRESS function removes characters specified in

the second argument from the first argument. So **COMPRESS(_Value_ , " *")** will remove the skip-pattern indicators; we are not dealing with them in this code. White space is not significant; the creators of the source file can include spaces or not, as they prefer. But we need something constant to deal with. So let's use the COMPRESS function to remove blanks. Of course, we can remove both the blanks and the asterisks with one compress function.

We will define the **%_SetPlusToMissing** macro later.

PROC FORMAT does not require that a CNTLIN= data set have an **End** variable, but if it does it must not have missing values. If we are dealing with a range of values, we need to partition our **_Value_** variable into the part before the hyphen (called **Start**) and the part after (called **End**). If there is no hyphen, we must populate the **End** variable with the contents of the **Start** variable.

READ VALUE LABEL

The **%_ReadValueLabel** macro was called by the **%_ReadCStatement** macro. The value label starts after the equal sign (**=**) and continues to the start of the next statement. Remember, it can span multiple input lines. So we need read, check if the new line is a new statement, and concatenate the new line to the label as long as the new line is still a continuation of the statement.

Starting after the equals sign (**=**), read the first of the value label. When you ask SAS to start input immediately after it finds a character string, it only searches forward for that character string. If our input pointer is already past the equals sign, then we will have to move the pointer back to the start of the line. So, to be safe, let's always back up. After reading the part of the label on the initial record of the **C** statement, we will append any line continuations to the label.

```
%Macro _ReadValueLabel ( debugging=0 ) ;

   Input @1 @"=" Label ?? $char99. @@ ;

   %_ReadNextRecordCode(
      debugging=&debugging
   )
   Label = left( Label ) ;

   Do while ( _NextRecordCode_ eq "/" ) ;

      Input _MoreLabel_ ?? $char99. @@ ;

      Label =
         trim( Label ) || " " ||
         left( _MoreLabel_ )
      ;

      %_ReadNextRecordCode(
         debugging=&debugging
      )

   End ;

%Mend _ReadValueLabel ;
```

We called the same **%_ReadNextRecordCode** macro that we called in the **%_ReadOneStatement**, **%_GoToNextStatement**, and **%_ReadVStatement** macros.

GET ON DOWN

SET PLUS TO MISSING

It seems the only macro we have yet to define is the **%_SetPlusToMissing** macro which was called by the **%_ReadValueRange** macro. The **%_SetPlusToMissing** will set the variable passed in with the **var=** parameter to the appropriate missing value based on the data type as defined by the **type=** parameter.

```
%Macro _SetPlusToMissing (
   var=Start
   type=Type
 , debugging=0
) ;

   Select ( &type ) ;
      When ( "C" ) &var = " " ;
      When ( "N" ) &var = "." ;
      Otherwise put
         "ERROR: For variable " VarName
         "-- " &var= "and " &type=
       / _inFile_
      ;
   End ;

%Mend _SetPlusToMissing ;
```

Another simple macro.

Hey, we're done!

CONCLUSION

As you see, jobs are much simpler to understand, program, and modify if they are approached from a top-down perspective. Look at the general requirements for the job and, if the steps are too complicated or if you simply don't understand them yet, code a macro call and worry about the coding of the macro later. Keep working in this fashion until the tasks get simple enough to easily code.

Remember, it is much easier to read code if you can see the beginning and the end of a block of code on one page – even better a half page.

REFERENCES

"Working on a SAS Job" is a parody of Lee Dorsey's "Working in a Coal Mine."

ACKNOWLEDGMENTS

I want to thank Ian Whitlock of Westat for his continual support and encouragement in my career growth. He was an inspiration before I met him, and has proven to be a wonderful mentor and friend since.

I also want to thank Dianne Rhodes of Westat, who directed my focus toward more career-enhancing facilities such as SAS Users Groups and the SAS-L list server when we both worked elsewhere.

Finally, I want to thank all the wonderful and insightful contributors to SAS-L for their selfless contributions. They have proven to be my most valuable aid as I learn how to be a SAS programmer.

USA and other countries. ® indicates USA registration.

Other brand and product names are registered trademarks or trademarks of their respective companies.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Edward Heaton
Westat
1650 Research Boulevard
Rockville, MD 20850
Work Phone: (301) 610-4818
Fax: (301) 294-3992
Email: EdwardHeaton@Westat.com

Paper #P814

The Power of PROC DATASETS

Lisa M. Davis, Blue Cross Blue Shield of Florida, Jacksonville, Florida

ABSTRACT

The DATASETS procedure can be used to do many functions that are normally done within a SAS data step more efficiently.
For example:

- Labeling and renaming variables
- Concatenating and indexing datasets

This paper will demonstrate the power of the PROC DATASETS procedure and the enhancements made in V8. This paper is intended for beginning to intermediate SAS users. PROC DATASETS is a powerful procedure that everyone needs to know.

INTRODUCTION

PROC DATASETS is a SAS utility used to manage more than one SAS file at a time. This procedure allows you to append, copy, delete, label, rename, index, and collect information about the dataset that has been modified, all in one step. The DATASETS procedure executes in order. The first statement executes first, then the second, and so on. This allows you to concatenate two data sets, then rename the variables, change the labels and create an index all in the same procedure. The ability to do so improves processing time, programming length, and data steps needed. The following is a list of statements used in the DATASETS procedure:

```
PROC DATASETS;
    AGE;
    AUDIT;
    CHANGE;
    CONTENTS;
    COPY;
       EXCLUDE;
       SELECT;
    DELETE;
    EXCHANGE;
    MODIFY;
        FORMAT;
        IC CREATE;
        IC DELETE;
        IC REACTIVB;
        INDEX CREATE;
        INDEX DELETE;
        INFORMAT;
        LABEL;
        RENAME;
    REPAIR;
    SAVE;
RUN;
QUIT;
```

This tutorial will show you why and how to use PROC DATASETS. We will not cover all statements and options with the DATASETS procedure, but you will walk away knowing how powerful this procedure is.

THE DATASETS STATEMENT

The DATASETS procedure is an interactive procedure that executes immediately and does not stop processing until QUIT or RUN CANCEL command is issued. The DATASETS statement executes a list of all of the members in a SAS library in the log of your program. The list can contain members with a member type of: data, view, access, catalog, any and program.

The general form of the DATASETS statement is:

```
PROC DATASETS LIBRARY=LIBREF
              MEMTYPE=MEM-LIST <OPTIONS>;
```

The LIBRARY= and MEMTYPE= are options, but I always stress to specify both a library reference and a member type so you know what library and member type you are working with. This is crucial when you have several member types in the same library that could be named the same.

If you haven't worked with a library over a period of time, this is a good way to find out what is in that library. Here is the code to do so:

```
libname mylib 'c:\temp';

proc datasets library=mylib memtype=data;
run; quit;
```

Refer to OUTPUT 1.1 in the appendix

The output generated in you r log, gives you the libref pointing to the library, engine the data set was created with, physical name, file name, name of the datasets that are located in that library, the memtype (in this case memtype = data), file size, and the day and time the data set was last modified. As you can see there are three data sets in the library: MORTGAGES, PLUS, SECUREDLOANS. These are the data sets we will be working with throughout this tutorial.

One other option I want to cover with the DATASETS statement is KILL. KILL deletes all data sets within a library automatically. The general form is:

```
proc datasets library=mylib memtype=data
kill;
```

Caution: KILL executes immediately before the DATASETS procedure completes processing.

THE CONTENTS STATEMENT

The CONTENTS statement acts the same as the CONTENTS procedure. This statement gives you information about the variables within a SAS library. How do you know which one to use? The CONTENTS statement is very useful when you are combining other DATASETS statements to manipulate a SAS library. Otherwise PROC CONTENTS is recommended to use.
The general form of the CONTENTS statement is:

```
CONTENTS DATA=LIBREF.MEMBER <OPTIONS>
```

DATA = is an option that is very useful to always use. This specifies which library and member you want contents on. The libref is not always needed in the case that the libref is specified in the DATASETS statement. _ALL_ is also an option that may be used when you want contents on all of the data sets that reside in that library.

To find out what variables reside in the data set SECUREDLOANS submit the following code:

```
libname mylib 'c:\temp';
```

```
proc datasets library=mylib memtype=data;
contents data=securedloans;
run; quit;
```

Refer to OUTPUT 1.2 in the appendix.
The CONTENTS statement is added after the DATASETS statement after all the changes have been done to that dataset. The CONTENTS statement gives you number of observations, engine created with, last date modified, and engine host information. It also gives you a list of variables within the data set, type of variable, length, format, position, informat, and labels. Examples of how the CONTENTS statement is used with other statements will be covered later in the tutorial.

THE APPEND STATEMENT

The APPEND statement is used to concatenate two SAS data sets together. SAS takes one data set and appends the second data set to the bottom of the first. Being able to do this in one step saves processing time and space allocation. Only SAS data sets can be concatenated together.
The general form of an APPEND statement is:

```
APPEND BASE=SAS DATASET DATA=SAS DATASET <FORCE>
```

The BASE= is the SAS dataset that you want the observations added to. The DATA= is the SAS dataset that you want added. FORCE is an option that is used when you want to force a concatenation when the two data sets have different variable names. The following code is an example of concatenating two data sets and viewing the contents after the two are appended.

```
libname mylib 'c:\temp';

proc datasets library=mylib memtype=data;
append base=securedloans data=mortgages;
contents data=securedloans;
run; quit;
```

Refer to OUTPUT 1.3 in the appendix
This example appends the MORTGAGES data set to the bottom of the SEURED LOANS data set. The contents is ran on the SECUREDLOANS data set showing that the number of observations in MORTGAGES have been added to SECUREDLOANS. The MORTGAGES data set still exists in the library. This was done in one procedure versus a data step, creating a third data set, and the CONTENTS procedure. This may not be noticeably faster with small data sets, but as the data sets exist of million of rows, the processing time surpasses by hours and space allocation is greatly reduced. Of course this means major savings, these days.

THE DELETE STATEMENT

The DELETE statement deletes specified data sets within a library. Multiple data sets or all of the data sets can be deleted at the same time in a library. The deletion occurs immediately, and does not wait for the DATASETS procedure to complete. For example: If you delete a member in the first line of the DATASETS procedure you cannot run a CONTENTS statement referring to that member. You will receive a 'this file does not exist' error. The advantage of using the DELETE statement is during a long process you can delete data sets that are no longer being used. This frees up space and reallocates this space to be used in your same process.

The general form of the DELETE statement is:

```
DELETE MEMBER-LIST
```

For an example:

```
libname mylib 'c:\temp';

proc datasets library=mylib memtype=data;
append base=securedloans data=mortgages;
delete mortgages;
contents data=securedloans;
run; quit;
```

The DELETE statement here deletes the data set MORTGAGES because this data set has been appended to the SECUREDLOANS data set and is no longer needed. The space MORTGAGES was occupying is now free to be used to store another data set.

THE MODIFY STATEMENT

The MODIFY statement; in my opinion is the most powerful and useful statement in the DATASETS procedure. Within the MODIFY statement you can label, rename, create and delete indexes, create integrity constraints, delete integrity constraints, reactivate integrity constraints, format, and informat variables within a library. These actions can only occur after a MODIFY statement. We will discuss several of these actions that are most used in this procedure. The structure of the MODIFY statement is:

```
MODIFY DATA SET <OPTIONS>;
    FORMAT;
    IC CREATE;
    IC DELETE;
    IC REACTIVB;
    INDEX CREATE;
    INDEX DELETE;
    INFORMAT;
    LABEL;
    RENAME;
```

The MODIFY statement alone points to the data set that you want to change. The LABEL option allows creating or deleting a label on the data set specified. For example:

```
libname mylib 'c:\temp';

proc datasets library=mylib memtype=data;
modify securedloans(label='Secured Loans');
contents data=securedloans;
run; quit;
```

Refer to OUTPUT 1.4 in the appendix
As you can see in the contents output you can see that the SECREDLOANS data set has been labeled 'SECURED LOANS'. If you wanted to delete this label you would use the label option and leave a blank ' '. Also with multiple MODIFY statements you can modify more than one dataset at a time. The reason you may want to do this is to prepare two data sets to be merged without all of the preparation data steps.

```
libname mylib 'c:\temp';

proc datasets library=mylib memtype=data;
modify securedloans(label='Secured Loans');
modify mortgages (label='Mortgage Loans');
contents data=securedloans;
contents data=mortgages; run;quit;
```

THE INDEX STATEMENT
The INDEX statement allows you to create or delete an index on a SAS data set. Creating an index on a SAS data set allows for more efficient processing of observations. If you wanted to do BY processing on two data sets with an index created, sorting is not needed. By eliminating sorting, again processing time and space is saved. The advantage of

creating an index instead of sorting is within the DATASETS procedure you can combine several statements to manipulate the data set in one procedure instead of multiple. If you wanted to merge two data sets you could do so without sorting. Once an index is created, you can rename, copy, label, etc… and the index will be transferred.
The general form of an INDEX statement is:

```
INDEX CREATE  VARIABLE-LIST  or
INDEX DELETE INDEX-LIST
```

If you want to merge two very large data sets by two variables, you can first create an index on these two variables on both data sets at the same time. For example:

```
libname mylib 'c:\temp';

proc datasets library=mylib memtype=data;
     modify securedloans(label='Secured Loans');
            index create accno cct_no;
     modify plus (label='Plus Customers');
            index create account companycost;
     contents data=securedloans;
run;quit;
```

Refer to OUTPUT 1.5 in the appendix
By creating an index on the SECUREDLOANS data set and PLUS data set, you have avoided two SORT procedures, saving time and space again. As you look in the indexes portion of the contents output, you can see that the indexes create two extensions of the data sets. These extensions are treated as the data set; so all indexes transfer through all modifications.

THE LABEL STATEMENT
The LABEL statement allows you to label variable within a data set. Multiple variables can be labeled within one MODIFY statement. Multiple variables from different data set can also be labeled within several MODIFY statements.
The general form of a LABEL statement is:

```
LABEL VARIABLE='LABEL';
```

For example:

```
libname mylib 'c:\temp';

proc datasets library=mylib memtype=data;
     modify securedloans(label='Secured Loans');
            label accno='Account Number'
                  cct_no='Cost Center';
     modify plus (label='Plus Customers');
            label acct='Account Number'
                  cost='Cost Center';
     contents data=securedloans;

run;quit;
```

Refer to OUPUT 1.6 in the appendix
As you can see in the variable list portion of the contents output you can see that ACCNO and CCT_NO have been labeled within the SECUREDLOANS data set. Along with ACCT and COST within the PLUS data set. The advantage of using the LABEL statement with the DATASETS procedure is that the labels are stored permanently in the data set. If you execute a LABEL statement within other procedures such as: PROC FREQ, PROC PRINT, etc, the label is only active for that procedure. With the labels being stored permanently, you do not have to worry about label consistency throughout the reports you produce.

THE RENAME STATEMENT
The RENAME statement allows you to rename variables within a data set. Multiple variables can be renamed at one time.
The general form of a RENAME statement is:

```
RENAME VARIALBLE=NEW VARIABLE
```

Once you rename a variable, the new name overwrites the old name. As you can tell we are building step-by-step of the DATASETS procedure to allow you to get the most benefit and power of this procedure. So if you wanted to merge two data sets by two variables that were named different in both data sets, you would rename the variables so they matched for merging. First you would want to create an index, so you could avoid sorting, second rename the variables so they match each other, avoiding one possibly two data steps. For example:

```
libname mylib 'c:\temp';

proc datasets library=mylib memtype=data;
modify securedloans(label='Secured Loans');
     index create accno cct_no;
     rename accno=accountnumber
            cct_no=costcenter;
     label accountnumber='Account Number'
           costcenter='Cost Center';

modify plus (label='Plus Customers');
     index create account companycost;
     rename account=accountnumber
            companycost=costcenter;
     label accountnumber='Account Number'
           costcenter='Cost Center';
contents data=securedloans;
run;quit;
```

Refer to OUPUT 1.7 in the appendix
In this example we are preparing our data sets to be able to merge. We have labeled our data sets in the MODIFY statements. We created indexes on the two data sets. Now we renamed ACCNO to ACCOUNTNUMBER and CCT_NO to COSTCENTER in the SECUREDLOANS data set. Then we did the same in the PLUS data set. We renamed ACCOUNT to ACCOUNTNUMBER and COMPANYCOST to COSTCENTER. After renaming the variables, the index is transferred to the new names of the variables. This is why it was crucial to create the index before renaming or modifying the data set any further. In the contents output of SECUREDLOANS you can see that the variables have been renamed and the indexes have been transferred to the new names.

THE FORMAT STATEMENT
The FORMAT statement is used to modify, change, or add a format onto a variable. You can also use the INFORMAT statement to change how the variable is read in. The FORMAT statement changes how the variable is put out.

The general form of the FORMAT and INFORMAT statements are:

```
FORMAT VARIABLE-LIST format    or
INFORMAT VARIABLE-LIST format
```

The following is an example of how the FORMAT statement is used:

```
libname mylib 'c:\temp';

proc datasets library=mylib memtype=data;
     modify securedloans(label='Secured
Loans');
```

```
            index create accno cct_no;
            rename accno=accountnumber
                   cct_no=costcenter;
            label accountnumber='Account Number'
                     costcenter='Cost Center';
            format accountnumber $12.
                   costcenter 8.;
     modify plus (label='Plus Customers');
            index create account companycost;
            rename account=accountnumber
               companycost=costcenter;
            label accountnumber='Account Number'
                   costcenter='Cost Center';
            format accountnumber $12.
                   costcenter 8.;
     contents data=securedloans;

run;quit;
```

In this example we have made ACCOUNTNUMBER to be outputted as a character with a length of 12, COSTCENTER a numeric with a length of 8. Notice that both the LABEL and FORMAT statements were done on the new variable names. This is allowed because the DATASETS procedure executes in order and automatically.

TYING EVERYTHING TOGETHER

Now that we have learned the basics of the DATASETS procedure, I want to give a complete example of everything we have learned and compare it to what you would have to do if you did not use the DATASETS procedure.
Example:

```
libname mylib 'c:\temp';

proc datasets library=mylib memtype=data;
     append base=securedloans data=mortgages;
     delete mortgages;
     modify securedloans(label='Secured Loans');
            index create accno cct_no;
            rename accno=accountnumber
                   cct_no=costcenter;
            label accountnumber='Account Number'
                   costcenter='Cost Center';
            format accountnumber $12.
                   costcenter 8.;
     modify plus (label='Plus Customers');
            index create account companycost;
            rename account=accountnumber
               companycost=costcenter;
            label accountnumber='Account Number'
                   costcenter='Cost Center';
            format accountnumber $12.
                   costcenter 8.;
     contents data=securedloans;

run;quit;
```

Refer to OUTPUT 1.8 in the appendix
This example gives us a complete look at the statements we have covered. This example is good for the following scenario:
You have three SAS data sets. Two of the data sets have the same data but about different products. You want to combine the two product data sets and merge it with the third data set to get demographic information on those customers with these precuts. So the steps would be:

1. Concatenate the two product data sets together
2. Delete the data set that was concatenated so you can save of space
3. Create an index so when you merge the two data sets you do not have to sort them
4. Rename the variables on the two data sets so they will be able to merge
5. Label the data set and variables to have consistency on reports
6. Format how you want the variables outputted on your reports
7. Get information about the two data sets to make sure everything is correct

All seven steps can be done in one procedure. Here is an example of what would have to been done if the DATASETS procedure was not used:

```
libname mylib 'c:\temp';

data mylib.three;
     format accountnumber $12.
                costcenter 8.;
  set mylib.one(rename=(accno=accountnumber
                     cct_no=costcenter))
    mylib.two(rename=(accno=accountnumber
                     cct_no=costcenter));
     label accountnumber='Account Number'
              costcenter='Cost Center';
     run;

proc datasets library=mylib memtype=data;
     delete one tow;
     run;quit;

proc contents data=mylib.three;
     run;

data mylib.plus2;
     format accountnumber $12.
                costcenter 8.;
  set mylib.plus(rename=(acct=accountnumber
                  companycost=costcenter));
     label accountnumber='Account Number'
              costcenter='Cost Center';
     run;

proc datasets library=mylib memtype=data;
     delete plus;
     run;quit;

proc contents data=mylib.plus2;
     run;

proc sort data=mylib.three;
     by accountnumber costcenter;
     run;

proc sort data=mylib.plus2;
     by accountnumber costcenter; run;
```

In this example you can see that the program is much longer (code wise) , multiple data steps and procedures were used. Processing time was always faster using the DATASETS procedure, but the time was greatly reduced when using large amounts of data. By using the second example, it requires you to know more syntax, procedures and data steps within SAS. If you know the DATASETS procedure you can do all of this with only knowing one procedure.

CONCLUSION

The DATASETS procedure is a powerful procedure to know. I only touched on the basis of this procedure; how to know what data sets exist in your library, information on the variable

within you data set, append two data sets together, indexing data sets, renaming variables, labeling data sets and variables, modifying data sets, deleting data sets, all within one procedure. There are a lot more things this procedure can do and I challenge you learn all you can about this procedure. Saving time, space and work is what our goal is as programmers. PROC DATASETS does all three for us with little effort.

REFERENCES

SAS Institute Inc. (1990), *SAS Procedures Guide, Version 6, Third Edition*, Cary, NC: SAS Institute Inc.

SAS Institute Inc. (1990), *SAS Language Reference, Version 6, Third Edition*, Cary, NC: SAS Institute Inc.

SAS Institute Inc. (1999)*, "SAS Procedures", SAS Version 8 Online Documentation*, Cary, NC: SAS Institute Inc.

ACKNOWLEDGMENTS

Special thanks to Kevin Finnerty for allowing the time to create this tutorial.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Lisa M. Davis
Blue Cross Blue Shield of Florida
4800 Deerwood Campus Parkway
DCC3-2
Jacksonville, Florida 32243
Work Phone: 904-905-3019
Fax: 904-905-1009
Email: lisa.davis@bcbsfl.com

APPENDIX

OUTPUT 1.1

```
                Libref:        MYLIB
                Engine:        V8
                Physical Name: c:\temp
                File Name:     c:\temp


                  File
   #  Name          Memtype     Size  Last Modified
_____________________________________________________________
1  MORTGAGES     DATA      648192   19JUL2000:21:53:54
2  PLUS          DATA      123904   19JUL2000:21:52:56
3  SECUREDLOANS  DATA      656384   19JUL2000:16:02:02
```

OUTPUT 1.2

```
                          The DATASETS Procedure

Data Set Name: MYLIB.SECUREDLOANS      Observations: 10000
Member Type: DATA                      Variables:    8
Engine:        V8                      Indexes:      0
Created:15:10 Tuesday, July 11, 2000   Observation Length: 64
Last Modified: 16:02 Wednesday, July 19, 2000
Deleted Observations: 0
Protection:                            Compressed:   NO
Data Set Type:                         Sorted:       YES
Label:

                 -----Engine/Host Dependent Information-----

                  Data Set Page Size:         8192
                  Number of Data Set Pages:   80
                  First Data Page:            1
                  Max Obs per Page:           127
                  Obs in First Data Page:     96
                  Number of Data Set Repairs: 0
                  File Name: c:\temp\securedloans.sas7bdat
                  Release Created:            8.0000M0
                  Host Created:               WIN_NT

#    Variable          Type    Len    Pos    Format Informat  Label
_____________________________________________________________________________
 1  ACCNO             Char     21     32    $21.    $21.     ACCNO
 3  ACC_OPN_DT        Num       8      8    DATE9.  DATE9.   ACC_OPN_DT
 6  ACC_PD_CTGY_CD    Char      3     53    $3.     $3.      ACC_PD_CTGY_CD
 2  CCT_NO            Num       8      0    11.     11.      CCT_NO
 4  CLS_DT            Num       8     16    DATE9.  DATE9.   CLS_DT
 7  PRD_PRMRY_TYPE_CD Char      3     56    $3.     $3.      PRD_PRMRY_TYPE_CD
 8  PRD_SECDRY_TYP_CD Char      3     59    $3.     $3.      PRD_SECDRY_TYP_CD
 5  acctbalance       Num       8     24    17.2    17.2     Average Account

                          -----Sort Information-----

                        Sortedby:      CCT_NO
                         Validated:    YES
                        Character Set: ANSI
```

OUTPUT 1.3

```
                    The DATASETS Procedure

Data Set Name: MYLIB.SECUREDLOANS    Observations:      20000
Member Type:   DATA                  Variables:             8
Engine:        V8                    Indexes:               0
Created:15:10 Tuesday, July 11, 2000 Observation Length:   64
Last Modified: 22:03 Wednesday, July 19, 2000
Deleted Observations: 0
Protection:                          Compressed:           NO
Data Set Type:                       Sorted:               NO
Label:
```

OUTPUT 1.4

```
                    The DATASETS Procedure

Data Set Name: MYLIB.SECUREDLOANS      Observations:     20000
Member Type:   DATA                    Variables:            8
Engine:        V8                      Indexes:              0
Created:15:10 Tuesday, July 11, 2000   Observation Length:  64
Last Modified: 22:19 Wednesday, July 19, 2000
Deleted Observations: 0
Protection:                            Compressed:          NO
Data Set Type:                         Sorted:              NO
Label:         Secured Loans
```

OUTPUT 1.5

```
                  -----Directory-----

                  Libref:        MYLIB
                  Engine:        V8
                  Physical Name: c:\temp
                  File Name:     c:\temp


                                  File
 #  Name           Memtype       Size     Last Modified
_____________________________________________________________
 1  MORTGAGES      DATA        656384   19JUL2000:22:26:26
 2  PLUS           DATA         74752   20JUL2000:00:28:04
    PLUS           INDEX        21504   20JUL2000:00:28:04
 3  PLUSCUSTOMERS  DATA        123904   19JUL2000:22:05:48
 4  SECUREDLOANS   DATA       1950720   20JUL2000:00:28:02
    SECUREDLOANS   INDEX       865280   20JUL2000:00:28:02
```

OUTPUT 1.6

-----Alphabetic List of Variables and Attributes-----

#	Variable	Type	Len	Pos	Format	Informat	Label
1	ACCNO	Char	21	32	$21.	$21.	Account Number
3	ACC_OPN_DT	Num	8	8	DATE9.	DATE9.	ACC_OPN_DT
6	ACC_PD_CTGY_CD	Char	3	53	$3.	$3.	ACC_PD_CTGY_CD
2	CCT_NO	Num	8	0	11.	11.	Cost Center
4	CLS_DT	Num	8	16	DATE9.	DATE9.	CLS_DT
7	PRD_PRMRY_TYPE_CD	Char	3	56	$3.	$3.	PRD_PRMRY_TYPE_CD
8	PRD_SECDRY_TYP_CD	Char	3	59	$3.	$3.	PRD_SECDRY_TYP_CD
5	acctbalance	Num	8	24	17.2	17.2	Average Account

OUTPUT 1.7

-----Alphabetic List of Variables and Attributes-----

#	Variable	Type	Len	Pos	Format	Informat	Label
3	ACC_OPN_DT	Num	8	8	DATE9.	DATE9.	ACC_OPN_DT
6	ACC_PD_CTGY_CD	Char	3	53	$3.	$3.	ACC_PD_CTGY_CD
4	CLS_DT	Num	8	16	DATE9.	DATE9.	CLS_DT
7	PRD_PRMRY_TYPE_CD	Char	3	56	$3.	$3.	PRD_PRMRY_TYPE_CD
8	PRD_SECDRY_TYP_CD	Char	3	59	$3.	$3.	PRD_SECDRY_TYP_CD
1	accountnumber	Char	21	32	$12.	$21.	Account Number
2	costcenter	Num	8	0	F8.	11.	Cost Center
5	acctbalance	Num	8	24	17.2	17.2	Average Account

-----Alphabetic List of Indexes and Attributes-----

#	Index	# of Unique Values
1	accountnumber	10000
2	costcenter	116

OUTPUT 1.8

-----Directory-----

Libref: MYLIB
Engine: V8
Physical Name: c:\temp
File Name: c:\temp

#	Name	Memtype	File Size	Last Modified
1	PLUS	DATA	74752	20JUL2000:00:33:34
	PLUS	INDEX	21504	20JUL2000:00:33:34
2	PLUSCUSTOMERS	DATA	123904	19JUL2000:22:05:48
3	SECUREDLOANS	DATA	1950720	20JUL2000:00:33:34
	SECUREDLOANS	INDEX	865280	20JUL2000:00:33:34

Data Set Name: MYLIB.SECUREDLOANS Observations: 20000
Member Type: DATA Variables: 8
Engine: V8 Indexes: 2
Created:15:10 Tuesday, July 11, 2000 Observation Length:64
Last Modified: 0:28 Thursday, July 20, 2000
Deleted Observations: 0
Protection: Compressed: NO
Data Set Type: Sorted: NO
Label: Secured Loans

-----Engine/Host Dependent Information-----

Data Set Page Size: 8192
Number of Data Set Pages: 238
First Data Page: 1
Max Obs per Page: 127
Obs in First Data Page: 96
Index File Page Size: 4096
Number of Index File Pages: 211
Number of Data Set Repairs: 0
File Name: c:\temp\securedloans.sas7bdat
Release Created: 8.0000M0
Host Created: WIN_NT

OUTPUT 1.8 CONTINUED

-----Alphabetic List of Variables and Attributes-----

#	Variable	Type	Len	Pos	Format	Informat	Label
3	ACC_OPN_DT	Num	8	8	DATE9.	DATE9.	ACC_OPN_DT
6	ACC_PD_CTGY_CD	Char	3	53	$3.	$3.	ACC_PD_CTGY_CD
4	CLS_DT	Num	8	16	DATE9.	DATE9.	CLS_DT
7	PRD_PRMRY_TYPE_CD	Char	3	56	$3.	$3.	PRD_PRMRY_TYPE_CD
8	PRD_SECDRY_TYP_CD	Char	3	59	$3.	$3.	PRD_SECDRY_TYP_CD
1	accountnumber	Char	21	32	$12.	$21.	Account Number
5	acctbalance	Num	8	24	17.2	17.2	Average Account
2	costcenter	Num	8	0	F8.	11.	Cost Center

-----Alphabetic List of Indexes and Attributes-----

#	Index	# of Unique Values
1	accountnumber	10000
2	costcenter	116

Evaluating the Use of Enterprise Guide in Introductory Statistics Classes

Sandra B. Donaghy and Joy M. Smith, North Carolina State University, Raleigh, NC

ABSTRACT

Enterprise Guide (EG), a Windows thin client, provides a point and click interface to the SAS ® System. EG allows users to access all data types supported by the SAS System and utilize the computing power of any Version 8 SAS Server to generate professional reports and graphics from the PC client interface.

In addition to the tasks that are available in the point and click environment, EG provides a program editor window where the full power of the SAS System is available through traditional SAS programming. The code generated using the point and click environment is viewable and can be useful for learning SAS programming.

This paper includes a handout that we plan to use in a pilot study this summer and our personal evaluation of EG for use in beginning statistics classes. The presentation will include an online demo and discussion of the pilot study.

INTRODUCTION

At NCSU, graduate students use SAS for their research data analysis. Many students report that prospective employers prefer candidates who are skilled SAS programmers. For these reasons, it would be very helpful if students were exposed to SAS in their introductory statistics classes. The complexity of the SAS System has made it difficult to introduce in beginning statistics classes where the focus is on teaching statistical concepts. We are currently evaluating EG to determine if it will make using SAS less complex for both students and faculty.

We intend to conduct a pilot study this summer to evaluate the use of EG in statistics classes. We believe students will be able to quickly perform homework assignments without having to learn SAS syntax. In addition, students who want to learn SAS programming could review the generated code. We are optimistic that EG will satisfy our need for a point and click interface to SAS.

In order for EG to be valuable, it must provide easy data access, analysis, and reporting with minimal need for support. This pilot study should help identify EG features that students find confusing. We are hopeful that the EG interface will provide a gentle introduction to SAS. As the students progress in their studies, they will benefit from learning SAS code and the code window should be useful at that point. The code window will be used in classes where instructors provide sample SAS programs to their students. The results of this pilot study will be shared at the paper presentation.

In addition to teaching, we provide computer and statistical consulting to researchers on campus. EG may be useful to these faculty, staff, and graduate students when they analyze research data. EG will affect this consulting, but we can only speculate on what the effects will be. We are glad that the option to write our own code exists in case a task does not exist or we do not know how to accomplish a task using the EG menus and tasks.

This paper is structured as a student handout so that large parts of it can be used in the pilot study. The handout introduces basic EG concepts and provides one practice exercise. The concepts introduced were carefully chosen to meet the anticipated needs of beginning statistics students. EG has many features that are not covered. We hope to encourage the use of EG by keeping this introduction simple for the instructors and students. We hope that students will benefit from the exposure to SAS and be better prepared to learn more advanced SAS features in the future.

TESTING ENVIRONMENT

Our tests were run on Microsoft Windows NT 4.0 and Microsoft Windows 2000 Professional Edition using EG version 1.2.0.242 and SAS version 8.2 loaded locally. No effort has been made to evaluate processing on remote servers since this arrangement will not be necessary in the classroom.

Currently at NCSU, there are two PC environments in which we anticipate students and researchers will use EG. In the computer labs, licensed software packages, including SAS, are distributed using Novell Client V4.60 for Windows NT and Zenworks Application Launcher. In this environment, student files are stored in their personal network file space. This storage space is accessed using Andrew File System (AFS). The SASUSER library is located in this network space. This setup provides automatic backups and private storage for student files. Processing and workspace are on the local PC. Statistics classes will be taught in these labs, and students who do not have access to a personal computer may use these labs for

their homework. Systems administrators will be responsible for EG Administrator settings.

Alternatively, licensed SAS users may load SAS and EG on personal computers. Many of these SAS users will have networked drives from which data can be accessed, but we expect they will process their data locally. Individuals using this setup will be responsible for the EG Administrator settings. In our experience no modifications are required, so this should not present a problem

We have noticed WORK libraries left behind by abnormally terminated SAS sessions in both of these environments. Any user can delete these libraries from the PC they are logged on to, but we have not found an efficient mechanism for removing these inactive libraries from the lab machines en masse. This is not a problem unless disk space is limited and these folders are allowed to accumulate.

STUDENT HANDOUT

WHY USE SAS AND ENTERPRISE GUIDE?

SAS is a very rich system of procedures that is used extensively at research universities, government agencies, and companies for data analysis and reporting. Many of our graduating students report that they are asked if they know SAS when they interview for jobs. According to the SAS web site, www.sas.com, "Ninety-eight percent of Fortune 100 and ninety percent of Fortune 500 companies are SAS customers." Also, "SAS serves more than 35,000 business, government and university sites in 110 countries." Clearly, experience with SAS is an asset to students who intend to pursue advanced degrees, perform research, or seek high paying jobs in information technology.

SAS Enterprise Guide (EG) is a Windows thin client. It provides a point and click interface to many parts of the SAS System and access to all SAS features using traditional programming methods. Students in this class will use EG to perform their assignments. EG will be used because it allows students to concentrate on the statistical concepts instead of the SAS code. Students interested in learning SAS syntax can view the generated SAS code in the code window. Usually the generated code contains optional statements in addition to the required statements. This makes the code interesting to more advanced SAS users, but probably more overwhelming to new SAS users.

EG provides very thorough online help and a tutorial. Students are encouraged to use them to answer their questions. To learn more select **Help → Enterprise Guide Help** or **Help → Getting Started Tutorial.**

PROJECTS

EG uses a project-based interface. You can have only one project open at a time. For class use, we recommend that you make each homework assignment a separate project and name them something like HW1, HW2, etc. The project window (top left panel) displays the active project and its associated data, code, notes, and results links. These links are referred to in the SAS documentation as nodes.

CREATING PROJECTS

To open a new project select: **File → Open** and select the **New** tab.

To open an existing project select: **File → Open** and select the **Existing** tab. Double click on the project file. The project will also open if you highlight it and select **OK**.

To save a new project select: **File → Save project-name As**
This method allows you to choose a name and storage location. It can also be used to change the name or location of your project. EG projects are binary files that have the extension .seg.

To save an existing project select: **File → Save project-name**
This method will save the project with its original name in its original location.

To close a project select**: File → Close project-name**
If you have created any temporary data sets, you will be asked to save them before you close the project.

Important Note: Be aware that your data is not stored in your project file. Your project will contain links to your data. It is important to know where your data is stored so you can back it up or manage it as needed. To learn more about data storage, see the **SAS Data Libraries** section of this paper.

PROJECT NODES

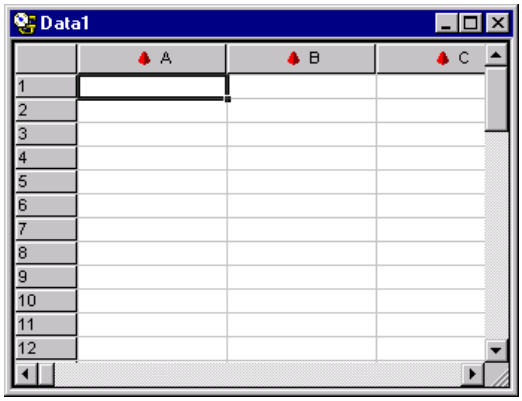
Nodes will be added to your project tree as you work. There will be nodes for data tables, query data views, analysis tasks, SAS code, notes, and results. You can open these nodes by double clicking on them. You can drag and drop these nodes to rearrange them. You can delete nodes you do not want.

PROJECT DATA

The first step in working with an EG project is to create a link to your data. To do this select **Insert→ Data,** or **Tools → Import Data.** You

can also drag and drop data from Windows Explorer. EG can use any SAS data set that has one of the following extensions: sd2, sd7, or sd7bdat. One advantage of EG is you can access various types of data files, i.e. Excel and Oracle, from any platform where SAS runs. The necessary conversions are transparent to the user. However, it is essential that you check your data carefully. Import problems have been observed. These problems occur when the data does not match the assumptions that SAS makes in order to import data transparently. In the vast majority of cases the assumptions are correct, but they may not be correct for your data. Class data sets will be fine, but if you use other data sets it is up to you to verify the data is imported correctly.

Within your project, data will appear in a data table that resembles a spreadsheet. Variable type, character or numeric, is indicated by the symbol beside the variable name. A red pyramid indicates character data and a blue ball indicates numeric data.



Entering data into the data table: Data can be entered directly into the data table. To save this data, select **File → Save** or **File → Save As**. Choose a folder in the **Save in** drop down list and enter a filename in the **File name** box. Data sets are stored as SAS Data Files (*.sas7bdat, V7 Long Names) unless a different file type is selected in the **Save as type** drop down list. If you forget to save this data, you will be asked to save it when you close the data set or project.

Adding an existing SAS data set to your project: **Insert → Data** and click the **Existing** tab. Select a folder from the **Look in** drop-down list. Select the data set and click **OK**, or double-click the filename. To drag and drop a SAS data set into your project, click and hold the left mouse button, drag the file from an Explorer window into the Project window and release the left mouse button.

Adding an Excel spreadsheet to your project: To create a SAS data set from an Excel spreadsheet select **Tools → Import Data** and double click on the Excel icon. Select a folder in the **Look in** drop down list and double click on the filename. If there are multiple sheets in the file, you will be able to choose the sheet you want. Follow the instructions provided for the rest of the steps. You will be able to select variables, rename variables, and add labels and formats. You will need to choose a data set name and storage location.

Excel spreadsheets can also be dropped into the project window. Using this method, a link to the spreadsheet will be added to your project tree instead of a SAS data set. You will not be able to edit this data.

Adding data using SAS code: A data set node will be added to your project tree for data sets created using SAS code. If you use a one level name, the data set will be stored in the WORK data library until you end your project. When you close your project you will be asked if you want to store the data set permanently. It is not required because the data set can be recreated by rerunning the SAS code. However, if you do not save the data set, the associated tasks will not be saved either.

Some instructors provide SAS programs on their Web sites that include both data and code. SAS Institute provides sample programs in the OnlineDoc and on their Web site. These programs can be copied and pasted into the code window. More information is provided in the section **Using the SAS Code Window.**

Note: Other methods for adding data to a project are available but will not be covered because they are more complex. Data problems should be rare if students are provided SAS data sets and they store them in either their SASUSER or EGTASK data library.

Editing data tables: Data tables are opened in Read-Only mode. To change the protection level to Update select: **Data→ Protected**. You will receive a message that asks if you want to change this data to Update mode and indicates that changes made will be applied directly to the data. You will also receive this prompt if you try to edit a data set that is in Read-Only mode.

Some changes made in the data grid can be undone using **Edit → Undo**. However, not all changes to a data grid can be undone. We recommend that you make a backup copy of your data before editing it .

Class data sets are protected so that they cannot be modified. If you need to modify a class data set, you must create a copy of the data set and modify your copy. You can use Windows Explorer to place a copy of the data set into your own file space, but do not change the data set name. To save a copy of a data set from within EG, select **Insert → Data** and highlight the desired data set. Right click on the data set node and select **Save data-set-name As**. Chose a folder in the **Save in** drop down list and click **Save**. Now you need to change the data node so it points to your data instead of the class data. To do this right click on the data node and select **Properties**. Click **Change,** choose a folder in the **Look in** drop down list, double click on the filename, and click **OK**. We were not able to use this method if the data set was stored in the WORK data library.

Exporting Project Data: SAS data sets can be exported and saved in many file formats. To export your data, highlight the data node, right click the mouse button, and select **Save data-set-name As/Export**. Choose a folder in the **Save in** drop down list, choose a file type from the **Save as type** drop down list, and click **Save.**

CREATING A COMPUTED COLUMN

A new column or variable can be created either directly in the data grid or using the Query Builder. Using the data grid method, the new variable is added to the existing SAS data set, but no record of how it was created is stored unless you copy the expression into the **Label** box. With the Query Builder method, a query data view is created that contains all the original data and the new column. The funnel icon in the project tree indicates a query data view. It may be confusing to have data nodes for both the original data set and the query data view. The advantage of the Query Builder is that the SQL code used to create the new column is stored with the query data view. The SQL code can be viewed by checking the **Preview the query code** box in the Query Builder window or by dragging and dropping the query data view into a code window.

Creating a computed column in the Data Grid: Open the data file and place the cursor on the column heading to the left of where you want to add the new column. Select **Insert →Columns** and click the **General** tab. Type the new column name in the **Name** box and the expression in the **Expression** box. We recommend that you copy the expression into the **Label** box so it is stored with the data. Click **OK**.

For example, if Y is an existing column, you could create a new column or variable, LY, using the expression LOG(Y).

Creating a computed column in the Query Builder: Select **Tools→Query→Create from Active Data**, click the **Select and Sort** tab, and click **New**. In the **General** tab, type the name of the new column in the **Alias** box. Now, click the **Expression** tab. You can use the Expression Builder to create the expression or type the expression directly into the Expression box. Click **OK** to add the new column. The new computed column appears in the list of computed columns on the left side of the Query Builder.

FILTERING DATA USING THE QUERY BUILDER

When performing statistical analysis, you may need to delete observations or subset the data. This can be done using the Query Builder. The Query Builder generates SQL code to perform these tasks. Check boxes at the bottom of the Query Builder window allow you to view the query result, code, and log. You can also open a code window and drag and drop the query data view from the project tree into the code window. The SQL code will appear in the code window.

Filtering Data: To filter data, highlight the data node and select **Data→ Filter.** Drag a variable from the left panel to the **Filter Data** window. The **Edit Filter Condition** window will appear to help you build an expression. The data set created will be a query data view. A data view does not actually contain data; it contains instructions on how to get data from the original data set. The funnel icon is used in the project tree to indicate a query data view.

Saving the query data view as a SAS data set: To save the query data view as a SAS data set, check the **Save as Data** box at the top of the Query Builder window. If you do this, two nodes will be added to your project tree; one for the SAS data set and one for the query data view. You will be able to tell them apart by their icon. The output SAS data set will have the name you entered in the **Name** box and it will be stored in the SASUSER library by default. To change the default storage location you can define the EGTASK libref. For instructions, see the **SAS Data Library** section of this paper.

JOINING DATA USING THE QUERY BUILDER

Up to sixteen tables can be joined using the Query Builder. To open the Query Builder window, select **Tools→ Query→ Create from Active Data**. Select the **Tables** tab and add data tables using the **Add Data** button at the bottom. EG will draw join lines between matching variables in the data sets, indicating how it will match the data sets. The diamond on the line will contain the join operator, usually an equal sign

(=). To modify the join, right click on the operator in the diamond and select **Modify Join**.

The default join is an inner join and the resulting data set will contain matching rows only. Other join types can be selected in the **Modify Join** box, which is opened by right clicking on the operator in the diamond. If you are joining two tables, your output choices are matching rows only; all rows from data1; all rows from data2; all rows from data1 and data2. These are referred to as inner, left outer, right outer, and full outer joins respectively. This is explained in detail in the EG online help.

CONCATENATING SAS DATA SETS
To concatenate SAS data sets, select **Data→Append Data**. If you need to rename variables before you append the data sets, this can be done in the data grid.

ADDING NOTES TO YOUR PROJECT
Documentation and comments can be added to your project using notes.
To insert a note in your project select: **Insert → Note.**

PROJECT TASKS
EG contains about 60 stored tasks. These tasks provide a point and click interface to many SAS procedures. Tasks can be accessed using the **Tools, Data, Analysis**, and **Graph** menus. They are also available in the Task window. In this window the task list can be viewed and sorted using the **Task by Category** tab or **Tasks by Name** tab.

Tasks always use the current data set. The current data set is the data set highlighted in your project tree. Many tasks provide an easy way to select analysis and classification variables; procedure options; additional graphic output; and output data sets. Tasks generate SAS code and produce results. Code and results nodes are added to the project tree beneath the task. You may need to expand the project tree in order to see all the added nodes.

THE CODE WINDOW
The EG code window is color-coded and provides syntax checking. To submit code, use one of these methods. To use the pull down menu, select **Code → Run on Local**. To use the toolbar, click the **Run on Local** icon. To use the mouse, click the right mouse button in the code window and select **Run on Local**.

When you run your analysis from a code window, a code node and all the usual nodes will be added to your project tree.

***Viewing EG generated SAS code*:** When you perform a task, EG generates the SAS code. This code can be viewed by opening the code node. At the beginning of the code, EG reports the task; the date and time of execution; the server (such as local PC); and the data source. Reviewing the code is a good way to learn SAS syntax. However, for a beginning user, the SAS code may be overwhelming. EG code contains many options and statements that are not required, especially when graphs are requested. Although this makes the code more complicated it is a good learning tool.

Rerunning EG generated code: The code for any task can be modified and rerun in a code window. To do this, open the code node you wish to repeat. When you attempt to make changes to the code you will receive a prompt that states: "This code has read-only attributes. Do you want to add this as a modifiable code window?" If you choose yes, a new modifiable code window will be opened and added to your project tree. This window will contain the code you want to modify.

To rerun code with a different data set, replace the data set name that follows the DATA= option on the procedure statement. This can be done globally using **Edit → Replace**. If the program creates a SAS data set, you may want to replace this data set name as well. Many procedures use the OUT= option to name output data sets. If you do not replace these names, the new data sets will replace the existing data sets when you rerun the code.

To identify a SAS data set, you must know its full name. One-way to determine the data set name is to highlight the data node in the Project window, click the right mouse button and select **Properties.** The **File name** box will display the full data set name. A second method is to drag and drop the data set node from the Project window to an open code window. A comment that reports the data set name will appear, along with a LIBNAME statement, if one was automatically created when the data was generated. For more information on LIBNAME statements see the **SAS Data Libraries** section or look in **Help**.

Entering and running code: You can enter and run SAS code in the code window. In some cases, this may be easier than using the menus. In some cases, it may be necessary because there is no programmed task in EG for what you need to do.

Copying SAS code from other sources, i.e. the Web: SAS programs that contain data and code can be copied from the other sources,

pasted into a code window, and submitted. This method allows you to run programs provided by your instructor, OnlineDoc, or the SAS Web site. http://www.sas.com/service/techsup/sample/sample_library.html .

To do this, select **Insert → Code** and click the **New** tab. Enter a name in the **Name** box and click **OK**. Copy and paste the SAS program and data into the code window. Right click in the code window and select **Run on Local**.

***Saving your SAS code**:* Your code will automatically be saved in your project tree, but you can save it outside your project by selecting: **File → Save code As.**

Saving all project code: **File → Export All Code.** Exporting all the code is not the same as saving the project. The code cannot re-create the project. Before saving all the code, you may want to delete unwanted nodes. To delete task nodes, right click on the task in the project window and select **Delete**.

RESULTS

Selecting your results style: By default**,** results are stored in HTML format using the EG Default Style. You can select another style from the drop-down list on the toolbar. Results will print much better if you select **Printer** instead of **EG Default**. You do not have to rerun the task, because the HTML file is dynamically updated when you select a new style.

Selecting your results file type: In addition to HTML, results can be generated as text, PDF, or RTF files. RTF files copy very nicely into WORD documents. PDF files have a professional appearance and you can easily print selected pages from Adobe Acrobat. Multiple file types can be produced at the same time. To choose file types and other results options, select **Tools→ Options** and click the **Results** tab.

Using EG Document Builder: You can use the EG Document Builder to combine the results of multiple tasks into a single document, but you cannot add text and ActiveX graphs may not be displayed. This limits the usefulness of the Document Builder. To build a document, select **Tools → Build Document.**

Copying your results into a Word document: You can cut and paste your results into a Word document. We recommend you choose the RTF file type if you plan to do this. Tables and graphs can be edited in Word.

Printing complete task results: To generate a complete listing of the task results, open the results by double clicking on the results node and select **File → Print.**

Printing a single piece of task output: Fully expand the results node in the Project window, open the desired piece of output and select **File → Print**. Notice the entire contents of the window are printed.

Emailing your Results:
To email your project code or data, highlight the node, right click, and select **Send To**.

SAS DATA LIBRARIES

Folders that contain SAS files are referred to as SAS data libraries. Library references, or librefs, are assigned using LIBNAME statements.

SASUSER is a special libref that is automatically defined by SAS and output data sets are stored in this library by default.

WORK is a special libref that is automatically defined by SAS and temporary files are stored in this library.

EGTASK is a special libref, which you can define to identify the library where you prefer EG to store output data sets. If EGTASK is defined, task output data sets will be stored in the defined EGTASK library rather than in the SASUSER library.

To view a list of data sets within assigned libraries, select **File→Open**. Select, **Local→Libraries** in the drop down list. Data sets can be renamed, copied, and deleted from within this window. If other servers are defined they can be found by selecting **File→Open→Servers**.

To determine what librefs are defined and what library they refer to, submit one of the following statements:
LIBNAME _all_ list;
LIBNAME sasuser list;
LIBNAME egtask list;
LIBNAME work list;

To define EGTASK, submit a LIBNAME statement like:
LIBNAME EGTASK “c:\st508”;
Replace ***c:\st508*** with the name of an existing folder where you want output data sets stored.

Note: SAS data sets stored in the SASUSER or EGTASK libraries are considered permanent because they will not be erased by SAS at the end of your EG session. Data sets stored in the WORK library are considered temporary because they will be erased at the end of your EG session.

To facilitate transparent data access, EG defines temporary librefs when you insert data into your project from a library that has not been previously identified. These librefs begin with the prefix "EC"; for example, ECLIB000. You will notice these librefs when you look at the SAS code, or if you drag and drop a data node into a code window.

EXERCISE

Before beginning, please change the following EG settings. All these settings will remain in effect between EG sessions, except defining EGTASK. EGTASK has to be defined at the beginning of each EG session. EG may not use EGTASK if it is not defined at the beginning of the session.

Turn the agent sound off: By default, an agent, called Genie, will appear to assist you. The agent provides helpful hints, but the sounds can be distracting. To turn the agent sound off, right click on the agent, and select **Advanced Character Options**. Clear the **Play spoken audio** and **Play character sound effects** check boxes. You may also want to set the **Speaking speed** to its fastest setting. This will improve the typing speed.

Change the style from EG* Default *to* Printer*: Select **Tools → Options** and click the **Results** tab. In the Style box, replace **EG Default** with **Printer** and click **OK**.

Select RTF output as well as HTML: This is only necessary if you want to paste your results into a WORD document. To choose the RTF file type or set other results options, select **Tools→ Options** and click the **Results** tab. Make your selections and click **OK**.

Define EGTASK*:* We recommend you define the EGTASK library at the beginning of each EG session so output data sets will be stored in your SAS data library. For instructions, see the **SAS Data Libraries** section of this paper.

EXERCISE 1

Objective*:* Perform Linear Regression using Crime Data

Data Description: This data set contains several variables from the statistical abstracts of the United States for the 50 states and District of Columbia (DC).

Variables: Murder Rate, Violent Crime Rate, Metropolitan Residents %, White %, High School Graduates %, Poverty Rate % and Single Parent %.

Data Source: Statistical Methods for the Social Sciences by Alan Agresti and Barbara Finlay, chapter 9.

Add the crime data set to your project and create the following:

1. Scatter plot of Murder versus Poverty (Task: Graph → Line)
2. Boxplot for Murder (Task: Analysis → Descriptive → Distribution Analysis)
3. Regression of Murder versus Poverty. Include the following plots: Observed vs. Predicted, Observed vs. Poverty with Prediction Line, and Residual Plot (Task: Analysis → Regression → Linear);
4. Omit the DC observation from the data set and repeat the Linear Regression (Task: Data→Filter then repeat step #3)
5. Prepare a document of your results; be sure to include your interpretation.

CONCLUSION

EG greatly simplifies data access. Students should be able to easily access a wide variety of data types from various platforms with few if any problems. It is important to realize that no software can deal with every possible data file, so it is up the student to verify that the data is imported correctly.

EG greatly simplifies many statistical tasks. Students should be able to generate statistical output without having to learn SAS syntax. This will allow the students to concentrate on their data and learning statistics.

EG greatly simplifies the creation of SAS graphs. SAS/GRAH code is difficult, so only dedicated students attempt to learn SAS/GRAPH. However, the simplicity with which graphs can be created in EG will lead to their increased use. The use of graphs will enhance the data analysis and interpretation of statistics.

EG should make simple data management tasks easy to perform. However, students with little data analysis experience will probably need instructions on joining data, filtering data, and creating new variables. Learning the SAS code for these tasks would probably be harder than learning the EG methods.

EG provides students an interactive and fairly convenient way to learn SAS programming. The generated code can be viewed in the code window. The generated code usually contains many options and statements that are not required. These are interesting to a programmer who is comfortable with SAS code, but for a beginning SAS user, they probably make SAS seem more complex than it has to be. Students can also copy and run sample programs that are available from many sources. EG will provide very limited exposure to the data step.

Understanding the data step is definitely a core concept in SAS programming. On the other hand, EG will provide exposure to SQL.

Traditional SAS programmers often rely on their programs and log files for documentation. EG provides logs for the tasks, but not for data sets created from Query Builder or modified in the data grid. It is true that the SQL code is saved with the query data view, but no code or log is saved with the SAS data set.

EG will probably be useful to graduate students and researchers at NCSU. However, because not all analyses are available as tasks, they may also need to write code. Students often run large simulations and batch jobs. We are not sure how these will work in our environment using EG. It will also be interesting to see how EG changes the consulting we do with these clients.

EG provides a rich selection of results file types; such as HTML, txt, PDF and RTF. This variety of file types provides many options for reporting results. We do not believe that the Document Builder is full featured enough for our needs, but the smooth incorporation of RTF files into Word documents will certainly be used extensively. Some campus members will use the HTML file formats for reporting on the Web. The email feature will be useful, especially to students who need to share their results with professors or committee members.

We intend to introduce EG in a statistics class before the meetings. We plan to evaluate whether EG can be used to provide beginning statistics students with a relatively painless introduction to SAS. We hope that EG will do this and that we will see the use of EG and SAS increase in these classes in the future.

REFERENCES

Rigsbee, Carol and Hemedinger, Chris (2001) "Delivering Information to the People Who Need to Know". Proceedings of the 26th SAS Users Group International Conference, Paper 145-26.

TRADEMARKS

ACKNOWLEDGMENTS

We sincerely appreciate Terry Bryon, Joe Wells, and Laura Grady for helping us get the software, installing it on the distributed platform, and answering questions related to its use in this environment. Thanks to Dr. Jackie Dietz for the information on how she uses SAS in her introductory statistics classes. Thanks to Bill Sawyer of SAS Technical Support for his patient and thorough responses to our questions.

CONTACT INFORMATION

Sandy Donaghy
Email: sandy_donaghy@ncsu.edu

Joy Smith
Email: joy_smith@ncsu.edu

Fuzzy Key Linkage

Robust Data Mining Methods for Real Databases

Sigurd W. Hermansen, Westat

Abstract

Results of data mining depend heavily on the quality of linkage keys within a search dataset and within its database target. Linkage failures due to errors or variations in linkage keys have few symptoms, and can hide or distort what data have to tell us. More robust methods have promise as remedies, but require careful planning and understanding of specialized technologies. A tour of fuzzy linkage issues and robust linkage methods precedes a review of the results of a recent linkage project. Sample SAS programs include tools and tips ranging from SOUNDEX() and SPEDIS() functions to hash indexing macroprograms.

Introduction

Relational Database Management Systems (RDBMS's) have evolved into a multi-billion dollar industry. In no small part the industry has succeeded because RDBMS's protect the integrity and quality of data. Large organizations have committed huge sums of money and many person hours to enterprise RDBMS's. But while typical RDBMS's effectively repel any attempt to insert duplicate key values in data tables and subvert database integrity, they remain remarkably vulnerable to other types of errors. Most obvious of all, linkage of an insert or update transaction to a database fails whenever the search key in the transaction fails to match bit-by-bit the target key in the database. If a transaction key contains the person ID US Social Security Number (SSN) of 105431002, for instance, instead of the correct 105431802, it will fail to link to the corresponding record for the same person. Correct linkages of tables in an RDBMS depend entirely on the accuracy of columns of data used as key values. Errors in the face values of keys, whatever the sources, not only lead to linkage errors, but also persist. Once admitted to a database, errors in keys seldom thereafter appear on the radar screen of a system administrator.

Do errors in primary and foreign keys actually occur in real databases? Pierce (1997) cites a number of reports indicating that in the early 1990's a near majority or better of US business executives recognized data quality problems in their companies. Arellano and Weber (1998) assert that the patient record duplication rate in single medical facilities falls in the 3%-10% range. Many who have assessed the accuracy of the US SSN as a personal identifier in federated databases, including the author, peg its accuracy at somewhere between 93% and 97%. These estimates suggest a 5% ±2% rate of error in attempts to link transactions or events to a master database .

Failures of keys to link properly have more impact where analysts are mining data for a few nuggets of information in a mountain of data, or where access to critical data requires a series of successful key links. In both situations, errors in keys propagate. Consider how a 1% key linkage failure rate propagates over a series of key links required for a basic summation query [SAS PROC SQL syntax],

```
SELECT DISTINCT Person_ID,
SUM(amount) as OUTCOME
FROM Events GROUP BY Person_ID;
```

Key linkage failures may hide the skew of the true distribution. Even small rates of errors produce bias and outliers, such as the summary of amounts per group by a count of related events (GT10), as shown below.

ID Group		GT10		
true	in DB	true	in DB	amount
10111	10111	T	T	300
10111	10111	T	T	100
13111	**12111**	T	F	200
10111	10111	T	T	100
12111	12111	F	F	100
13111	13111	T	T	100
10111	**18111**	T	F	400

Result of summation query:

	OUTCOME	
GT10	true	computed
T	1,200	600
F	100	700

Errors can affect both the counts of related events and the amounts being summed. Chains of keys that link data in subsidiary tables to master records, typical in SQL views, prove even more vulnerable to errors. Transposing digits in a short integer key likely converts one key into another key value already used to link a different set of data, as in

Table T1

ID	status
21	Negative
12	Positive

Table T2

ID2	ID
43	21
34	21*

Table 3

ID3	ID2	subject
76	43	patientXYZ
67	34	patientRST

*in truth, T2.ID=12

VIEW V1:

```
SELECT T23.subject,T1.status FROM
T1
  INNER JOIN
(SELECT T2.ID,T3.subject AS subject
  FROM T2 INNER JOIN T3
  ON T2.ID2=T3.ID2) AS T23
ON T1.ID=T23.ID;
```

In this case, a transposition in T3.ID links PatientRST to "Negative" and not to the correct value of "Positive", yet does not trigger a referential integrity constraint. The RDBMS validation scheme fails and the error remains unnoticed. Key linkage errors such as these undermine the efforts of data miners to extract interesting and important information from data warehouses and distributed databases. Many database administrators may have good reason to believe that critical identifying keys in their databases have much lower error rates. Others, whose databases support applications such as direct marketing, might view a 5% linkage failure rate as perfectly acceptable. All others need to consider more robust linkage methods. Knowledge is power, but bad information quickly pollutes a knowledge base.

Alternative Linkage Keys

Robust linkage methods take advantage of alternative key patterns. An alternative key may work when linkage on a primary key pattern fails. If linkage on a 10-digit integer key fails, for instance, an alternative key consisting of standardized names and a date of birth could have a reasonable chance of being a correct match. So would other alternatives, such as a partial sequence of digits in a numeric identifier combined with either a standardized last name or a month and year of birth. Others, such as a match on first name and zip-code, would not.

Alternative linkage keys have to meet at least a couple of basic requirements. First and foremost, a key has to have a fairly high degree of discriminatory power. A weak identifier often finds too many matches that contain too little information to rule them out, much less verify them. Second, the alternative key has to have a good chance of linking correctly when the primary key fails to link. Two alternative linkage keys with independent 5% error rates, for example, have an expected joint failure rate of 0.25% or only 1/20th the rate of either taken alone. For independent 1% error rates, the combined rate falls to 1/100th of the rate of either taken alone. Fuzzy key linkage gains much of its power by providing alternatives that we would not need in a world of perfect information, yet, in the real world prove necessary to prevent costly linkage failures.

Because linkage failures present no obvious symptoms in a typical database system, the information that these failure hide often surprises clients. As data miners' close cousins, statisticians, know all too well, it takes a lot more evidence and effort to build a good case for a finding that goes against the grain of conventional wisdom, but it scores a lot more points. To compete effectively with an established database administration group, a data miner needs to offer alternatives to routine methods.

Nonetheless, any scheme that involves alternative linkage keys inevitably creates problems for database programmers and, by extension, database clients. The latter group includes not only persons who depend on enterprise RDBMS's for information, but also clients of networks, of Internet search engines, and of wireless services. These groups are growing rapidly and becoming increasingly dependent on fuzzy key linkage for information. Who among us has not found it frustrating to search through results of a Web search and still not find a Web page that should be there? A Boolean alternative (x OR y) search may find a few more relevant pages, but it often buries them in an ocean of irrelevant pages. In Silicon Valley speak, the sounds of terms associated with robust database searches, "disjunctive query" (Claussen et al, 1996), "iceberg query" (Fang et al, 1998), "curse of dimensionality"(Beyer et al, 1999), "semi-structured data" (McHugh et al, 1977), forewarn us of the computational burden of alternative key linkage.

Of course a decision to integrate alternative linkage keys into database access does not settle the issue. A data miner must also choose the right degree of fuzziness in key linkage. Suppose a data miner uses a search key to locate instances of something that occurs at a rate of approximately one-percent in a database. If the data miner selects an alternative key that matches in error to 1% of the same database, the specificity of key linkage cannot exceed 50% on average. For each true match selected, fuzzy linkage would select on average one false match.

Fuzzy Linkage Methods

Fuzzy key linkage has at least one thing in common with drug therapy. A few "active ingredients" (AI) help alleviate the problem of errors in linkage keys, but each has side-effects that have to be managed carefully with "buffering agents" (BA). The active

ingredients in fuzzy linkage increase dramatically the time and resources needed to compare two sets of records and determine which records to link. The buffering agents do everything possible to make up for losses of efficiency and bring the linkage process sufficiently up to speed to make it feasible.

The order in which different active ingredients get used in the linkage process proves critical. Initial stages of linkage have to strip irrelevant data from key values and filter data as they are being read into buffer caches under operating system control, and do so before the linkage program moves them into working memory or disk space.

Reduced Structure Databases and Data Shaping (AI)

As the scale of databases and the dimensions of alternative keys increase, the idea of loading all key values into a single database, much less contiguous memory, becomes increasingly an academic fantasy. A more realistic linkage model leaves very large data objects in place, outside the key linkage application, and lets the linkage program select only key values and relevant data for processing within the application.

Alternative keys usually represent a dimension of the real world, such as place, interval of time, and other context cues, plus event outcomes, attributes, or other facts that in some sense belong to an entity. In distributed or federated databases, alternative key values retain their meaning while integer key values removed from the context of a RDBMS lose their meaning. An integer makes a good employee ID in an enterprise database, but a poor ID for a person in a database that spans enterprises.

The so-called Star Schema for data ware-housing makes it easier to develop a logical view of data that includes alternative and overlapping information. Earlier articles, especially Hermansen (2000), present ideas for implementing alternative keys, disentangling data from file systems, and restructuring databases into forms that better support alternative logical views.

Real database case study(1): *A database contains over 10 million records of blood donations. The number of donation records per donor varies from one to several hundred. Multiple observations of donor demographics show surprising variations within sets of records for individual donors. Related donation and donor tables allow full capture of observed responses by donors as well as most likely attributes based on modes of multiple responses. The accuracy of the donor database improves over time as true responses have a better chance of repeating than random errors.*

Compression, Piping, Filtering, and Parallel Processing (BA)

No way around it: alternative linkage keys crowd whatever bandwidth a network and OS have to offer. Even bit-mapped composite key indexes become unwieldy. Multiple columns containing names, addresses, date/times, category labels, and capsule descriptions replace neat, continuous sequences of nine-digit ID's.

Harry X Lime 1923256 ……Vienna Austria
replaces
105342118

Practical remedies include

1) *compression of data streams on a database server and decompression in a pipe that an linkage program reads:*

   State-of-the-art mainframes implement data compression and piping transparently. Smaller machines leave it up to the programmer to compress data files and set up pipes to decompress them in stream. In the SAS System (Unix in this case) the FILENAME statement,

```
FILENAME zipPipe PIPE 'gzcat <file
   path(s) with .zip* extension>';
```

   reads zipped files through an INPUT process. The programmer can enter a list of file names in a specific order, or specify a regular expression that yields a list. (The last asterisk in *.zip* may only prove necessary when files have hidden version numbers.) In either case the source data files remain in place while data stream into the linkage program. When reading a very large set of records with a SAS program, a pipe often works faster than inputting data directly from an intermediate SAS dataset;

2) *filtering data while cached in memory buffers, before they move to the working storage that a linkage program allocates:*

   In the SAS System, an INPUT statement in a DATA STEP VIEW and a PROC SQL statement referencing the DATA STEP VIEW in a FROM clause caches data in memory where a SQL WHERE clause acts as a filter. Only data that meet initial conditions pass through the filter and enter a SAS WORK dataset;

3) *extracting minimal subsets of data from database servers using views:*

   As a rule a database server does a more efficient job than an application program of handling basic operations on its data tables. A SQL SELECT statement or equivalent in a stored view has primary access to indexes, integrity constraints, and other metadata of the database object, and it executes in an environment tuned to allow quick access.

4) *running data extraction programs in parallel on multiple processors, or even on multiple database servers:*

   Some database systems allow more than one thread of a key linkage program to execute in parallel on different processors. The MP CONNECT procedure under the SAS/CONNECT® product, for example, lets the programmer RSUBMIT different sections of a program to different processors on a SAS server, or to different database servers, where they can execute in parallel. Doninger (2001) and Bentley (2000) describe the benefits of parallel processing with MP CONNECT and include examples. Parallel execution of views on different database servers, for example, makes good use of this new feature of SAS Version 8.

Real database case study (2): *A database programmer reported recently on SAS-L that subsetting data into a SAS dataset via a DB2 view cut CPU time to 11% of that required to read the full database and then subset it. It also reduced elapsed time by a factor of six (see SAS-L Archives, subject: RE: SQL summarization question, 12/14/2000).*

Data Blurring, Condensing, and Degrees of Similarity (AI)

A linkage key, at least after encoding for storage in a digital computer, amounts to nothing more than a pattern of bits. To attain a higher degree of specificity in key linkage, one must either add information (more bits) or simplify the pattern (using some form of metadata template for valid patterns). To attain a higher degree of sensitivity of key linkage, one must either suppress information (mask bits) or simplify the pattern. Greater specificity means fewer false links among keys; greater sensitivity means fewer failures to find true links. Suppressing bits in a key pattern prior to comparing two keys obviously risks trading more false links for fewer failures to find true links. Confining a sequence of bits (a field in a record) to a limited domain has some chance of increasing sensitivity of linkage by reducing meaningless variations in keys related to the same entity. Fuzzy key linkage attempts to achieve better sensitivity of linkage with the least loss of specificity.

Although the term "fuzzy", as in "fuzzy math", suggests a vague or inconsistent method of comparison, fuzzy key linkage actually provides more precise and consistent results of comparisons. Fuzzy key linkage resolves comparisons of vague and inconsistent identifiers, and does so in a way that makes better use of information in data than bit-by-bit comparisons of keys. It simply takes a lot more time and effort to eliminate alternatives.

Real database case study(3): *The article by Cheryl Doninger cited above appeared first under the name "Cheryl Garner". Under the SAS Web page, "Technical Documents: SAS Technical Support Documents--TS 600 to TS699", a search on the criterion "multiprocessing AND Garner" produced nothing, but a search on the alternative "multiprocessing AND Cheryl" located the correct document.*

Operators and functions specifically developed for fuzzy key linkage make it easier to compare certain forms of alternatives. Each of three general types has a special purpose.

"Blurring" and "condensing" functions transform instances in a domain of values into a domain that has a smaller number of distinct values. Blurring maps similar values to one value; it reduces incidental variation in a key. Condensing reduces the remapped values to a set (distinct values). The SOUNDEX() function or operator (=*), for instance, condenses a set of surname strings to a relatively small number of distinct values:

SURNAME	SOUNDEX
Neill	N4
Neal	N4
Neil	N4
Niell	N4
Neall	N4
Nil	N4
Nel	N4
Nill	N4
Nell	N4
Nilson	N425
Nelson	N425
O'Neil	O54
O'Neal	O54
Oneill	O54

Blurring and condensing facilitate indexing of keys. An index on a blurred and condensed key occupies less bandwidth and memory, and it clusters similar

key values. A SOUNDEX() transform of any of the nine similar surnames beginning with an “N” and ending with an “L” (above) will match to an index containing “N4”.

A “degree of similarity” operator or function compares two key values and produces a numeric value within an upper and lower bound. The upper bound indicates identical keys; the other bound indicates no similarities. Combined with a decision rule, usually based on an explicit, contextual, or default threshold value, the fuzzy operator or function reduces to a Boolean. It aggregates the results of comparisons of alternative keys into a numeric score, accepts as true links those with scores that exceed a threshold, and rejects the others. As an example, the SAS SPEDIS() or “spelling distance” function calculates a cost of rearranging one string to form another, where each basic operation used to rearrange the string has a cost associated with it. A CASE clause in SAS SQL implements SPEDIS() in a way that sets a neutral value of 0.4 should either of two US SSN strings turn up missing, and a value in the range of zero to one if the comparison goes forward.

```
case  when t1.&SSN1="" or
t2.&SSN2=""
   then 0.4
   else max((1-length(t1.&SSN1)*
 pedis(t1.&SSN1,t2.&SSN2)/200)),0.1)
end as SSNcost
```

A programmer can use the calculated variable SSNcost in a Boolean “OR” expression, as in

```
WHERE (calculated SSNcost > 0.5
      AND t1.surname=t2.surname)
       OR (calculated SSNcost > 0.8
      AND t1.frstname=t2.frstname),
```

to implement linkage on alternative key patterns, or combine it with another degree of similarity, to achieve the same goal.

So-called “regular expressions” and other pattern-matching operators and functions (such as the SAS INDEX() function) normally point to a location in a string or file, or return a "not found" value. This feature in particular facilitates checks for alternative patterns in strings, numbers, or other semi-structured elements in databases. Rhoads (1997) demonstrates practical uses of pattern-matching functions. These include tests for a match on a template. Extensions, such as a series of searches for the location of a phone

Data Standardization, Cleansing, and Summaries (BA)

Specialized programs for "mailing list hygiene", standardizing codes, parsing text into fields, and other database maintenance tasks are beginning to appear in greater numbers and variety each year. Patridge (1988) and the www.sconsig.com Web site offer both free and commercial database standardization, cleansing, and summarization programs. Pre-processing can obviously reduce the risk of fuzzy linkage failures and false matches. Partially for that reason, database administrators are paying more attention to data quality metadata, including audit trails. Best practice combines data quality control with robust key linkage methods. To help fill in that niche, SAS® has recently purchased Dataflux and its Blue Fusion and dfPower Match standardization and fuzzy linkage products.

A number of data warehouse developers are rediscovering that old warhorse, the SAS PROC FREQ, and even more sophisticated stuff such as stratified sampling, linear regression, and cluster analysis. Linkage quality really comes down to keeping expected costs of errors within limits.

Real database case study(4): *In a database of >5M blood donation records linked by a non-informative donor ID to 1.5M donors, we grouped by donor and identified unusual sequences of screening test results. We separated out borderline cases, verified the results of database searches, estimated 0.05% (95% CI 0-1.5%) frank technical errors in data management, testing systems, and process controls (Busch et al, 2000), and recommended process enhancements in the blood collection industry to help detect and prevent false-negative results.*

Blocking and Screening on Synthetic, Disjuctive Keys (AI)

RDBMS performance depends heavily on indexes of search keys that clients use to link database records to transactions. As volumes of data approach the limits of a database, platform, and network, indexes take on an increasingly important, and constraining, role. Indexes bound a search on that index to a small block of key values. A *deus ex machina* database tuner can conjure up indexes to optimize transactions, but in very large databases searches on alternative keys mire in quicksand.

"Blocking", an old trick in record linkage methodology, implements a series of searches on partial primary key candidates. A clever blocking

scheme might have an initial search on surname and date of birth, followed by search on date of birth and postal code, and finally a search on surname and postal code. Later stages consider only new candidates for links, not those confirmed as correct links in a prior stage. A good blocking strategy implements alternatives (surname AND DOB) OR (DOB and PostCode) OR (surname and PostCode) so that errors or variations in any one field do not cause linkage failures. The fact that each block consists of a conjunctive (AND) query means that an index can keep the computational burden of an indexed search within bounds.

Blocking has one major disadvantage. It takes multiple passes through a set of data to complete the screening process. When searching databases with many millions of rows in a single table, a single-pass solution makes better sense.

A better screening strategy defines a "synthetic key" for each block and creates an index for each. Each of the synthetic keys represents an alternative linkage key or fragments of an alternative key. Table 1 provides a picture of definitions of eight keys (columns) synthesized from eleven fragments or transforms of alternative keys.
Whether character or numeric, key patterns reduce to strings of bits. By design each synthetic key has sufficient discriminatory power to bind only to similar key patterns, but relatively narrow bandwidth. For instance, it takes just seventeen bytes to represent a first initial of a first name, a soundex transform of a surname, and a Julian DOB. On anything larger than a low-end PC, a number of such indexes will fit

Table 1: Synthetic Linkage Keys

	k_1	k_2	k_3	k_4	k_5	k_6	k_7	k_8
SSN				✓				
SSN5					✓			
SSN4						✓		
LN	✓	✓			✓	✓		
SLN			✓				✓	✓
FN	✓	✓						✓
FN1			✓		✓	✓	✓	
MI								
DOB			✓		✓	✓		
DOB*	✓						✓	

Glossary:

k_i : synthetic search keys
SSN5: digits 1-5 of SSN in decreasing order;
SSN4: digits 6-9 of SSN in decreasing order;
LN: last name;
SLN: yield of Soundex(LN);
FN: first name;
FN1: first letter of first name;
MI: middle initial (not used in screening);
DOB: date of birth:
DOB* date of birth +/- (U/L) 32 days;
MDX: day and month of birth;
Sex: (not used in screening)

into addressable memory. It then becomes technically feasible to

- transform and synthesize k alternative keys or fragments of keys from a moderately large (say, 100K rows) search dataset and build multiple indexes;
- load all of the indexes into memory and hold them there;
- scan a huge dataset one row at a time, transform and synthesize alternative keys for that row, and check each synthetic key against its corresponding index;
- select from the huge dataset only those rows linked to any one or more of the indexes.

The indexes implement a disjunctive “query flock” (Tsur, 1998) that screens for possible links during one pass through a huge dataset. Rows of data that fail to match any of the multiple key patterns pass through the screen. Those that match at least one of the patterns get set aside for further attention.

Multiple, Concurrent Key or Hash Indexes and Rescreening (BA)

Clearly a flock of indexes has to be compact to load into memory and easy and quick to search. The balanced B-Tree indexes used in RDBMS’s conform to the first constraint. In early attempts to implement screening on multiple indexes, we read data from files and used them to write SAS FORMATS. If the SAS expression PUT(key,ndxi.) yielded a “+”, the key value matched the i^{th} index. This effective implementation in SAS of a B-Tree index, called “Big Formats” on SAS-L, worked surprisingly well. Nonetheless, subsequent postings by Paul Dorfman on SAS-L proved and demonstrated that hash indexes implemented in SAS worked much quicker. Testing of hash indexes against big formats, data step merges, and SQL query optimizations, by Ian Whitlock and others, established the equivalence or superiority of

Dorfman's hash indexes, and the dramatic improvements they bring to linkage of very large volumes of data.

Though ideal choices in theory, hash indexes prove cryptic and difficult to specify. Almost all of the SAS programmers who had a penchant for refining Knuth's sorting and searching algorithms have by now found jobs in Palo Alto or Seattle, and are writing C++ object classes and Java applets. Fortunately, Dorfman and a few others have remained loyal to the craft. To make it easier for the rest of us, Dorfman has written SAS macro-programs %hsize, %hload, and %hsearch:.

```
%macro hsize (data=,hid=,load=.5);
   %global z&hid;
   data _null_;
      p = ceil(nobs / &load);
      do until (j = u + 1);
         p ++ 1;
         u = ceil(sqrt(p));
         do j=2 to u;
            if mod(p,j) = 0 then leave;
         end;
      end;
        call
        symput("z&hid",compress(put(p,best.))
        );
        put "info: size computed for hash
             table &hid is " p +(-1) '.';
      put;
      stop;
      set &data nobs=nobs;
   run;
%mend hsize;

%macro hload (data=,hid=,key=,pibl=6);
   %global t&hid p&hid ;
   %local dsid nvars found varfound vname
        varnum vnum;
   %local keytyp keylen xpibl;
   %let dsid  = %sysfunc(open(&data,i));
   %let nvars = %sysfunc(attrn(&dsid,nvars));
   %let varfound = 0;
   %do varnum=1 %to &nvars;
        %let vname=
          %sysfunc(varname(&dsid,&varnum));
        %if %upcase(&vname) = %upcase(&key)
           %then %do;
               %let varfound = 1;
               %let vnum = &varnum;
            %end;
   %end;
   %if &varfound = 0 %then %do;
      do;
         put "error: key=&key variable not
              found in dataset &data..";
         abort;
      end;
      %let rc = %sysfunc(close(&dsid));
      %goto mexit;
   %end;
   %let keytyp =
        %sysfunc(vartype(&dsid,&vnum));
   %if %upcase(&keytyp) = %upcase(c) %then
        %let keytyp = $;
   %else %let keytyp = ;
   %let t&hid = &keytyp;
   %let keylen   =
        %sysfunc(varlen(&dsid,&vnum));
   %let rc       = %sysfunc(close(&dsid));
   %if &pibl > 6 %then %then %do;
      %let  pibl = 6;
      %put info: maximum value for pibl=
          exceeded. pibl=&pibl assumed.;
   %end;
   %let p&hid = &pibl;
   do;
      array h&hid (0:&&z&hid) &keytyp &keylen
         _temporary_;
      array l&hid (0:&&z&hid) 8
        _temporary_;
      ___r = &&z&hid;
      eof = 0;
      do until (eof);
         set &data (keep=&key) end=eof;
         %if &keytyp = $ %then %do;
              ___h =
    mod(input(&key,pib&pibl..),&&z&hid) + 1;
         %end;
         %else %do;
            ___h = mod(&key,&&z&hid) + 1;
         %end;
         if l&hid (___h) > . then do;
            l&hid:
            if &key = h&hid(___h) then
               continue;
            if l&hid(___h) ne 0 then do;
               ___h = l&hid(___h);
               goto l&hid;
            end;
            do while (l&hid(___r) > .);
               ___r +- 1;
            end;
            l&hid(___h) = ___r;
            ___h        = ___r;
         end;
         h&hid (___h) = &key;
         l&hid (___h) = 0;
      end;
      eof = 0;
      drop ___h ___r ;
   end;
   %mexit:
%mend hload;

%macro hsearch (hid=, key=, match=);
   do;
      drop  ___h;
      &match = 0;
      %if &&t&hid = $ %then %do;
          ___h =
mod(input(&key,pib&&p&hid...),&&z&hid) + 1;
      %end;
      %else %do;
          ___h = mod(&key,&&z&hid) + 1;
      %end;
      if l&hid(___h) > . then do;
         s&hid:
         if h&hid(___h)=&key then &match = 1;
         else if l&hid(___h) ne 0 then do;
            ___h = l&hid(___h);
            goto s&hid;
         end;
      end;
   end;
%mend hsearch;
```

The data= parameters require the name of a SAS dataset or view. The hid= parameters name the specific index being sized, loaded, and searched. The key= parameter identifies SAS variables that contain a value of the synthetic key either written to or matched to the index. The match= parameter names the Boolean result of an index search. These program excerpts show actual calls of %hsize, %hload, and %hsearch:

```
%hsize(data = mdx , hid = mdx , load = &lf );
data rslt.headid ….
       .
```

```
    .
%hload(data=mdx,hid=mdx,key= mdx,pibl=6);
    .
    .
    .
do until (eof);
  infile header end=eof;
  input
  @ 01 rtype       $char01.
    .
    .
    .
 %hsearch(hid= mdx,key=mdx,match=m_mdx );
    .
    .
    .
```

After screening, all of the rows of data selected from the huge dataset link to at least one of the rows of data in the search dataset, but some rows in the search dataset may not have linked to any of the rows in the huge dataset. A simple reversal of the screening process selects only those rows of data in the search dataset that match at least one row in the results of screening. We call this step "rescreening".

Where each row in the huge table has a very small chance of being a correct link, early elimination of unlikely candidates for linking greatly reduces the costs of later stages of the linkage process. Screening and rescreening cut large datasets down to manageable size.

Fuzzy Scoring and Ranking on Degrees of Similarity of Linkage Keys (AI)

Once screening has reduced a huge target dataset to, say, a mere million or so rows, and rescreening has reduced the number of rows in the search dataset by perhaps fifty to eighty percent, more intensive linkage methods become feasible. So-called probabilistic methods assign weights for individual field matches and mismatches for each pair of records, and sum the logs of these weights across fields. Estimated error rates in correct links, given a field match, and estimated coincidental links, given field frequencies, determine the composite weight or score. The higher the score for a pair of records, the higher the linkage program ranks them.

Probabilistic linkage and related methods have evolved over a span of some forty years into statistical modelling for control of both linkage failures and incorrect links. Winkler (2000) assesses the current state of record linkage methodology. Proceedings of a recent conference (Alvey and Jamerson, eds. 1997) on record linkage includes a historical perspective on development of specialized key linkage programs: OX-LINK (Oxford Medical Record Linkage System), GIRLS (Generalized Records Linkage System), from Statistics Canada; and, AUTOMATCH, originally developed by Matt Jaro at the US Bureau of the Census,

A relatively simple scoring program requires some guesswork about the values to assign to field match and mismatch weights and to a cut-off score. The values of weights generally increase with the relative importance of a field match to the chance of a correct match. Neutral weights for missing values help us focus on whatever subset of information a row of data contains.

In this implementation of a simple scoring program, a SAS macroprogram allows a user to assign variables names as parameters.

```
*** MATCH PROGRAM ***;

%macro mtch(    DSN1=  ,    DSN2=          ,
            SubmitID=  ,SourceID=          ,
                SSN1=  ,    SSN2=          ,
            LstName1=  ,LstName2=          ,
            FstName1=  ,FstName2=          ,
                 MI1=  ,     MI2=          ,
                Sex1=  ,    Sex2=          ,
               Race1=  ,   Race2=          ,
                DOB1=  ,    DOB2=          ,
                DOD1=  ,    DOD2=          ,
            Zipcode1=  ,Zipcode2=          ,
            Zipcode3=    ,Zipcode4=        ,
            RECCODE1=    ,RECCODE2=
,
              STATE1=    ,  STATE2=
,
                 key1=   ,    key2=
,
             keyval1=    , keyval2=
,
             Rectype=    ,  OutDSN=
,
                   c=           );


 proc sql;
  create table &OutDSN as
  select t1.&SubmitID as SubmitID,
         t2.&SourceID as SourceID,
        t1.&STATE1 as STATE,
        t1.&RACE1 as RACE,
        t2.&RECCODE2 as RECCODE,
        t1.&Zipcode1 as ZIP,
        t2.&Zipcode2 as ZIPX,
        t2.&Zipcode4 as ZIPP,
case when t1.&SSN1=t2.&SSN2 and t1.&SSN1 ne
          "000000000" and
(soundex(UPCASE(t1.&LstName1))=
          soundex(UPCASE(t2.&LstName2))
          or t1.&DOB1=T2.&DOB2) then 1.0
     when t1.&SSN1=t2.&SSN2 and t1.&SSN1 ne
          "000000000"         then 0.5
     when
index(UPCASE(t2.&FstName2),substr(UPCASE(t1.&
FstName1),1,3)) and
soundex(UPCASE(t1.&LstName1))=soundex(UPCASE(
t2.&LstName2)) and t1.&DOB1=T2.&DOB2 then 0.2
     when
    UPCASE(t1.&FstName1)=UPCASE(t2.&FstName2)
     and t1.&Sex1="F1" and t1.&DOB1=T2.&DOB2
                              then 0.05
                              else 0
end as bonus,
case when t1.&SSN1="000000000" or
               t2.&SSN2="000000000" then 0.4
                                 else max((1-
               (length(t1.&SSN1)*spedis(t1.&S
               SN1,t2.&SSN2)/200)),0.1)
end as SSNcost,
```

```
case when
       UPCASE(t1.&LstName1)=
       UPCASE(t2.&LstName2)
                                then 0.9
     when soundex(UPCASE(t1.&LstName1))=
          soundex(UPCASE(t2.&LstName2))
                                then 0.6
     when T1.&Sex1 = "F1"       then 0.4
                                else 0.1
end as SDXLN,
case when
    UPCASE(t2.&FstName2)=UPCASE(t1.&FstName1)
                                then 0.9
     when index(UPCASE(t2.&FstName2),
  substr(UPCASE(t1.&FstName1),1,3))
                                then 0.6
     when index(UPCASE(t2.&FstName2),
   substr(UPCASE(t1.&FstName1),1,1))
                                then 0.4
                                else 0.2
end as FN2,

%if (&MI2 ne) %then
    case when substr(UPCASE(t1.&MI1),1,1)=
substr(UPCASE(t2.&MI2),1,1)   then 0.8
                                else 0.2
    end as MI1,
 ;
 %if (&Sex2 ne) %then
    case when t1.&Sex1=t2.&Sex2
                                then 0.5
                                else 0.2
    end as SexMtch,
 ;
 case   when t2.&DOB2 = t1.&DOB1
                                then 0.7
        when month(t1.&DOB1)=month(t2.&DOB2)
and day(t1.&DOB1)=day(t2.&DOB2)
                                then 0.6
         when t2.&DOB2 <= 1.05*t1.&DOB1
          and t2.&DOB2 >= 0.95*t1.&DOB1
                                then 0.4
                                else 0.2
   end as BtwnDOB,
   %if (&SourceID ne SSNC)
        %then t2.&SSN2 as SSNX, ;
              t1.&SSN1 as SSNC,
            t1.&LstName1,t2.&LstName2 as LNX,
            t1.&FstName1,t2.&FstName2 as FNX,
          t1.&Sex1,t1.&DOB1,t2.&DOB2 as DOBX,
              t2.&DOD2,calculated bonus +
               (calculated SSNcost*
                calculated SDXLN*
                calculated FN2*
                calculated BtwnDOB) as score
               from &DSN1 as t1,&DSN2 as t2
               where calculated score gt
                 0.4*0.8*0.6*0.7 * &c
                 and t1.&key1 = t2.&key2
                 and t1.&keyval1 = t2.&keyval2
               order by calculated score
                DESCENDING,SubmitID,SourceID
               ;
 quit;

%mend mtch;
```

The structure of the SQL program makes it relatively easy to adapt to other purposes and to port to other SQL implementations.

Grouping Links and Decisions by Score Range (BA)

In some cases we expect more than one event row in a target dataset to link to one and the same person row in the search dataset; one event row in the target dataset linked to more than one person row in the search dataset indicates at least one error. In lower score ranges, the number of cross-linked events should increase sharply. Clerical reviewers can verify small samples (≈300 links) of linked pairs drawn from different score ranges. Frequencies of reviewer decisions by scores make it possible to evaluate linkage performance within different ranges of scores.

Real database case study(5): *During 2000 a particularly difficult linkage task required linkage of personal information on each member of a study cohort of around one-hundred forty-five thousand persons to a database of some twenty million exposure measurements containing names, demographic information, and a supposedly unique identifying number (US SSN) for each person. Some in the cohort should not link to any exposure measurements, and some should link to more than one. Researchers expected about ninety-seven thousand persons in the cohort to link to at least one exposure measurement. Roughly ninety thousand cohort records linked on the primary person key, SSN, to at least one exposure measurement.*

Fuzzy linkage on a primary and on alternative keys linked the expected number of around ninety-seven thousand persons to at least one exposure measurement. About forty-five thousand of over two-hundred fifty thousand linked exposure measures required clerical reviews. A relatively large fraction of the ninety-seven thousand linked persons, 8.5%, linked to an exposure record on an alternative key, but not on the primary key.
Many linked on alternative keys had small errors in the primary key but had full or partial matches on names and demographic data. These almost certainly qualified as correct links. Around 2% or so of cases of records linked on identical primary keys, then failed to match on any alternative key or fragment of a key. Researchers reclassified these cases as linkage errors and dropped them from the set of linked records.

Conclusions

Fuzzy key linkage has an important role in data quality improvement of RDBMS's and other data repositories, and in linkage across databases. The computational burden of linkage of alternative keys means that such a task needs careful planning and good choices of resources. The SAS® System provides a rich variety of tools for conducting a linkage project and a basis for implementing new tools.

Acknowledgments

Paul Dorfman contributed many valuable ideas as well as programs and technical advice. Kellar Wilson and Lillie Stephenson tested variants of methods presented in the paper. Other Westat colleagues, especially Mike Rhoads and Ian Whitlock, and many contributors to SAS-L have for no fault of their own contributed to this effort.

References

Alvey, W. and B. Jamerson, eds. "Record Linkage Techniques – 1997", Proceedings of an International Workshop and Exposition. Washington, DC, 1997.

Arellano, M., Weber, G. "Issues in identification and linkage of patient records across an integrated delivery system", J. Healthcare Information Management, (3) Fall, 1998:43-52.

Bentley, J. "SAS Multi-Process Connect: What, When, Where, How, and Why". Proceedings of SESUG 2K, Charlotte, NC, 2000.

Beyer, K., J. Goldstein, R. Ramakrishnan and U. Shaft. "When Is `Nearest Neighbor' Meaningful?", Proceedings 7th International Conference on Database Theory (ICDT'99), pp.217-235, Jerusalem, Israel, 1999.

Busch, M., K. Watanabe, J. Smith, S. Hermansen, R. Thomson, "False-negative testing errors in routine viral marker screening of blood donors", TRANSFUSION 2000;40:585-589.

Doninger, C. "Multiprocessing with Version 8 of the SAS System", SAS Institute Inc, (2001) ftp.sas.com/techsup/download/technote/ts632.pdf

Dorfman, P. "Table lookup via Direct Addressing: Key-Indexing, Bitmapping, Hashing", Proceedings of SESUG 2K, Charlotte, NC, 2000.

Hermansen, S. 'Think Thin 2-D: "Reduced Structure" Database Architecture', Proceedings of SESUG 2K, Paper#1002, Charlotte, NC, 2000.

McHugh, J., S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data. SIGMOD Record, 26(3):54-66, September 1997.

Patridge, C. "The Fuzzy Feeling SAS Provides: Electronic Matching of Records without Common Keys", 1998 http://www.sas.com/ service/doc/periodicals/obs/obswww15/index.html

Pierce, E., "Modeling Database Error Rates", DATA QUALITY 3(1) September, 1997.

Rhoads, M., "Some Practical Ways to Use the New SAS Pattern-Matching Functions", Proceedings of the 22nd Annual SAS Users' Group International Coference: 72, San Diego, CA, 1997. `http://www2.sas.com/proceedings/sugi22/CODERS/PAPER72.PDF.`

Tsur, D., Ullman, J., Abiteboul, S., Clifton, C., Motwani, R., Nestorov, S., Rosenthal, A. "Query flocks: A generalization of association rule mining", Proceedings of the 1998 ACM SIGMOD Conference on Management of Data, 1-12, Seattle, WA, June, 1998.

Winkler, W. "The State of Record Linkage and Current Research Problems", Bureau of the Census, Suitland, MD, 2000.

Author Contact Information

Sigurd W. Hermansen
WESTAT, An Employee–Owned Research Corporation
1650 Research Blvd.
Rockville, MD 20850
USA
phone: 301.251.4268
e-mail: hermans1@westat.com

Point, Set, Match (Merge) – A Beginner's Lesson

Jennifer Hoff Lindquist, Institute for Clinical and Epidemiologic Research,
Veterans Affairs Medical Center, Durham, NC

The phrase "Point, Set and Match" is used in tennis when the final game winning point is scored. Those terms are also special SAS techniques that can make you a champion SAS programmer. Projects can collect data from a number of different sources. Combining the information into a cohesive data structure is essential for resource utilization. One of the most valuable resources is your time. By combining and collapsing data into a small number of datasets, information can be accessed and retrieved more quickly and already properly linked. Two techniques used with manipulating existing data sets are SET and MERGE.

SET

The SET statement is often used in two ways – copying and appending.

Set – Copy

To avoid corrupting a permanent SAS data set, copy of the data set is desirable. Suppose the permanent SAS dataset In.Source A consists of 5 observation with 4 variables. The syntax to create a copy of the data set is Set <dataset name>.

```
SAS Code:    Data White;
                Set In.SourceA;
             Run;
```

The contents of the White data set are an exact replicate of the orignal In.Source A data set.

White data set

ID	GRP	AGE	ELIG
1	A	30	Y
2	A	40	N
3	A	50	N
4	A	60	Y
5	A	70	

Set – Append

The SET statement can be used to append or stack data sets.

Let data set Yellow consist of 3 observations with 3 variables

Yellow Data Set

ID	GRP	ELIG
3	B	Yes
5	B	No
6	B	Yes

```
SAS code:  Data TwoSets;
              Set White Yellow;
           Run;
```

Any number of data sets could be listed. The white data set contributes 5 observations and the yellow data set tacks on 3 observations. All the variables in the two data sets are included. If any variables are only in one of the data sets, the variable is included in the concatenated dataset. The observations which originated from a dataset without a particular variable has the variable added to the observation with a missing value.

SAS Output:

ID	GRP	AGE	ELIG
1	A	30	Y
2	A	40	N
3	A	50	N

4	A	60	Y
5	A	70	
3	B	.	Yes
5	B	.	No
6	B	.	Yes

The observations are simply stacked starting with the data set listed first. SAS then continues tacking on the observations to the "bottom" of the list for each data set listed in the SET statement. This is especially useful when consolidating data sets with mutually exclusive records but the same variables.

MERGE

MERGE-GENERAL

Joining records "side by side" instead of stacking is another data consolidation technique. Many times your want to create a data set with one observation per patient/person. A merge statement then is more applicable. The remainder of the paper will be devoted to discussing the various types of merges – One to One Merge, Match Merge, One to Many Merge and Many to Many Merge.

MERGE-ONE TO ONE MERGE

The first type of merge is a one to one merge. The accidental or careless use of this merge can produce disastrous results. In all SAS merges a data set listed first (on the left) is overwritten by the corresponding data in the data set on the right. In a one to one merge the observations are conjoined by their relative positions-the first observation with the first observation, etc.

White Data Set

ID	GRP	AGE	ELIG
1	A	30	Y
2	A	40	N
3	A	50	N
4	A	60	Y
5	A	70	

Yellow Data Set

	ID	GRP	ELIG
↔	3	B	Yes
↔	5	B	No
↔	6	B	Yes

SAS Code

```
Data MergeSet;
    Merge White Yellow;
Run;
```

In this one to one merge, the values in the first three observations in the white data set are wiped out by the overlapping variables in the yellow data set even though they are NOT referring to the same individuals.

SAS Output

ID	GRP	AGE	ELIG
3	B	30	Yes
5	B	40	No
6	B	50	Yes
4	A	60	Y
5	A	70	

The resulting data set has "lost" patients with ids 1 and 2. The age for patient #3 appears to be 30 when it is actually the age for patient #1. Other errors include the ages for patients 5 and 6. Patient #5 has ages 30 years apart! Due to this potential to lose/corrupt data, the one to one merge is best avoided.

MERGE - MATCH MERGE

A refinement of the one to one merge is the match merge. Using the BY statement, variables are listed which SAS uses to match observations. Data sets must be sorted on the same variables as listed in the match merge BY statement. More than two data sets may be included in the merge statement. More than one variable can be listed on the BY statement. But only one BY statement is allowed for each Merge statement.

SAS Code:
```
Proc Sort data=white;
    By Id;
Proc sort data=yellow;
    By Id;
Run;

Data Mergeby;
    Merge white Yellow;
    By ID;
Run;
```

SAS Output

ID	GRP	AGE	ELIG
1	A	30	Y
2	A	40	N
3	B	50	Yes
4	A	60	Y
5	B	70	No
6	B		Yes

If two data sets have variables with the same name, the value of the variable in the data set named last will overwrite the value of the variable in the data set name previously. The ELIG for Patient #3 in the white date set was “N” but in the yellow data set ELIG was “Yes”. Since the order in the Merge statement was white then yellow the value in the yellow data set appears in the merged dataset. However, due to the overwrite property this conflict of eligibility status is lost.

The match merge and the one to one merge differ in syntax only in the use of the BY statement. It is (too) easy to inadvertently leave off the BY statement. Results are NOT the same! SAS has acknowledged that this can be a problem. In version 8, there is a system option called MERGENOBY. It has 3 settings – None, Warning, Error. The Warning setting will write a Warning message in the log whenever a merge is performed without a BY statement but will continue processing. The Error setting will write an Error message in the log and halt processing. The None setting – no message is written in the log. I strongly recommend using at least the Warning option.

MERGE -IN Option

A useful option with both the SET and MERGE statements is the IN statement. The syntax is data set name (IN= temporary variable). The temporary variable is assigned a one if the observation came from that data set. The temporary variable is receives a value of zero if the observation is not in that data set. The temporary variable exists only for the length of time it takes to process the observation. It is not accessible after the completion of the data step. If the information will be needed later, a regular variable can copy the value of the temporary variable;

SAS Code:
```
Data MergeSource;
    Merge White (IN=InWt) Yellow(IN=InYel);
    By Id;
        If InWt=1 and InYel=1;
            *Alternate if InWt=InYel;
        WtFileInd=InWt;
Run;
```

An intermediate internal processing snapshot shows the values of the temporary variables.

ID	GRP	AGE	ELIG	InWt	Inyel	ID	GRP	ELIG
1	A	30	Y	1	0			
2	A	40	N	1	0			
3	A	50	N	1	1	3	B	Yes
4	A	60	Y	1	0			
5	A	70		1	1	5	B	No
				0	1	6	B	Yes

Due to the subsetting if statement the observation must be in both the white and the yellow data sets to make the eligibility requirements for the MergeSource data set. The temporary variables InWt and InYel are not in the resulting data set. The problem remains with the second data set overwriting the first dataset.

Resulting Data set

ID	GRP	AGE	ELIG	WtFileInd
3	B	50	Yes	1
5	B	70	No	1

MERGE - RENAME Option

An option to avoid some of the overwrite problems is to rename the variables in the merge. The syntax is after the dataset name (rename=(old variable=new variable))

```
SAS Code        Data MergeRen;
                   Merge White(rename=(ELIG=WELIG)) Yellow(rename=(ELIG=YELIG));
                   By ID;
                Run;
```

The original data set supplies the value to the new variable as long as it was the dataset contributing the observation.

SAS Output

ID	GRP	AGE	WELIG	YELIG
1	A	30	Y	
2	A	40	N	
3	B	50	N	Yes
4	A	60	Y	
5	B	70		No
6	B			Yes

By using the rename option, it is possible to detect the inconsistency in patient #3 data.

MERGE - ONE TO MANY MERGE or MANY TO ONE MERGE

A third major category of merges is the One to Many and the closely related Many to One merges. The syntax is the same as the matched merge. However, it is important to know which data set has the "many" observations and which data set has the "one" observation. The order the data sets are listed in the MERGE statement makes a difference. The logistics of the merge is basically the same. The items in the right data set overwrites the data in the left data set.

```
SAS Code:       Data One2Many;
                   Merge white green;
                   By id;
                Run;
```

Visualizing the data sets side by side will help show what happens.

White Data Set

ID	GRP	AGE	ELIG
1	A	30	F
2	A	40	M
3	A	50	M
4	A	60	F
5	A	70	

Green Data Set

ID	GRP	TYPE
3	C	a
3	C	b
3	C	c
5	C	b
5	C	c

Results of a One to Many Merge

ID	GRP	AGE	ELIG	TYPE
1	A	30	Y	
2	A	40	N	
3	C	50	N	a
3	C	50		b
3	C	50		c

4	A	60	Y	
5	C	70		b
5	C	70		c

The values in variables AGE and ELIG are retained until the value of the BY GRP variable changes.

If the order is reverse, a Many to One merge results in a different data set.

SAS Code:
```
Data Many2One;
    Merge green white;
    By id;
Run;
```

Looking at the data sets side by side, recall the data on the right overwrites the data on the left.

Green Data Set

ID	GRP	TYPE
3	C	a
3	C	b
3	C	c
5	C	b
5	C	c

White Data Set

ID	GRP	AGE	ELIG
1	A	30	Y
2	A	40	N
3	A	50	N
4	A	60	Y
5	A	70	

The results of the Many to One Merge

ID	GRP	TYPE	AGE	ELIG
1	A		30	Y
2	A		40	N
3	A	a	50	N
3	C	b	.	
3	C	c	.	
4	A		60	Y
5	A	b	70	
5	C	c	.	

A Many to One merge is possible. However, the values of AGE and ELIG were not retained. The Many to One and the One to Many data sets are different. The differences are highlighted.

Results of a One to Many Merge

ID	GRP	AGE	ELIG	TYPE
1	A	30	Y	
2	A	40	N	
3	C	50	N	a
3	C	50		b
3	C	50		c
4	A	60	Y	
5	C	70		b
5	C	70		c

Be aware the simple change in the order the data sets are listed DOES make a difference.

It is import to know your data. Look at the proc contents before performing merges. Print several observations of the original data sets and the merged dataset. Check and make sure you are getting the results you expect.

MERGE - MANY to MANY MERGE - General

The last category is a Many to Many merge. This type of merge prompts regularly recurring questions on SAS-L, a mail serve list for SAS questions. The problem is the same basic syntax does not yield the desired results for a Many to Many situation!

Green Data Set

ID	GRP	Type
3	C	a
3	C	b
5	C	c
5	C	b
5	C	d

Red Data Set

ID	CAT
3	10
3	20
5	50
5	60

The usual DESIRED results are a Cartesian cross product with 10 observations.

ID	GRP	TYPE	CAT
3	C	a	10
3	C	b	10
3	C	c	10
3	C	a	20
3	C	b	20
3	C	c	20
5	C	b	50
5	C	c	50
5	C	b	60
5	C	c	60

However, the expected SAS code does NOT produce the above results.

SAS Code:

```
Data Many2ManyERROR;
        Merge red green;
        By Id;
Run;
```

Results

ID	GRP	TYPE	CAT
3	C	a	10
3	C	b	20
3	C	c	20
5	C	b	50
5	C	c	60

Possible Solutions include using a SQL procedure or manipulate the dataset with the POINT command.

MERGE – MANY to MANY USING SQL

The SQL procedure implements the Structured Query Language. Using proc SQL a Cartesian cross product data set can be produced.

SAS Code:

```
Proc SQL;
Create table manySQL as
        Select *
        From green, red
        Where green.id=red.id
        ;
Quit;
```

Explanation of SQL code
The phrase “Create table many SQL as” creates a data set named manySQL, storing the results of the query expression.

The code “Select *” includes all the variables all of the data sets listed in the next snippet of code. An alternative is to name the variables you want to keep in the data set.
The names of the source data sets are identified with “From green,red”.
The instructions “Where green.id=red.id” states the condition.

POINT in a MANY to MANY MERGE

Instead of using Proc SQL, it is possible to create the data set in a data step. This is accomplished through accessing observations in one data set sequentially and the observations in the other directly using the POINT= statement.

The vast majority of my work I process data sequentially. That is accessing the observations in the order in which they appear in the physical file. Usually every observation needs to be examined, and processed so sequential access is adequate. When working with small datasets sequential access is not a problem.

On occasion it is advantageous to access observations directly. You can go straight to a particular observation without having to handle all of the observations that come before it in the physical file. The POINT= option with the SET statement will communicate to jump to a particular observation. Suppose you had a large data set with 1 million observations named BigSet and you knew ELIG was missing for some observations in the last 100 observations. Instead of sorting or processing the 999,900 other observations you can go directly to the last 100 observations to identify those with missing ELIG values.

SAS Code:
```
Data  FindMissing;
        Do I = 999900 to 1000000;
          Set BigSet point=I;
          If ELIG='' then output;
        End;
      Stop;
Run;
```

When you use the POINT= option, a STOP statement must be included. If the STOP is inadvertently left off, a continuous or endless loop occurs. You are pointing to specific observations and SAS never will find/read the end of file indicator.

The Many to Many Merge data step solution using the Point option is given below. The Green data set is being processed sequentially. For each observation in the Green data set each observation in the Red data set is accessed directly in the Do Loop. The values read with the SET statement are automatically retained until another observation read from that data set. So each observation of the Red dataset is paired with the retained observation from the Green data set. If the tempId variable in the Green data set matches the Id variable in the Red data set then the observation is outputted to the Cross product data set.

```
Data CrossProduct (drop=tempID);
  Set green(rename=(id=tempId));
  NumInRedSet=4;
        Do i=1 to NumInRedSet;
              Set Red Point=i;
            If tempid=id then output;
        End;
Run;
```

Green Data Set			Red Dataset	
ID	GRP	Type	ID	CAT
3	C	a	3	10
3	C	b	3	20
5	C	c	5	50
5	C	b	5	60
5	C	d		

TempId	GRP	Type	i	ID	CAT	
3	C	a	1	3	10	Output

3	C	a	2	3	20	Output
3	C	a	3	5	50	
3	C	a	4	5	60	
3	C	b	1	3	10	Output
3	C	b	2	3	20	Output
3	C	b	3	5	50	
3	C	b	4	5	60	
3	C	c	1	3	10	Output
3	C	c	2	3	20	Output
3	C	c	3	5	50	
3	C	c	4	5	60	
5	C	b	1	3	10	
5	C	b	2	3	20	
5	C	b	3	5	50	Output
5	C	b	4	5	60	Output
5	C	c	1	3	10	
5	C	c	2	3	20	
5	C	c	3	5	50	Output
5	C	c	4	5	60	Output

Results

ID	GRP	TYPE	CAT
3	C	a	10
3	C	a	20
3	C	b	10
3	C	b	20
3	C	c	10
3	C	c	20
5	C	b	50
5	C	b	60
5	C	c	50
5	C	c	60

CONCLUSION

Trying to locate a particular data element in a large number of small, scattered data sets can be frustrating. Combining data sets with SET and MERGE statements can create data sets which are more comprehensive, cohesive and easier to utilize. The SET statement can be used to copy or append data sets. The MERGE statement used properly pulls together the common data elements for a unit of measurement. Usually a Matched Merge is preferred over a One to One Merge. Using the IN and RENAME options will refine newly created data sets. Many to Many merges are not necessarily intuitive. However, use the proc SQL or the Data Step point examples as templates will get you started on the right path. Combining and manipulating data sets does take a certain amount of skill. But just like tennis, with practice, POINT, SET, and MATCH (Merge), will become part of your winning game set.

Data Cleaning and Base SAS Functions

Caroline Bahler, Meridian Software Inc

Introduction

Functions are small programming subroutines and can be defined as the "work horses" of any data cleansing operation. "Dirty data", unfortunately, is the norm especially within demographic data where input errors are common. In addition, often there is the necessity of converting a variable within a data source from one data type into another (for example from a character date to SAS® date) in order to conform to pre-existing data. This paper is not an exhaustive study of all functions available within SAS® to cleanse data. Instead the objective of this paper is to discuss the most commonly used base functions within the following categories: data type conversion (input/put), character, date/time, and "geographic".

General Comments on Data Cleansing

Data cleaning, cleansing, or scrubbing all are synonymous terms for the same process – the removal of data values that are incorrect from a data source[5]. Dirty data refers to data that contains incorrect/ erroneous data values.

Data cleansing is an art not a science. Each set of data that needs to be cleaned has its own set of headaches and cleansing solutions. Therefore, the following functions allow the "cleanser" to tackle many types of problem in the basic cleansing line instead of being specific solutions for a defined situation.

Data cleansing requires the following information:

- Is there a pre-existing data source, either a database table or data set that the new data will be added to?
- Are there any business rules that need to be used during cleansing? Often one of the cleansing chores is to convert a field into another using a set of criteria.
- What are the cleansing problems in the new data? Before any cleansing effort can begin a inventory of all of the obvious flaws in the data needs to be compiled.

Finally, some general rules of data cleansing:

- The data is ALWAYS dirtier than you thought it was.
- New problems will always come to light once the obvious ones have been solved.
- Data cleansing is an on-going process that never stops.

Overview of Functions

Data Type Conversion Functions

Frequently a variable value needs to be converted from one format to another. For example, data within a new mailing list contains the zip code as a numeric value but your permanent customer data set has zip code as a character variable. The zip code can be converted from numeric to character using the PUT function:

```
data newlist;
 set newdata.maillist;

 zipcode = PUT(zip,z5.);
run;
```

In the previous example, a new character variable called zip code was created utilizing the PUT function. Conversely, if the zip code in the new mail list is character but it needs to be numeric then the INPUT function can be used[2]. For example,

```
data newlist;
 set newdata.maillist;

 zipcode = INPUT(zip,8.);
run;
```

In addition, to character / numeric conversions the PUT and INPUT functions can be used in the conversion of data/time values into character variables and vice versa.

Character Functions

Frequently it is necessary to change the form of a character variable or use only a portion of that value. For example, you might need to uppercase all letters within the variable value. In this case, a new variable does not need to be defined for the function to be used.

The following is a list of character functions that are extremely useful in data cleansing.

Function	Use
Compress	Removes specified characters from a variable. One use is to remove unnecessary spaces from a variable.
Index, indexc, indexw	These functions return the starting position for a character, character string, or word, and are extremely useful in determining where to start

Function	Use
	or stop when sub stringing a variable.
Left	Left justifies the variable value.
Length	Returns the number of characters with a character variable value.
Lowcase	Lower cases all letters within a variable value.
Right	Right justifies the variable value.
Scan	Returns a portion of the variable value as defined by a delimiter. For example, the delimiter could be a space, comma, semi-colon etc.
Substr	Returns a portion of the variable value based on a starting position and number of characters.
Translate	Replaces a specific character with characters that are specified.
Tranwrd	Replaces a portion of the character string (word) with another character string or word. For example, a delimiter was supposed to be a comma but data in some cases contains a colon. This function could be used to replace the comma with a colon.
Trim	Removes the trailing blanks from the right-hand side of a variable value.
Upcase	Upper cases all letters within a variable value.

If you need to use one of these functions on a numeric variable then it is preferable to first convert the numeric value into a character value (see previous section). By default, conversion from numeric to character will occur when using these functions within the DATA step with a warning placed at the end of the DATA step.

For example –

A new mailing list contains a date value that is a character and it needs to be converted into a SAS date value. An additional challenge is that the character value does not match any date informats.

Date character value format - Mon dd, yyyy

The solution to this conversion has two (2) steps –

1. Need to re-arrange the date character value so that the date is in the following format – ddmonyyyy, i.e. date9. informat.
2. Convert the new character value to a date value.

```
data newlist;
 set newdata.maillist;

/* Extract month, day and year */
/* from the date character var[a] */

   m = scan(date,1,' ');
   d = scan(year,2,' ');
   y = scan(year,2,',');
   dd = compress(d||m||y,' ,');

/* Convert mon, day, year into */
/* new date variable[b]          */
   newdate = input(dd,date9.);
 run;
```

a) In this case the SCAN function was used, but the SUBSTR function could also have been used to extract the month, day, and year from the original character date variable. The SCAN function was used because the data values contained a space or comma delimiter. Note that the comma was used to delimit the year and the text portion was the second and NOT the third. The reason for this is the text string has only two pieces, month and day, before the comma and year after the comma, when the comma is used as the only delimiter. The SUBSTR function would have been the only choice if a delimiter had not been available.

b) Conversion of the resulting mon, day and year variables into a new variable was accomplished by utilizing the COMPRESS function and INPUT functions. The COMPRESS function was used to remove any spaces present within the three (3) concatenated variables and to remove the comma within the day variable value. Note – by choosing to use the scan function for extracting the day value from the original date variable, the comma was left with the day value since there was no space between the day and comma. Finally, the use of the INPUT function creates a new variable with a SAS date value.

Parsing along –

In many data cleansing scenarios, a single data variable contains multiple pieces of data that need to be split into separate variables. If there is no delimiter between them, then the variable must be divided using the SUBSTR (substring) function.

The SUBSTR function requires a starting point and the number of characters to be kept in the new variable. In some cases however, the starting point may not be constant. In those cases then several

other character functions can be useful in determining where to start sub stringing.

Some examples -

- The last three characters of a variable are an id that requires a new variable. An additional hurdle is that the variable length is not constant.

  To extract the last 3 characters in this case the LENGTH function is used to define the starting position.

```
Data cleandata;
  Set dirtydata;
 a = substr(oldid,length(oldid)-3);
 put a;
run;
```

Oldid	New Id
A123B24	B24
AS1456B35	B35

- A character or a specific set of characters occur where the character string starts. Using the data from the last example, the last 3 characters can be extracted using INDEX to define the starting position.

```
data cleandata;
  set dirtydata;
 oldidx = upcase(oldid);
 a = substr(oldid,index(oldidx,'B'),3);
 put a;
run;
```

  In this case, the length of the character string to be extracted was specified. Note – case is important here so the following variation removes any case problems without affecting the case of the extracted string.

Date/Time Functions

Date/Time functions are a set of functions that return portions of date time, date, or time values or convert numeric values for month, day and year or hour, minute and seconds into SAS date or time values. These functions are especially useful for extracting the date and time from a date time value or converting separate month, day and year values into a SAS date value.

The following is a list of date/time functions that are extremely useful in data cleansing.

Function	Use
Month	Returns the month from a date value
Day	Returns the day from a date value
Year	Returns the year from a date value
Hour	Returns the hour from a time value
Minute	Returns the minute from a time value
Second	Returns the second from a time value
Datepart	Returns the date only from a date time value
Timepart	Returns the time only from a date time value
MDY	Returns a date value from the numeric values for month, day and year into a date value
HMS	Returns a time value from the numeric values for hour, minutes and seconds
Today()	Returns the current date value.
Date()	Returns the current date value.
Datetime()	Returns the current datetime value.

For example –

The new mailing list has in one case separate variables for month, day, and year for one date. The problem is that this data needs to be added to a pre-existing data set that contains this information as a single SAS date. If the data is numeric, then the use of the MDY function converts the separate variables into a single date value variable. However, if the data is character then the conversion to numeric should occur first and then the conversion to the date value.

The following codes shows how this two(2) part process can occur within one (1) statement.

```
data newlist;
  set newdata.maillist;

  newdate=mdy(input(mon,2.),
              input(day,2.),
              input(yr,4.);
run;
```

Note: as noted before if the character variables are not converted to numeric before the use in the MDY function, SAS will automatically convert these values to numeric and issue a warning at the end of the DATA step. However, good programming practices prefer the conversion of character variables into numeric before their use in a function like MDY.

"Geographic" Functions

"Geographic" functions consist of a set of state and zip code functions that can be used to verify state name spelling and state abbreviations (to a point). All useful when data cleansing, especially the conversion of the zip code to the abbreviation. This

is a function that can be used to verify that the abbreviation for the state is correct. However, this conversion has another use in identifying the zip codes that are potentially incorrect.

The following is a list of date/time functions that are extremely useful in data cleansing.

Function	Use
Stname	Returns state name in all upper case from state abbreviation.
Stnamel	Returns state name in mixed case from state abbreviation.
Zipname	Return state name in upper case from zip code.
Zipnamel	Returns state name in mixed case from zip code.
Zipstate	Returns state abbreviation from zip code.

For example –

```
data newlist;
  set newdata.maillist;

  if state ne zipstate(zip) then
     stateflag=1;
   else
     stateflag=0;
Run;
```

In the example, above the value returned by the ZIPSTATE function is compared to the variable containing the state abbreviation. If the two state abbreviations do not match, then a flag is set.

Putting it all together

Appendix 1 is an example of using all of the function types to cleanse a set of data that is going to be added to a pre-existing data table in a data warehouse. Table 1 lists the data in its "raw" form. All variables within the "raw" data set are character variables.

The following changes need to be made:

- Change moddate to datetime value
- Upper case all state abbreviations
- Ensure all phone numbers use only a dash as divider.
- Add identifier – the data needs a character variable that uniquely identifies each row. The identifier needs to start with 1000.
- Determine if state abbreviations match zip code determined abbreviations

Table 2 lists the data after cleansing and table 3 is a listing that identifies the case where the state abbreviations do not match. Note – the mismatch could be caused by either a data entry problem with the state abbreviation or a data entry problem with the zip code. In this case, our program has not identified the actual problem. Instead the program has identified only that there is a problem.

Conclusion

This paper was not an exhaustive study of all functions available within SAS® to cleanse data. Instead it discussed the most common base functions used to perform:

- data type conversions
- parse or change the justification or case of character variables
- parse and create date/time values
- determine state names from state abbreviations and zip codes

References

1. Functions and Call Routines, Base SAS Software. SAS On-line Documentation version 8. SAS Institute, Inc. Cary, NC.
2. Delwiche, Lora D. and Slaughter, Susan J. 1998. The Little SAS Book, Second Edition, SAS Institute, Inc. Cary NC. pp 204-205
3. Zip codes for basic example – **www.usps.com**
4. Howard, Neil. 1999. Introduction to SAS Functions. Proceeding of the Twenty-fourth Annual SAS User's Group International Conference. SAS Institute, Inc. Cary NC. pp 393-399.
5. Karp, Andrew. 1999. Working with SAS Date and Time Functions Proceeding of the Twenty-fourth Annual SAS User's Group International Conference. SAS Institute, Inc. Cary NC. pp 400-406.
6. Cody, Ron. 2000. Cody's Data Cleaning Techniques Using SAS Software. SAS Institute, Inc. Cary NC.

Trademarks

Contact Information

Caroline Bahler
Meridian Software, Inc.
12204 Old Creedmoor Road
Raleigh, NC 27613
(919) 518-1070
merccb@meridian-software.com

Appendix 1 - Basic Example

Cleansing Program.

```
data clean(drop = date time i moddate);
 set newdata.maillist;
 format datetime datetime21.;

 retain i 1000;

 /* identifier */
 i = i + 1;
 id = put(i,4.);

 /* conversion to datetime */
 date = compress(scan(moddate,2,' '),',')||
            scan(moddate,1,' ')||
            scan(moddate,3,' ');
 time = scan(moddate,4,' ');
 datetime = input(compress(date||":"||time),datetime21.);

 /* upper case state */
 st = upcase(st);

/* ensuring dash is divider in phone */
 phone = tranwrd(phone,'/','-');

 /* zip check on state abbrev */
 stabbrv = zipstate(zip);
 if stabbrv ne st then flag = "*";

run;
```

Table 1. Original Data

Id	First	Last	Address	City	State	zip	Area	Phone	Moddate
1001	Brenda	Jones	101 1st St	Omaha	NE	68101	123	147-2457	Jan 17, 2001 20:07:49
1002	Jim	Smith	5 Keyland Lane	Portland	ME	04103	213	125-4596	Jan 17, 2001 20:07:49
1003	John	Handford	3269 Graceland Ave	Memphis	MI	37501	111	235-9875	Jan 17, 2001 20:07:49
1004	Jane	Kew	5684 Jonesboro Rd	Blowing Rock	NC	28605	102	286-5468	Jan 17, 2001 20:07:49
1005	Mary	Roderick	201 Garland Dr	Atlanta	ga	30344	412	965/5692	Jan 17, 2001 20:07:49

Table 2. Cleaned Data

Id	First	Last	Address	City	State	zip	Area	Phone	Moddate
1001	Brenda	Jones	101 1st St	Omaha	NE	68101	123	147-2457	17JAN2001:20:07:49
1002	Jim	Smith	5 Keyland Lane	Portland	ME	04103	213	125-4596	17JAN2001:20:07:49
1003	John	Handford	3269 Graceland Ave	Memphis	MI	37501	111	235-9875	17JAN2001:20:07:49
1004	Jane	Kew	5684 Jonesboro Rd	Blowing Rock	NC	28605	102	286-5468	17JAN2001:20:07:49
1005	Mary	Roderick	201 Garland Dr	Atlanta	GA	30344	412	965-5692	17JAN2001:20:07:49

Table 3. State abbreviation check

Id	Orig State	State	Flag
1001	NE	NE	
1002	ME	ME	
1003	MI	TN	*
1004	NC	NC	
1005	**ga**	**GA**	

PROC REPORT: How To Get Started

Malachy J. Foley

University of North Carolina at Chapel Hill, NC

ABSTRACT

PROC REPORT started as a soupped-up version of PROC PRINT. Now this unique product combines features from PROC PRINT, SORT, FREQ, MEANS, and TABULATE. Because of its special blend of possibilities, PROC REPORT is often the easiest way to do an elegant data listing or a report with descriptive statistics.

This tutorial shows how to use the batch versions of PROC REPORT. Via examples, it thoroughly explores the use of PROC REPORT in creating specialized data listings.

INTRODUCTION

Many people shy away from PROC REPORT because of it's mysterious defaults. Or put another way, PROC REPORT acts differently than other SAS® PROC's. However, once you see how PROC REPORT works, you may never want to go back to other PROC's.

PROC REPORT does the yeoman's work of PROC PRINT, SORT, FREQ, MEANS, and TABULATE, and PUT-Statement Formatting (DATA _NULL_) all in one procedure.

The purpose of this paper is to examine, in detail, the how to produce listings with batch-mode PROC REPORT. That is, this paper will look at the PRINT, SORT and DATA _NULL_ aspects of PROC REPORT. The FREQ, MEANS and TABULATE aspects will be examined in a subsequent article. All of the examples in this paper are designed to function using SAS version 6.12 or higher.

This tutorial is intended for SAS programmers with knowledge of the SAS dataset structure, and exposure to the SAS PRINT and CONTENTS procedures. After completing this article, the reader should be able to do data listing using PROC REPORT and have the foundation for going on to learn how to do descriptive statistics with the procedure.

SAMPLE INPUT DATA SET

The following two Exhibits are a PROC PRINT and a partial PROC CONTENTS of an example data set. This data set is going to be used as input to all the examples of PROC REPORT presented in this paper.

Exhibit 1. Listing of Input Data Set

```
PROC PRINT DATA=ORG NOOBS UNIFORM;
RUN;
```

OUTPUT... file=ORG

ID	SITE	DUM	NAME	SEX	AGE
A01	RU	1	SUE	F	58
A02	LA	1	X	M	58
A04	RU	1	TOM	M	21
A07	LA	1	LEE	F	47
A08	LA	1	KAY	F	29
A10	RU	1		M	36

Exhibit 2. Partial CONTENTS of Data Set

Variables Ordered by Position

#	Variable	Type	Len	Pos	Label
1	ID	Char	4	0	
2	SITE	Char	2	4	
3	DUM	Num	8	6	
4	NAME	Char	3	14	
5	SEX	Char	1	17	
6	AGE	Num	8	18	Age in years

FEATURES & DEFAULTS OF PROC REPORT

Observe Exhibit 3. It shows a very simple PROC REPORT. This example illustrates the procedure's defaults.

Exhibit 3. REPORT's defaults in Batch Mode

```
PROC REPORT DATA=ORG NOWINDOWS;
RUN;
```

ID	SITE	DUM	NAME	SEX	Age in years
A01	RU	1	SUE	F	58
A02	LA	1	X	M	58
A04	RU	1	TOM	M	21
A07	LA	1	LEE	F	47
A08	LA	1	KAY	F	29
A10	RU	1		M	36

The only option used in Exhibit 3 is the NOWINDOWS option. This option is required in batch mode and whenever you want your output sent to a file rather than a window. Since this paper is dedicated to batch mode processing, all examples will use the NOWINDOWS option. Everything else in the Exhibit is a default.

As you can see from Exhibit 3, PROC REPORT's defaults are different than PROC PRINT's defaults. In fact, some of REPORT's defaults are different than the defaults used in the rest of SAS. Below is a list of REPORT's defaults. "Same" and "Dif" indicate if the default is the same or different than PROC PRINT.

- Recs/rows ordered as they appear in data set- Same
- Variables/Columns in position order (Proc Contents) -Same
- UNIFORM is default. -Dif
- No Record Numbers (NOOBS) is default - Dif
- Labels (not var names) used as headers - Dif
- REPORT needs NOWINDOWS option. - Dif
- Default spacing not as nice as Proc PRINT. – Dif

VARIABLE NAMES AS COLUMN HEADINGS

As a default, PROC REPORT uses the variable labels as column headings. This is different than PROC PRINT which uses variable names as column headings. If you want to create a report that uses variable names as column headings, the NOLABELS system option will do the trick. This is illustrated in the next exhibit.

Exhibit 4. Variable Names as Col Headings

```
     OPTIONS NOLABEL;
     PROC REPORT DATA=ORG NOWINDOWS;
     RUN;
```

ID	SI TE	DUM	NAM E	S E X	AGE
A01	RU	1	SUE	F	58
A02	LA	1	X	M	58
A04	RU	1	TOM	M	21
A07	LA	1	LEE	F	47
A08	LA	1	KAY	F	29
A10	RU	1		M	36

MYSTERIOUS RESULTS

One of the things people don't like about PROC REPORT is that sometimes it gives some unexpected results.

Look at Exhibit 5. This is a simple PROC REPORT which is exactly the same as Exhibit 3, except that the input file is subsetted to two variables. But something strange happens. Rather than listing the values in the input data set from Exhibit 1, it sums the values! This is because the two variables (DUM and AGE) are numeric. When all the variables in the input file are numeric, PROC REPORT does a sum as a default.

Exhibit 5. Unexpected Results

```
PROC REPORT
      NOWINOWS
      DATA=ORG (KEEP=DUM AGE)
    ;
RUN;
```

OUTPUT

DUM	Age in years
6	249

THE DEFINE STATEMENT

To avoid having the sum of numeric variables, one or more of the input variables must be defined as DISPLAY. The DISPLAY option forces each observation to Print. Furthermore, it is good programming practice to always define each variable that is in the output report. The following is an example of how to use the DEFINE statement.

This example also shows that the NOWINDOW option can be abbreviated as NOWD.

Exhibit 6. The DEFINE statement and NOWD

```
PROC REPORT DATA=ORG (Keep=DUM AGE) NOWD ;
       DEFINE  DUM  /DISPLAY;
       DEFINE  AGE  /DISPLAY;
RUN;
```

DUM	Age in years
1	58
1	58
1	21
1	47
1	29
1	36

ORDERING/SUBSETTING COLUMNS (VARS)

In the previous example, the KEEP data set option was

used to subset the variables to be displayed in the output. Another way to control which variables are outputted, is to use the COLUMN statement. This statement is similar to the VAR statement in PROC PRINT. It determines which variables are going to be in the report and in what order the variables are displayed.

Exhibit 7. Ordering/Subsetting Cols/Vars

```
PROC REPORT   NOWD   DATA=ORG  ;
    COLUMN  AGE  DUM;
    DEFINE  AGE  /DISPLAY;
    DEFINE  DUM  /DISPLAY;
RUN;
```

Age in Years	DUM
58	1
58	1
21	1
47	1
29	1
36	1

SYNTAX SO FAR

As seen, in the batch mode you must always use the NOWINDOWS=NOWD option in PROC REPORT. Also, iIt is good programming practice to always use the COLUMN and DEFINE statements. Furthermore, the NOLABEL option and the DISPLAY manipulation type have been introduced. As such, the syntax of PROC REPORT thus far is.

Exhibit 8. PROC REPORT's Syntax So Far

```
OPTIONS NOLABEL;
PROC REPORT
         NOWD
         DATA=file-name
           (OBS= WHERE=() DROP= KEEP=)
        ;
    COLUMN list-of-variables;
    DEFINE  var-name /DISPLAY;
    DEFINE  var-name /DISPLAY;
RUN;
```

WHY PEOPLE DO/DO NOT USE PROC REPORT

By now, it is becoming apparent why people avoid PROC REPORT. As seen in the previous Exhibit, PROC REPORT almost always requires more statements than PRINT. Also, PROC REPORT's defaults are to different than PROC PRINT's and generally unfamiliar. Sometime, like in Exhibit 5, PROC REPORT seems to act strangely. Moreover, REPORT defaults do not space fields nicely.

What may not be so apparent is why people DO use PROC REPORT. The short answer is that PROC REPORT is much more flexible than PRINT and allows you to create more professional looking reports. The significant capabilities of REPORT will become noticeable as this paper proceeds.

To appreciate the flexibility of PROC REPORT, horizontal spacing is examined in the next section.

HORIZONTAL SPACING AND MORE

The following Exhibit shows a PROC-PRINT type of report of the data using, but PROC REPORT. Notice that the output is not as nice as PROC PRINT's defaults provide (see Exhibit 1). In the output of the next Exhibit, the column titles are crazy and spacing between the columns is not uniform.

Ehibit 9. A Proc-PRINT Type of Listing

```
PROC REPORT NOWD
            DATA=ORG (WHERE=(ID<"A08")) ;
    COL ID SITE NAME AGE ;
    DEFINE ID/DISPLAY;
    DEFINE SITE/DISPLAY;
    DEFINE NAME/DISPLAY;
    DEFINE AGE/DISPLAY;
RUN;
```

ID	SITE	NAME	Age in years
A01	RU	SUE	58
A02	LA	X	58
A04	RU	TOM	21
A07	LA	LEE	47

To figure out how to make the PROC REPORT listing look nicer, one must understand how REPORT creates horizontal spacing. Actually, horizontal spacing is a combination of spacing between fields, width of the fields, format of the fields and justification of the data within the fields. All of this can be summarized as follows.

- Default Spacing between fields = 2 blanks
- Default Justification (=Alignment)
  - RIGHT for Numeric Fields
  - LEFT for Character Fields

- If no format specified, then Proc REPORT uses
  - Best9. for Numeric Fields
  - $w. for Character Fields (w=width)

- Default Width= Format width

If you examine Exhibit 9, it becomes apparent that the crazy titles and spacing is caused by the default widths. It is quite easy to upgrade the spacing with by adding a width specification to the define statements as follows.

Exhibit 10.Cleaning Up Exhibit 9's Listing

```
PROC REPORT HEADLINE HEADSKIP NOWD
            DATA=ORG(WHERE=(ID<"A07"));
    COL ID SITE NAME AGE ;
    DEFINE ID   /DISPLAY;
    DEFINE SITE /DISPLAY WIDTH=4 RIGHT;
    DEFINE NAME /DISPLAY WIDTH=4 RIGHT;
    DEFINE AGE  /DISPLAY "AGE" WIDTH=3;
RUN;
```

ID	SITE	NAME	AGE
A01	RU	SUE	58
A02	LA	X	58
A04	RU	TOM	21

You will see in Exhibit 10 that aside from adding the WIDTH option on the DEFINE statement several other things were done.

To start with, the HEADLINE and HEADSKIP options were add to the PROC REPORT statement. HEADLINE creates the line below the column titles and HEADSKIP creates the blank line below the headline.

Also, the "Age" column title was added to the DEFINE statement. This title overrides the label in the descriptor part of the input data set. Column labels or titles can come from a variety of sources. How PROC REPORT chooses which label to use is described in the next exhibit.

Exhibit 11. Label Hierarchy

REPORT uses the 1st label it finds in this list.

- *col-heading"* (DEFINE option)
- OPTIONS NOLABEL; (implies Var Names)
- LABEL Statement in Proc REPORT
- LABEL in the Data Descriptor
- Variable Names

You may have noticed in Exhibit 10 that several options were used in the DEFINE statement. Actually the DEFINE statement is one of the places were REPORT derives a lot of its power. The DEFINE statement lets you do just about anything you want to the column of data it is defining. The next Exhibit outlines some of the options available in the DEFINE statement.

Exhibit 12. Some DEFINE Statement Options

- DISPLAY
- Col-Heading in quotes
- SPACING= (Overrides all Spacings)
- FORMAT= (Overrides all other Formats)
- WIDTH= (Default=Format Width)
- Justification Specifications
  - RIGHT
  - LEFT
  - CENTER

Notice that the DEFINE statement gives you almost total horizontal control. Also, notice that the DEFINE statement options override almost all other options in your program.

As such, the author suggest that you use the DEFINE statement to define all of your horizontal spacing rather than use other methods. There are several reasons for the suggestion. First, you don't have to memorize the various hierarchies, like the label hierarchy of Exhibit 11. You know that whatever is in the DEFINE statement is what is going to happen. Second, you don't have to keep track of what has already been defined elsewhere in your program. For example, you don't have to remember if you have labels defined in the descriptor part of your program or if you have a subsequent label statement, etc.

While you don't have to remember hierarchy rules to use REPORT, it probably is a good idea to remember the justification rules. These rules follow.

Exhibit 13. Justification Rules
(Right, Left, Center)

Justification applies to
- col-headings
- data values.

Default Justification (=Alignment)
- RIGHT for Numeric Fields
- LEFT for Character Fields

Numerical values:
- always remain right justified within FORMATs

- FORMATs are justified within the WIDTH.

For col-headings & character values:

- values are justified within the WIDTH. (without regard for the $w. FORMAT).
- leading blanks are retained.
- trailing blanks are eliminated.

Just how these Justification rules work are demonstrated in the next two Exhibits. Exhibit 14 shows the case of right justification, and Exhibit 15 shows the case of left justification.

Exhibit 14. RIGHT Justification

```
DEFINE AGE / 'AGE∆' SPACING=3 WIDTH=6
             FORMAT=4. RIGHT;
-----------------------------------------
COL=  123456789        s=Spacing location
TYPE= ssswwwwww        w=Width location
TYPE=      ffff        f=Format location
           AGE         ∆=Blank space
          ∆∆58
-----------------------------------------
```

Exhibit 15. LEFT Justification

```
DEFINE AGE / '∆AGE'" SPACING=3 WIDTH=6
             FORMAT=4. LEFT;
-----------------------------------------
COL=  123456789        s=Spacing location
TYPE= ssswwwwww        w=Width location
TYPE=    ffff          f=Format location
         ∆AGE          ∆=Blank space
         ∆∆58
-----------------------------------------
```

You will note from the previous two examples that REPORT gives you almost complete horizontal control of where you are putting your data. One thing it does not give you is trailing blanks.

FORCING A TRAILING BLANK

Notice the last line of Exhibit 13. It says that for column headings and character data trailing blanks are always eliminated by REPORT. Sometimes you want trailing blanks so that your data is justified correctly. One way of creating trailing blanks is given in the next example.

This trick works on most ASCII-based computers and PC's. The example shown in Exhibit 16 is identical to Exhibit 14 except that a &blk is placed behind the AGE column label in the DEFINE statement. The &blk character is created in the DATA _NULL_ step at the beginning of the Exhibit.

You do not need to understand how the CALL SYMPUT works to use this trick. Merely put the DATA _NULL_ somewhere at the beginning of your program and then use the &blk character whenever you need it.

One note of CAUTION… you must use double quotes (rather than single quotes) around the &blk for this trick to function.

Exhibit 16. Forcing Trailing Blanks

```
-----------------------------------------
DATA _NULL_;
   CALL SYMPUT('blk','FF'X);
RUN;

DEFINE AGE / " AGE&blk" SPACING=3 WIDTH=6
             FORMAT=4. RIGHT;
-----------------------------------------

COL=  123456789        s=Spacing location
TYPE= ssswwwwww        w=Width location
TYPE=      ffff        f=Format location
           AGE∆
           ∆∆58
-----------------------------------------
```

THE FLOW OPTION

Exhibits 14 to 16 discussed how to place your data almost anywhere you want to within a line. The next example reveals how to make a very long character value, like a note or a comment, to span several lines of output. This is yet another feature of PROC REPORT which makes it so powerful. The feature is called FLOW. Sometimes people will use REPORT rather than PRINT just because of the FLOW option. Observe how the FLOW option in the DEFINE statement for the note variable works.

Exhibit 17. The Flow Option

```
-----------------------------------------
PROC REPORT Headline NOWD DATA=ORG;
    COL ID NAME AGE NOTE;
    DEFINE NAME /DISPLAY WIDTH=4 RIGHT;
    DEFINE AGE  /DISPLAY "AGE" WIDTH=3;
    DEFINE NOTE /DISPLAY WIDTH=13 FLOW;
-----------------------------------------
```

```
ID    NAME  AGE  NOTE
A01    SUE   58  This is an ex
                 of FLOW.
A02      X   58  No flow here.
A04    TOM   21  Adverse
                 Event.
```

COLUMN HEADINGS

The previous sections demonstrate how REPORT gives you almost complete control of how to place your data horizontally. This section shows how REPORT lets you to play with the column heading.

The following exhibit shows how the dash is automatically expanded around a column heading (NOTE) and how a slash may be used to split column labels onto two lines.

Exhibit 18. Expanding dash.
Splitting slash.

```
PROC REPORT Headline NOWD DATA=ORG;
  COL ID NAME AGE NOTE;
  DEFINE NAME /DISPLAY WIDTH=7 RIGHT
            'First/Name"';
  DEFINE AGE  /DISPLAY "AGE" WIDTH=3;
  DEFINE NOTE /DISPLAY WIDTH=13 FLOW
           '- Note- '; RUN;
```

```
           First
ID       Name  AGE  ---- NOTE ---
A01       SUE   58  This is an ex
                    of FLOW.
```

The next exhibit shows how the STAR is automatically expanded around a column heading (NOTE) and how a SPLIT= option works. The SPLIT option in PROC REPORT functions much as it does in PROC PRINT.

Exhibit 19. Expanding star.
Split= option.

```
PROC REPORT Headline NOWD DATA=ORG
       SPLIT='*';
    COL ID NAME AGE NOTE;
    DEFINE NAME /DISPLAY WIDTH=7 RIGHT
             'First*Name'
    DEFINE AGE  /DISPLAY "AGE" WIDTH=3;
    DEFINE NOTE /DISPLAY WIDTH=13 FLOW
            '* Note *'; RUN;
```

```
           First
ID       Name  AGE  **** NOTE ***
A01       SUE   58  This is an ex
                    of FLOW.
```

Exhibit 20 illustrates how to create a column heading that spans several columns. Observe how the parenthesis indicate which columns the special title is to span. Specifically, the parentheses embrace the ID and NAME variables.

Exhibit 20. Special Column Heading

```
PROC REPORT Headline NOWD DATA=ORG;
    COL ('-ID INFO-' ID NAME) AGE NOTE;
    DEFINE ID   /DISPLAY 'ID#';
    DEFINE NAME /DISPLAY WIDTH=6 RIGHT;
    DEFINE AGE  /DISPLAY  WIDTH=3;
    DEFINE NOTE /DISPLAY WIDTH=13 FLOW
            '* Note *';    RUN;
```

```
     --ID INFO---
     ID#      NAME  AGE  **** NOTE ***
     A01       SUE   58  This is an ex
                         of FLOW.
```

The next exhibit details how to add a line above the column headings. This line could be called an "overline".

Note how the underline symbol in quotes is automatically expanded out to cover all the column headings to form the overline.

Here, like in the previous example, the parentheses indicate which columns are to be spanned with the special heading. The difference here is that there are two sets of parentheses. The outermost set is for the overline (to cover all the columns). The innermost set is to cover or span only the variable ID and NAME.

Exhibit 21 also demonstrates the automatic expansion of the greater-than and less-than symbols around the Note column title.

Exhibit 21. Overlining the column headings

```
PROC REPORT Headline NOWD DATA=ORG;
   COL ('_ _'('-ID INFO-' ID NAME) Age
                                 Note);
    DEFINE ID / 'ID#';
    DEFINE NAME /DISPLAY WIDTH=6 RIGHT;
    DEFINE AGE  /DISPLAY  WIDTH=3;
    DEFINE NOTE /DISPLAY WIDTH=13 FLOW
            '< Note >';    RUN;
```

```
--ID INFO---
ID#      NAME  AGE  <<<< NOTE >>>
A01       SUE   58  This is an ex
                    of FLOW.
```

In this section, many different column heading or labeling techniques have been illustrated. These techniques are summarized in the following table.

Exhibit 22.Summary of Column Headings

- Controlling the Breaks (stacking text)
  - 'xxx' 'yyy' (multiple pairs of quotes)
  - 'xxx/yyy' (slash is default split char.)
  - 'xxx*yyy' (with SPLIT='*' Report Option)
- Underlining
  HEADLINE Proc Report Statement Option
- Overlining (with text or characters)
  COL ('text' var var) var var;
- Extending Headers to Column Width
- 'char header-text char'
  (where char pair is ** == ++
  -- _ _ .. < > > <)
- Justification (LEFT, CENTER, RIGHT)

WHY PEOPLE DO USE PROC REPORT

Earlier in the article, several reasons shy people might not use PROC REPORT were given. Now that some of the features of RPEORT have been explored, it is only fair to state some of the reasons why people DO use REPORT. Basically all the reasons are REPORT features. With REPORT you have:

- Complete Horizontal Control
- Complete Col-heading Control
- Justification Control
- The FLOW option
- More Professional Looking Reports

And this is just the beginning of the list of features. The list goes on and on. In the next section, another feature of REPORT is examined. Namely, how you can do sorts within PROC REPORT.

ORDERING LINES IN OUTPUT - SORT

Until now, only the DISPLAY manipulation option of the DEFINE Statement has been used. There are data manipulation options available. One of them is the ORDER option. This option sort the rows before the output is printed. Here is an example.

Exhibit 23. The ORDER Option.

```
  Proc REPORT nowd headline headskip
              Data=ORG;
    COL SITE NAME AGE ;
    DEFINE SITE /ORDER WIDTH=4;
    DEFINE NAME /DISPLAY WIDTH=4;
    DEFINE AGE  /DISPLAY "AGE" WIDTH=3;
```

```
          SITE  NAME  AGE

           LA     X     58
                 LEE    47
                 KAY    29
           RU    SUE    58
                 TOM    21
                        36
```

Notice how in Exhibit 23 the records from the input data set described in Exhibit 1 are now sorted according to SITE. This was done with the ORDER option.

Also notice that in Exhibit 23 the SITE value is not repeated on each line, but rather given only once on its first occurrence. This overhang feature is automatic when you use the ORDER option.

The next exhibit is the same as the previous exhibit, except that now two variables are declared as ORDER type variables. Thus, you will get the output sorted on two variables. What's more, you should get two overhanging variables. However, in this example, each value of the second variable is distinct, so the overhanging effect is not evident.

Exhibit 24. Sort on two variables

```
  Proc REPORT nowd headline headskip
              Data=ORG;
    COL SITE NAME AGE ;
    DEFINE SITE /ORDER WIDTH=4;
    DEFINE NAME /ORDER WIDTH=4;
    DEFINE AGE  /DISPLAY "AGE" WIDTH=3;
```

```
SITE   NAME   AGE

 LA      X     58
        KAY    29
        LEE    47
 RU     SUE    58
        TOM    21
----------------------------------------
```

You might see that something is wrong with the output in the preceding exhibit. Namely, the record or row for the person with a missing Name is not printed. This is the way REPORT handles missing values on ORDER type variables. It does not print them. To get around this feature, you must add the MISSING option. The following exhibit is the same as Exhibit 24, except it includes the MISSING option .

```
Exhibit 25. MISSING Option
----------------------------------------
  Proc REPORT nowd headline headskip
              Data=ORG MISSING;
    COL SITE NAME AGE ;
    DEFINE SITE /ORDER WIDTH=4;
    DEFINE NAME /ORDER WIDTH=4;
    DEFINE AGE  /DISPLAY "AGE" WIDTH=3;
----------------------------------------
          SITE   NAME   AGE

           LA      X     58
                  KAY    29
                  LEE    47
           RU            36
                  SUE    58
                  TOM    21
----------------------------------------
```

TYPES OF STATEMENTS

REPORT like other PROC's allows you do use several general SAS statements, In addition to all the PROC REPORT statements like COL and DEFINE. Here is a list of which general statements are supported by RPEORT.

```
Exhibit 26. General SAS Statements.
----------------------------------------
Supported by PROC REPORT.
  • WHERE
  • LABEL - not recommended
  • FORMAT- not recommended
  • TITLE
  • FOOTNOTE
  • BY

NOT supported by PROC REPORT
  • KEEP
  • DROP
----------------------------------------
```

This article assumes the reader is familiar with these general SAS statements and knows how to use them.

FORMAT HIERARCHY

Exhibit 26 says that the LABEL and FORMAT statements are not recommended. This again is the suggestion to set your label and format values in the DEFINE statement for each variable. This suggestion is made so that you do not have to remember what the different hierarchies are nor remember what has been previously defined in your program. However, if you do wish to use something other than the DEFINE statement to specify your labels and formats, you should be aware of the hierarchies. The label hierarchy was presented in Exhibit 11 and the format hierarchy follows.

```
Exhibit 26. Format Hierarchy
----------------------------------------
REPORT accepts the 1st Format from the
following list that fits in WIDTH.
(Thus, WIDTH can affect FORMAT/Values.)

  • FORMAT=  (DEFINE option)
  • FORMAT Statement in Proc REPORT
  • FORMAT in the Data Descriptor
  • Default Formats as follows
  • Best9.  for Numeric Fields
  • $w. for Character Fields (w=width)
  • If Value does not fit in Format
  • Numeric Fields are filled with *
  • Character Fields are truncated
----------------------------------------
```

Notice that WIDTH's do affect the FORMAT and values, and that WIDTH size defaults to the FORMAT size. As such, the author makes two suggestions in choosing FORMAT's and WIDTH's for REPORT

- Choose FORMATs with more columns than your maximum expected value.
- Choose WIDTHs greater than or equal to the FORMATs.

ORDER= OPTION

Another programming suggestion is to always use the ORDER= option when you use the ORDER option in a DEFINE statement. Exhibit 26 is an example of how to use this the ORDER= option. The reason for the suggestion is that in every other PROC the order is different than in PROC REPORT. So, it is easy to expect the wrong sort. Exhibit 27 explains the different ORDER= Options.

Exhibit 26. Order= Option

```
  Proc REPORT nowd missing headline
          headskip Data=ORG;
    COL  SITE NAME AGE ;
    WHERE AGE>30. ;
    DEFINE SITE /ORDER WIDTH=14
          FORMAT=$SITE. ORDER=INTERNAL;
    DEFINE NAME /ORDER WIDTH=4;
    DEFINE AGE  /DISPLAY "AGE" WIDTH=3;
```

SITE	NAME	AGE
LOS ANGELES	X	58
	LEE	47
DURHAM-RALEIGH		36
	SUE	58

Exhibit 27. ORDER= Option Values

ORDER= (*one of the following*)
DATA or FORMATTED or FREQ or INTERNAL

Meaning of Options

- DATA - Order recs. as in the data set.
- FORMATTED - Sort values after formatting.
- INTERNAL - Sort values before formatting them.
- FREQ - Sort values by frequency of occurrence in the data set.

Defaults

- DATA when manipulation-type is DISPLAY.
- FORMATTED when manip.-type is ORDER.
- (in every other PROC, default is INTERNAL.)

THE BREAK STATEMENT

Finally, the BREAK statement is introduced in the next exhibit. BREAKS can be use in a variety of ways. In this example, it is used to put a blank line (a SKIP) after each different SITE.

Exhibit 28. BREAK Statement

```
Proc REPORT nowd missing headline DATA=ORG
    COL  SITE NAME AGE ;
    WHERE AGE>30. ;
    DEFINE SITE /ORDER WIDTH=4;
    DEFINE NAME /ORDER WIDTH=4;
    DEFINE AGE  /DISPLAY "AGE" WIDTH=3;
    BREAK AFTER SITE /SKIP;
RUN;
```

SITE	NAME	AGE
LA	X	58
	LEE	47
RU		36
	SUE	58

SYNTAX FOR LISTINGS

This article is an introduction to how to use PROC REPORT to create listings. Exhibit 29 shows all of the syntax that has been covered in this article and serves as an example of what can be done.

Exhibit 29. Syntax for Listings

```
OPTIONS NOLABEL LS= PS=;
PROC REPORT   NOWD  MISSING SPLIT='char'
      HEADLINE HEADSKIP SPACING=
      DATA=file-nam
        (OBS= Where=() Drop= Keep=);
  WHERE expression ;
  TITLE 'your-message' ;
  FOOTNOTE 'your-message';
  COLUMN ('Δheader&blk' list-of-vars)
            more-vars;
  DEFINE  var-name /<DISPLAY or ORDER>
          'Δcol-head&blk"
           SPACING=  WIDTH=  FORMAT=  FLOW
           RIGHT  LEFT  CENTER
           ORDER= DESCENDING;
  BREAK AFTER order-var /SKIP;
RUN;
```

PROGRAMMING RECOMMENDATIONS

Throughout this paper, the author has made several suggestions on how one might program a PROC REPORT. The following bullets summarize these

suggestions/tips.

- Always use NOWD
- Use MISSING most of the time.
- Use DEFINE for every variable.
- Use a Manipulation type (Display/Order) for every DEFINE.
- Use ORDER= with ORDER manipulation type.
- Use DEFINE for horizontal control (spacing, width, format)
- Don't use FORMAT and LABEL statements.
- Indent and align code.
- Always use RUN statement.

CONCLUSION

They say "there is no such thing as a free lunch". So it is with creating listings of data sets. When you need a quick listing, PROC PRINT is your best choice. However, when you need a more formal listing, PROC REPORT is probably your best choice. REPORT provides listing features that are difficult or impossible to obtain with PROC PRINT. For example,

- Fancy headings.
- FLOW of character value onto multiple lines.
- Total horizontal control.
- Break lines

REPORT almost gives you all of the features of using PUT Statement Formatting (DATA _NULL_), and without as much work.

Nonetheless, you don't get something for nothing. REPRORT, unlike PRINT, pretty much requires you use a define statement for every variable you output. On the other hand, this is a small price to pay for near-PUT-statement capabilities.

Furthermore, PROC REPORT goes way beyond great looking listings. It offers the ability to create descriptive statistics that are usually found in the FREQ, MEANS, and TABULATE procedures. Once you are familiar with all the listing features of REPORT, it is fairly easy to go and create great-looking reports that include descriptive statistics. How to use PROC RPEORT to create statistical reports is the subject of a future paper.

REFERENCE

SAS Technical Report P-258, *Using the REPORT Procecure in a Nonwindowing Environment*, Release 6.07, Cary, NC: SAS Institute Inc., 1993.

TRADEMARKS

AUTHOR CONTACT

The author welcomes comments, questions, corrections and suggestions.

Malachy J. Foley
2502 Foxwood Dr.
Chapel Hill, NC 27514

Email: FOLEY@unc.edu

DIRECT ADDRESSING TECHNIQUES OF TABLE LOOK-UP

PART 1. KEY-INDEXING - BITMAPPING
PART 2. HASHING

Paul M. Dorfman,
CitiCorp AT&T Universal Card, Jacksonville, FL

ABSTRACT

Table look-up is the most time-consuming part of many SAS programs. Base SAS offers a rich collection of built-in searching techniques. Merging, SQL joins, formats, SAS indexes - all serve the purpose of locating relevant data. For custom programming, SAS offers arrays, whose direct addressability lends itself to implementing just about any searching algorithm. Array-based lookup is not a ready-to-go food; it has to be cooked at home. However, it may result in dishes digested by the computer more easily, more programmatically nutritious, and with fewer computer resources ending up in the garbage disposer.

This paper shows how arrays can be used to organize the fastest class of in-memory table look-up -- direct-address searching. Three such techniques -- key-indexing, bitmapping, hashing -- are considered in a logical sequence using a real-life example of matching two data files by a common key. The results of benchmarking presented in the paper show that home-cooked direct addressing methods beat even the quickest ready-to-go tools like the "large formats" by a wide margin. As such, they can be indispensable in any massive data processing setting, where speed and efficiency considerations are paramount.

INTRODUCTION

Table lookup being one of the most frequent data processing operations, SAS provides a rich collection of built-in searching techniques. Merging, SQL joins, formats, indexes, to name a few, all serve the purpose of looking up relevant data. In addition, SAS Language incorporates arrays – the data structures ideal for implementing just about any searching algorithm "by hand". SAS arrays are not ready-to-go tools: Array-based lookups have to be custom-coded and tuned. However, this approach is more flexible and often results in programs that search faster and use fewer resources than the "heavy artillery".

This paper concentrates on a group of in-memory lookup methods based on direct or almost direct addressing into a temporary SAS array. First, we shall consider *key-indexed search.* Then we will try to expand its domain by viewing an *array as a bitmap*. Finally, we will see how to generalize the core idea of key-indexing to arrive at a hybrid search method called *hashing*.

To make the discussion less abstract, we will consider a common task of matching two data files by a common variable. This will help us how different lookup techniques compare to each other and to some of the ready-to-go methods such as "large formats" and SQL.

Consider an unsorted file SMALL containing N_SMALL records with a key variable KEY and satellite variable S_SAT. Another unsorted file, LARGE, with N_LARGE records, also contains KEY and a satellite field L_SAT. *Let us assume, for the time being, that the keys are integers.* Imagine that LARGE is so big that sorting is not an option; however, also assume that we have enough memory to hold all keys from SMALL at once. Under these conditions, *What is the most efficient way to subset LARGE based on the values of KEY in SMALL* to produce a file MATCH? SAS offers a number of ready-to-go tools based on in-memory table lookup. For example:

1. Compile unduplicated keys from SMALL into a format using CNTLIN= option, and search it for each KEY read from LARGE.
2. Join the files using BUFFERSIZE large enough to prompt the SQL optimizer to use SQXJHSH access method.
3. Load the keys from SMALL into a sorted array, then use a hand-coded binary or interpolation search.

With plenty of methods available, why try something else? Because there are faster and more efficient ways to do the trick!

I. KEY-INDEXING

Most of the ready-to-go and hand-coded searching methods are based on *comparing a search key to all or some keys in a memory table.* It makes them *principally limited* since generally, no comparison-based method can search in fewer iterations than binary search. We could therefore try to remove the limitation by *doing away with key comparisons* altogether. But is it possible to search for a key without at least one comparison? The answer is "yes" and given by a radically different searching philosophy called *direct addressing,* that finds its pure expression in *key-indexed search.* Its idea is simple. Imagine that all keys are 1-digit numbers from 0 to 9, and that SMALL has just 9 records:

```
OBS    | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
-------+---+---+---+---+---+---+---+---+----
 KEY   | 2 | 3 | 5 | 2 | 7 | 9 | 5 | 7 | 3
-------+---+---+---+---+---+---+---+---+----
 S_SAT | 1 | 2 | 3 | 0 | 4 | 5 | 6 | 9 | 7
```

Let us create a temporary array HKEY with *one node (location, address) allocated for each possible key value,* and initialize the contents of all buckets to a missing value. (In SAS, such initialization will be done be default. In the case the satellites might have legitimate missing values, the table can be primed using special missing values.) The array HKEY can be thought of as the following table in memory:

```
-----------------------------------------------
 H     | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
-------+---+---+---+---+---+---+---+---+---+---
 HKEY  | . | . | . | . | . | . | . | . | . | . |
-----------------------------------------------
```

Now, for each key from SMALL, let us look at the array location H *whose index is equal to the value of the KEY,* that is, simply at HKEY(KEY). Since we have created a separate bucket for each possible key value, we are always guaranteed to find the address with H=KEY. Let us check first if the node is empty, i.e. if HKEY(KEY) is missing. If it is empty, let us move the satellite S_SAT to H=KEY. After repeating this procedure for all nine test keys, HKEY acquires the following shape:

```
-----------------------------------------------
 H     | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
-------+---+---+---+---+---+---+---+---+---+---
 HKEY  | . | . | 1 | 2 | . | 3 | . | 4 | . | 5
-----------------------------------------------
```

What we have just created is termed a *key-indexed table*. It comprises two types of *entries*: *empty and occupied*. Inserting the satellite only when the node is empty retains its first instance corresponding to a repeating key value; otherwise, the last instance would be used. Either way, *duplicate keys are deleted automatically* as the table is loaded. If SMALL has no satellites or they are of no interest, the entries of the key-indexed table could be marked as occupied by moving 1 into the node, the unduplication effect remaining intact.

Given a search key to look for, all we have to do is examine the table location *whose index is equal to the key*. If the corresponding location is empty (missing), the key is not in the table. If it is occupied, the search has been successful, and the node contains the satellite value related to the search key. For example, if KEY=1, the search fails since the address 01 of the table is empty. If KEY=7, we have to look at the node 07. It is occupied; therefore, the key is found, and the node returns the satellite value HKEY(7)=4.

Note that *searching is implicitly incorporated in the process of loading the table*: To determine if the node is empty, we in effect search the table to find out if the key has already been marked in the table as present. If it has, it is a duplicate, and need not be inserted, so we can merrily proceed to the next record. The nature of the process makes it unnecessary to sort SMALL or insert the keys themselves, because effectively, the keys are "inserted" by making their corresponding nodes occupied by satellites or, in lieu of the satellites, by a non-missing value, e.g. 1.

The utter simplicity of key-indexing lends itself to a very simple DATA step implementation. Suppose, for example, that we are dealing with integer keys ranging from –4E6 to +4E6. The range thus naturally defines the bounds of the array HKEY representing the key-indexed table:

```
** Key-Indexed Load and Search **;

data match;
   array hkey (-4000000:4000000) _temporary_;
   ** load key-indexed table from small;
   do until (eof1);
      set small end=eof1;
      if hkey(key) = . then hkey(key) = s_sat;
   end;
   ** for each obs in large, search table and output matches;
   do until (eof2);
      set large end=eof2;
      s_sat = hkey(key);
      if s_sat > . then output;
   end;
run;
```

From the nature of the algorithm, it is clear that *no lookup method is simpler and/or can run faster than key-indexing*: It completes any search, hit or miss, without comparing any keys, via a single array reference. It also possesses the fundamental property: Its

speed does not depend on the number of keys "inserted" into the table, i.e. any single act of key-indexed search takes precisely the same time.

To see how well key-indexing performs, it was compared in load and search phases to formatting, SQXJHSH, and other methods presented below, for N_LARGE=2E6 and a number of N_SMALL values using SMALL and LARGE. The results shown in the Section "TESTING" testify that - at a high memory expense - key-indexing completely dominates the competition. For instance, it out-performs MERGE running against (pre-sorted!) input as 5:1.

Well, if key-indexed search is all that good, why not use it at all times and instead of everything else? Unfortunately, there is a fly on the ointment. As we have seen, key-indexing, due to its very nature, is practically applicable only when the lookup keys are integers falling in a limited range. For our test keys taking on only as many as 8,000,001 distinct values, sufficient array space can be allocated using about 64 MB of memory. Having 80 MB of memory, one can get away with 7-digit keys. However, to deal with 9-digit SSN, an array with 1 billion elements would be needed, which is almost impossible even with the modern memories, while 16-digit credit card numbers would make key-indexing a technical utopia.

On the other hand, there is a plenitude of real-world applications where key values do indeed fall in a limited range. In all such cases, key-indexing, with its blazing speed and simplicity, is beyond competition. Here are some examples:

1. SAS date is simply the number of days between a given date and 01JAN1960. Any date value, from the lowest possible in the SAS System up to year 4000 can be accommodated by a [-138061:380217] table, and it will occupy mere 4 MB of real storage (RAM).
2. SAS times. An array sized as [0:86400] will key-index any SAS time value.
3. ICD9/CPT4 codes. If some character is a letter, it can be converted into a number to 1-26 range, and then the entire code can be represented as a limited-range integer, probably not exceeding 1 million.
4. PIB2. informat maps any 1- and 2-byte character key onto the [0:65535] range.
5. Any fractional key if limited in range when multiplied by a suitable scaling constant.

So, key-indexing can be really useful and extremely fast in a variety of data processing situations. And yet, it remains inherently limited to the domain of restricted-range, integer keys. But the idea on which the method is based is so beautiful that it would be a shame to let it go underused just because of its greed for memory. It is worth trying to expand it. However, to do so, we must find a practical way to keep memory usage at bay.

The question is: How? First, let us observe that both the speed of key-indexing and its limitations rest upon several simple facts:

- The lookup table is directly addressed by the value of a key itself.
- The entire set of possible key values is addressable. It means that a separate node must be allocated for each possible value a search key can assume.
- No comparisons between the search key and any key in the table are made.

Based on these facts, we can devise two principal approaches that could loosen the restriction self-imposed by key-indexed search.

1. Keep all possible key values addressed, but *expand the addressable range of keys* by making much smarter use of the available memory resources. For, instance, we can try to key-index bits instead of bytes.
2. Eliminate *any restrictions* imposed on the nature of lookup keys. This can be done by dropping the requirements that (a) no two distinct keys shall reside in one node, and (b) no comparisons between the search key and keys in the table shall be allowed.

The first approach results in a technique called *bitmapping*. The second path leads to a more versatile hybrid searching method known as *hashing*.

II. BITMAPPING

Suppose we have a situation where the satellite information in SMALL is of no interest for us, and therefore we need not drag S_SAT through the memory to the output. In such a case, the key-indexed table only serves one purpose: To indicate whether a memory node, whose index corresponds to the key value, is empty or occupied. The occupied nodes can be populated with 1, and the empty ones can be either left missing or else initialized to zero. Now if a node number KEY contains 1, the key whose value is KEY is present. Otherwise, if the node number KEY contains 0, the key is not in the table. Hence, all a table node must be able to tell is whether it contains 0 or 1. Such functionality can be amply served by nothing more than a single bit. Yet looking back at our key-indexing implementation, we see that it uses full 8 *bytes*, the memory length of a numeric array item, to store a binary value. This is 64 times the difference! But if we to realize such a potential, how do we make efficient use of bits in such a setting?

If it were possible to have a temporary array with 1 bit per item, the question would not even arise – we would simply have the bits addressed directly via an index. Unfortunately, the shortest *memory* length reserved by a temporary SAS array element is not 1 bit but always full 64 bits regardless of the declared *expression* length. If, for example, a temporary array is allocated as $1, its item's expression length is 1, but its memory length is still full 8 bytes. Hence, to properly index the bits that compose an array element, some additional computations are needed.
At first glance, it seems natural to try bitmapping a character array with the shortest allowable memory length, $8, and the number of elements equal to the number of all possible key values divided by 64. This would obviously allow cutting memory usage 64 times, just as projected above. Say, for the time being, that we are dealing with 8-digit natural keys. In order to be able to address all of them, we will need, accordingly, 100 million (1E+8) bits. The equivalent amount of real storage, about 12 MB, is nowadays insignificant, especially compared to what key-indexing would require (760 MB). The entire universe of all possible keys can be thus covered by the bits of an array consisting of 1,562,500 8-byte character items. To mark a key as present, we might proceed as follows:

1. Declare ARRAY BITMAP (0:1562500) $8 _TEMPORARY_.
2. Compute X= INT(KEY/64). Now X points to the array item containing the bit having to be turned on.
3. Compute R=1+MOD(KEY, 64). Now R points to the correct bit.

To turn the bit on, we would compute X and R. If BITMAP(X) is blank (none of bits is on yet), we would set it to the binary zero "0000000000000000"X, then convert it to a 64-byte character variable BITSTR mirroring the bit content of BITMAP(X) , set R-th byte to 1, reconstruct BITMAP(X), and reinsert it into the proper array location:

```
BITSTR = PUT(BITMAP(X),$BINARY64.);
SUBSTR(BITSTR,R,1) = '1';
BITMAP(X) = INPUT(BITSTR,$BINARY64.);
```

This way, marking the keys from SMALL as present one by one, we would eventually "compile" the entire bitmap. Now, given a key from LARGE, how would we search for it? First of all, we would find X and test BITMAP(X). If it is blank, then we have a miss since obviously none of this item's bits has been turned on. Otherwise, if the entire item is not blank, one or more of its bits are on, and we have to compute R and check the R-th bit using either of the two expressions:

- `SUBSTR(PUT(BITMAP(X),$BINARY64.),R,1) EQ '1'`
- `INPUTN(BITMAP(X),'BITS1.'||PUT(R,Z2.))`

Both of them evaluate to 1 if the R-th bit is 1, and to 0 if the R-th bit is 0.

This looks fairly simple and actually does work! Unfortunately, when this scheme was implemented and tested, it returned lackluster performance. And, after some reflection, it should be no surprise: Just to examine a bit or turn it on, we in effect have to either memory-write full 64 bytes or use a modified informat which, unfortunately, executes quite slowly. (Note that in version 8.1, the situation might have changed because supposedly, 8.1 will allow to allocate character temporary arrays with $1 memory length.)

To achieve a decent searching speed (the name of the whole game), we must find a way to spot individual bits much more rapidly. In turn, this can only be achieved by performing some kind of fast computation on the entire array item rather than by breaking it apart – which immediately suggests using a numeric array instead of the character one. That brings about two interrelated questions:

1. If a numeric array is chosen to represent the bits of a bitmap, what sort of rapid operation can be performed on a numeric item in order to turn its R-th bit on?
2. Given a numeric item, what kind of rapid direct calculation can we use to find out whether R-th bit is on?

The first issue is resolved easily. If R-th bit of a numeric item is off, adding 2**(R-1) to the entire item is equivalent to turning its R-th bit on. Moreover, we do not even have to compute the binary power on the fly, since a series of consecutive binary powers can be pre-computed and stored in an auxiliary array. To resolve the second issue, it is first necessary to delve into a number of subtle caveats.

Caveat 1. It must be never attempted to turn a bit on if it is already turned on. Otherwise the added unity will become a carry turning one or more of the higher bits to 1, thus wreaking havoc in the entire bitmap. So, the answer to question 1 above cannot be found without answering question 2 first. (Of course, bit checking is superfluous if BITMAP(X) is missing, as none of the bits has been set to 1 yet.)

Caveat 2. Before the very first bit can be turned on, a missing item must be set to 0. However, initializing the entire array is not necessary, because at the searching stage, the missing items can be simply skipped.

Caveat 3. How many bits per array element can we use in such a manner? Since they must be limited to the mantissa, it leaves us with 56 usable bits per element under OS/390 and 53 bits under NT, so the divisor 64 in the formulae above must be changed accordingly. Thus, by switching to a numeric array, we sacrifice about 15% memory utilization for the sake of speed.

Now we can answer question 2 posed above, but to do so, let us first figure out how to determine whether, say, the 4th least significant *decimal* digit of a numeric variable N is not zero. Naturally, we would divide N by 1E+4 and find the remainder. If the latter is not less than 1E+3, the digit being tested is not zero; otherwise, it is zero. By induction, it can be concluded that a boolean expression

```
MOD(N,10**R) => 10**(R-1)
```

indicates whether R-th decimal place is "on" or "off". By the same token, the expression

```
MOD(N,2**R) => 2**(R-1)
```

returns 0 or 1 depending on whether R-th bit of N is turned off or on. So, in effect, the result to which the expression above evaluates is set directly to the value of R-th bit of N's mantissa. That gives us all we need to key a numeric-array bitmap:

Step 1. Allocate a numeric temporary array BITMAP bound from 0 to 10**[number of key digits]/M, where M=56 or 53, depending on OS. Create an auxiliary array B and fill it with the serial powers of 2.
Step 2. Read a record with KEY from SMALL.
Step 3. Locate the array element: X=INT(KEY/M). If BITMAP(X) is missing, then set it to 0.
Step 4. Locate the right bit within the array element: R=MOD(KEY, M). If BITMAP(X) is missing, go straight to Step 5. Otherwise check the bit. If it is already 1, then the key is a duplicate: return to Step 2.
Step 5. Turn R-th bit on: BITMAP(X) ++ B(R), and go back to Step 2.

Given a bitmap "compiled" by the procedure just detailed, searching for a key is equally simple:

Step 1. Read a record from LARGE.
Step 2. Locate the array element: X=INT(KEY/M). If BITMAP(X) is missing, the search is unsuccessful, so go back to Step 1.
Step 3. Locate the bit: R = MOD(KEY, M).
Step 4. Check the bit. If it is zero, the key is not found, hence go to Step 1. Otherwise, write the record out and go back to Step 1.

Translating the algorithm into the SAS Language is now straightforward, but first the possibility of negative keys should be accounted for. With key-indexing, it is natural and easy to do by giving the lower bound of the table the lowest negative key value. In the case of a bitmap, we have to rummage around with the keys a little bit first: Shift them up by the absolute value of the lowest key (say, MINKEY), rescale the upper BITMAP bound accordingly, and leave the lower bound at 0.

```
*** Bitmap Table Load And Search ***;

%let m      =   56; *** if OS/390, else 53;
%let minkey = -4e6;
%let maxkey = +4e6;
%let lo = %sysfunc(floor(&minkey/&m));
%let hi = %sysfunc(floor(&maxkey/&m));
%let hb = %eval(&hi - &lo);

data match (keep=key l_sat);
   array bitmap (0:&hb) _temporary_;
   array b      (0: &m) _temporary_;

   ** precompute powers of 2;
   do x=0 to &m; b(x) = 2**x; end;

   ** load the bitmap from SMALL;
   do until (eof1);
      set small end=eof1;
      x = int((key - &minkey) / &m);
      if bitmap(x) eq . then bitmap(x) = 0;
      r = key - &minkey - x*&m;
      if mod(bitmap(x),b(r+1)) < b(r) then bitmap(x) ++ b(r);
   end;

  ** search bitmap for keys from LARGE, output matches;
   do until (eof2);
      set large end=eof2;
      x = int((key - &minkey) / &m);
      if bitmap(x) eq . then continue;
      r = key - &minkey - x*&m;
      if mod(bitmap(x),b(r+1)) => b(r) then output;
   end;
   stop;
run;
```

The macro assignments at the top take care of the necessary shifting and rescaling. M=53 will work under any OS; however, under a system with 56-bit mantissa, say OS/390, it is possible to choose M=56 instead and thus improve memory utilization. The first DO loop populates the auxiliary array B with the powers of 2. The process in the <EOF1> DO loop can be thought of as "bitmap compilation". Finally, the <EOF2> DO loop performs the actual bitmap search and outputs the records from LARGE whose key imprints are found in the bitmap. Because of the extra computations, bitmapping runs about 50 per cent slower than key-indexing, yet it uses 53 to 56 times less memory and is still twice as fast as MERGE after sorting!

In the approach implemented above, the bitmap compilation loop resides in the same step where it is searched. However, a bitmap can also be compiled in a separate step, stored in a file, and reused thereafter. To do so, we would only have to change the DATA statement to, for instance,

```
data lib.bitmap (keep=byte8);
```

and replace the entire <EOF2> loop with

```
do x=0 to &hb;
   byte8 = bitmap(x);
   output;
end;
```

The name of the variable BYTE8 is, of course, absolutely arbitrary. At this point, the BYTE8 in each observation of the SAS data set LIB.BITMAP corresponds to exactly one array item of the would-be bitmap. A bitmap saved in this manner can be then utilized any time a file similar to LARGE needs to be subset, validated, scrubbed, etc., based on the bit pattern stored in the bitmap, only in the beginning of the searching step, the array BITMAP must be loaded in memory from LIB.BITMAP one item at a time. This is accomplished by replacing the <EOF1> DO loop with

```
do x=0 to bitobs-1;
   set lib.bitmap nobs=bitobs;
   bitmap(x) = byte8;
end;
```

Principally, key-indexing and bitmapping are nearly twins:

- The table size is dictated solely by the overall key range and not by the number of lookup keys.
- Neither the "driver" file nor "master" file has to be sorted.
- Duplicate lookup keys are eliminated automatically as the table is "loaded".
- The speed of searching does not depend on the number of keys.

However, there are a few differences worth noting as well:

- Given the same memory resource, bitmapping has the addressable key range 53 to 56 times that of key-indexing.
- A bitmap cannot be used for dragging lookup satellites through memory into the output.
- With key-indexing, locating and rejecting a key takes exactly the same time. With bitmapping, a successful search requires, on the average, slightly more computing, and therefore, more time.

Because of its relatively wide key range and purely direct-addressing nature, bitmapping operates with incredible speed in a niche no other searching method can touch. As an example, imagine SMALL file with 50 million 8-digit keys (hardly "small" but let us stick with the name), and LARGE with mere 100 million records – figures not unusual in data warehouses nowadays. Sorting either one for MERGE is not exactly painless operation, especially if L_SAT tail is long, or say LARGE is actually is a view into a RDBMS table. Storing the lookup keys in a format is practically hopeless, as memory usage by a format usually tops 600 MB already at mere 10 million keys. Key-indexing would consume about 800 MB. A hash table (described later) would need at least 400 MB. However, a bitmap can be safely compiled with 120 million bits of temporary array storage in the worst case scenario (M=53). It means that it can be easily accommodated within only 15 MB of memory! If we decide to store the bitmap on disk, it will take about 1.9 million 1-variable observations; loading such a file into an array is a matter of several seconds. What is more, the lookup speed and memory usage will remain exactly the same, no matter whether we have 100, 100,000, or 100,000,000 keys to search.

Thus, the bitmapping niche can be defined as "no-matter-how-many-short-keys". Bitmap is a champion when we only need to rapidly find out if the record with a given key should be selected, and if memory resources are sufficient for key-indexing the entire key range into memory bits.

But how can we capitalize on direct addressing if the keys have a huge, say 16-digit, range, or are long character strings (256-radix integers), and therefore neither key-indexing nor bitmapping can do the job? Welcome to *hashing* !

III. HASHING

As noted above, compared to key-indexing, bitmapping changes nothing principally – it simply expands the workable universe of keys about 53+ times by using memory more efficiently. Hashing methods approach the problem quite differently: They eliminate the requirement of a separate slot for each possible key *and* allow some amount of comparisons between the search key and keys in the table. A simple example might be the easiest way of making the idea transparent. Let us suppose that SMALL contains just 10 3-digit keys:

185 971 400 260 922 970 543 532 050 067

To use key-indexing, we would have to allocate a table sized as [0:999] and map each key to the node corresponding to its value. Out of 1000 table nodes, only 10 will end up occupied, while the rest will play the role of placeholders, that is, will be simply wasted! The crucial question is, therefore, *Can we get away with a smaller table* of a reasonable size, only somewhat larger than the number of keys at hand, and still be able to take advantage of direct addressing?

Let us choose some number HSIZE greater than the number of keys N_SMALL in SMALL, for instance, 17, and allocate an array sized as HKEY(0:17). Let us agree to call the array HKEY the *hash table*, HSIZE - the *hash table size*, and the ratio N_SMALL/HSIZE - the *load factor*. Thus, the load factor shows the number of lookup keys relative to the total number of nodes in the hash table, in other words, how *sparse* the hash table is. In our example, the load factor equals 0.588, that is, the hash table is about 41 percent sparse.

Imagine some rapidly-computing function H(KEY) taking a key as an argument and returning an address into HKEY, unique to each key supplied, so that H(KEY) would map each key to its own location in a one-to-one manner. Were such *perfect hash function* available, we would only have to plug it in the code for key-indexed search and be done. Such functions *are* possible; however, they are quite difficult to discover, and once one is found, it can only be used for the same set of keys: Adding just an extra key will ruin everything.

A lot less rigid method can be obtained if we give up the one-to-one requirement of the relationship between the keys and table addresses and let H(KEY) *map two or more distinct keys to the same location* in HKEY. Of course, if more than one key is sent to the same node, a phenomenon termed a *collision* occurs, and we must invoke some *collision resolution policy* in order to tell the keys apart in the process of insertion or searching. Thus, we arrive at the *core concept behind hashing*. If the hash function H(KEY) is good enough to map only a few keys to any particular hash address H, in other words, spread the keys evenly throughout the table, we can adopt the following strategy:

1. Given a search KEY, use H(KEY) to *hash* KEY to some address H in the table.
2. If the address H is empty, the search is unsuccessful, since no key has ever hashed to H.
3. If the address is occupied, search all the keys that have hashed to H *sequentially*.

Thus, hashing is a typical *hybrid algorithm*: It combines direct addressing with *sequential search*, a method based on comparisons between keys. The *average* number of keys mapping to any hash node equals N_SMALL/HSIZE, i.e. the load factor. If the hash table is not full and the keys are spread uniformly, the average number of key comparisons required to find or reject a key is less than 1. Also, searching for a key should be the faster, the sparser the table is. So, to make a good practical use of a hash table, we ought to:

1. Choose a proper hash function H(KEY).
2. Find an efficient way of resolving collisions.

Before we could formulate the requirements for a *good hash function* let us consider how a *bad hash function* would behave. On one extreme, if a function is lightning fast but maps all keys or their majority to the same hash address, it defies the very purpose of distributing keys among different addresses: For then we would have to search all these keys sequentially! Hence, for a good hash function, it is paramount that it should map the keys evenly across all hash table nodes, without burdening some addresses with huge clusters of keys and leaving the rest of the slots empty. On the other extreme, if a function maps the keys extremely uniformly but takes an inordinate time to compute, it is of no good use, either - direct addressing itself would become a bottleneck. Now we can formulate some rational requirements a good hash function should satisfy:

1. Its computation should be as fast as possible.
2. It should distribute the keys uniformly to minimize collisions.
3. It must return addresses in the range from 0 to HSIZE-1.

There is a number of mapping methods conforming to these requirements [1]. We will discuss and use the simplest technique called the *division method*, which utilizes the remainder modulo:

```
H = MOD (KEY, HSIZE);
```

It certainly fits requirement 3, since for any value of KEY, this function always returns an integer in the range from 0 to HSIZE-1. It also satisfies requirement 1, for although it incorporates a division, its computation is still reasonably fast. However, to satisfy requirement 2, the value for HSIZE must be chosen rather carefully. The number theory tells us (see, e.g., [1]) that if HSIZE is a *prime number* and not too close to the power of 2, the MOD function tends to spread the keys uniformly across the nodes, with the majority of the occupied nodes receiving 1 to 3 keys. Let us see how this would work for our sample set of 10 keys. If we choose the "target" load factor as 0.625 and divide it into the number of keys, we obtain 16. The first prime number greater or equal to 16 is 17, so let us select HSIZE=17. (The actual load factor is now 10/17 = 0.588.) We may want, therefore, to allocate the table as

```
ARRAY HKEY (0:17) _TEMPORARY_;
```

To obtain a hash address, KEY is divided by HSIZE=17, and the remainder H is computed. H points to the H-th slot in the table where KEY must be inserted. Repeating this operation for every test key, we end up with the following pattern (the numbers atop the table indicate the corresponding array buckets, and the colliding keys are shown in boldface):

```
00    .   .   .
01  970   .   .
02  971   .   .
03    .   .   .
04  922   .   .
05  260 532   .
06    .   .   .
07    .   .   .
08    .   .   .
09  400   .   .
10    .   .   .
11    .   .   .
12    .   .   .
13    .   .   .
14    .   .   .
15  185   .   .
16  543 050 067
17    .   .   .
```

The keys 970, 971, 922, 400, and 185 all map to their slots in HKEY one-to-one. The keys 260 and 532 produce a single collision at the address 05, and the keys 543, 050, and 067 result in a double collision in the node 16. If this table is to be stored in memory and searched, the collisions at the locations 05 and 16 have to be *resolved*.

Before we move on, let us solve the small technical problem of finding the correct prime HSIZE, given the file SMALL and load factor LOAD. Instead of computing it by hand or from a table of primes, it can be calculated and stored in a macro variable LOAD dynamically using a short (and extremely fast) SAS program:

```
%let load = 0.8;
data _null_;
   do p=ceil(p/&load) by 1 until (j = up + 1);
      up = ceil(sqrt(p));
      do j=2 to up until (not mod(p,j)); end;
   end;
   call symput('hsize',left(put(p,best.)));
   stop;
   set small nobs=p;
run;
```

As we already know, selecting a decent hash function is just one part of the deal: No matter how good the function is, it is practically guaranteed that some keys will hash to the same addresses in the table, so we have to devise a method of resolving collisions. This is another point at which hashing radically deviates from key-indexing and bitmapping where we needed not store the keys in the table itself. With hashing, the keys themselves have to reside in the table, because they will have to be compared to a search key unless the search key hashes to an empty node. Various *collision resolution policies* differ in the ways by means of which colliding keys are stored, linked as "belonging" to the same hash address, and traversed. Let us consider them one at a time.

1. Separate Chaining

One way of resolving collisions suggests itself naturally once we cast a rapid glance at the distribution of our 10 keys among the addresses of the hash table shown in the previous section. Keys "attached" to each occupied address form visible "chains" - consisting of a single key in the absence of collisions. Making use of such chains to resolve collisions is logically called *separate chaining*.

There are two ways the chains of keys can be utilized in terms of the SAS DATA step. First, the keys comprising the chains could be stored *outside the table* by placing them in the occurrences of a two-dimensional array. A significant drawback of this method, however, is poor memory utilization. If we have 100,000 keys in SMALL and a single "bad" address colliding 10 keys, we will be forced to create a 2-dimensional array sized as (0:10, 0:100000) to resolve the collisions. Even with the load factor 1, it requires 10 times the memory the keys would occupy by themselves. On the positive side, the 2-dimensional chaining is quite fast, and it can work with load factors greater than 1, if necessary. So, if good memory utilization is not a paramount consideration, the method could be recommended. (Feel free to contact the author for the details of implementation.) What is more, because 2-dimensional separate chaining provides a natural way of working with long chains, it turns out to be extremely valuable when hashing is used for *external* searching, i.e. searching on disk rather than in high-speed memory. It will be mentioned once again in the section "Applications".

Returning to the main course of the paper, memory-resident hashing, the idea of chaining can be exploited in a much neater fashion than by using a huge, and mostly wasted, 2-dimensional array! Once the philosophy of allocating the main storage for colliding keys is changed from sequential to *linked* , we arrive at an extremely elegant collision resolution policy, both very fast and reasonably memory-efficient.

2. Coalesced Chaining

The *core idea* of this method is to place the chains of colliding keys into the hash table itself and combine the keys mapping to the same node in a linked list, with the head residing at the colliding address. Setting the last link of each chain to null designates the end of the chain, thus helping us tell where to stop when the list is traversed serially. Since the linked lists are thus allowed to overlap in the hash table sharing the same storage locations, this approach is termed *coalesced list chaining*, or, shorter*, coalesced chaining*. To make it possible, all we need is a numeric array of link items LINK, parallel to the "main" hash table where the keys are inserted. It is extremely important that in order for this method to work, *at least one entry in the table must be empty.* Otherwise if the table is full, there would be no empty node where a null link should point in order to terminate the loop traversing the list. Since a table allocated as (0:HSIZE) has HSIZE+1 entries, but the modulo-based hash function only addresses HSIZE nodes from 0 to HSIZE-1, this requirement will always be satisfied. Let us agree to *always leave the address 0 empty* by hashing keys as

```
MOD (KEY, HSIZE) + 1.
```

That is, if KEY modulo HSIZE is 05, it will map to the address 06. Adding a unity to the modulo has the additional advantage of allowing to use 0 as a null value for the end-of-chain. As stated above, we have to allocate two parallel arrays, one for the hash table itself and one to hold the links:

```
ARRAY HKEY (0:&HSIZE) _TEMPORARY_;
ARRAY LINK (0:&HSIZE) _TEMPORARY_;
```

Now we are ready to spell a detailed plan of loading a coalesced list hash table:

STEP 1. Set a counter variable R to the top address: R=HSIZE.
STEP 2. Hash: H=MOD(KEY,HSIZE) + 1.
STEP 3. If LINK(H)= ., the node is empty, no list is attached to it. Go to step 8 to insert the key.
STEP 4. Otherwise *traverse the chain* to find if the key is already in the table:

A. If KEY=HKEY(H) the key is duplicate. Get the next key and return to step 2.
B. Else If HKEY(H) is not 0, it is not the end of the list yet. Set H= HLINK(H) and repeat step 4.

STEP 5. Find an empty address closest to the top: Decrement R until LINK(R)= .
STEP 6. Store the key at this address: HKEY(R)=KEY.
STEP 7. Memorize where KEY actually belongs: LINK(H)=R; H=R.
STEP 8. Insert KEY into the address H and set its link to null: HKEY(H)=KEY; LINK(H)=0. Now the node has been marked as occupied.

Let us see, by inserting one key at a time, what kind of linked list hash table is actually created by this process for our 10 sample keys and HSIZE=17. Please refer to the resultant table, Table A1, in the Appendix. The key being inserted, as well as the colliding keys, are shown in boldface. The two bottom rows represent the final state of the loaded table. A peek at it quickly reveals how the collisions are being handled:

All the way up to the attempt to insert KEY=532, each key finds its unique slot without any contention. But at KEY=532, we have the first collision, because it hashes to the address 06, already occupied by the key 260. In accordance with the algorithm, we look at the link at address 06 and find it to be zero. Therefore, it is the end-of-chain - and the only key in the chain so far. The first available empty address counting from the top of the table (right to left on the diagram above) is 15. The new key 532 goes there, and the node is marked as occupied with 00 in its link field. To tell the key 260 where its successor in the chain, 532, resides, we store the address of 532, i.e. 15 in the link field of node 06 that holds 260.

The keys 543, 050, and 067, all hashing to the address 17, are placed in the table in the same manner. The first key in this chain, 543, must be stored at this address, and there it is. The link of the address 17 is not 0, hence, it is not the end of the list. Instead, LINK(17)=14. This is the node where the next key in the chain, 050, must reside, and it is there, indeed. But once again, the list must continue because the address 14 contains a non-zero link, LINK(14)=13. Finally, we find the key 067 in the node 13, and it is the last key colliding at the hash address 17, for LINK(04)=0.

As opposed to the colliding keys, the keys hashing to their addresses uniquely, bump in a zero link at once. For example, the key 922 hashes to the address 05, with LINK(05)=0. Now the reason of leaving the address 00 always empty should be transparent. We are using 0 to indicate the end of chain (null link), but actually a chain traversal terminates when a null, i.e. missing value, link field has been encountered. A zero in a link field will always lead to address 00, and since it is always missing, the traversal will inevitably stop.

At this point, it should be crystal clear how this linked table organization facilitates searching. Suppose that we need to look for KEY=051. It hashes to the address 01 where the link field LINK(01) is missing. That is, none of the keys in the table has ever hashed to this address, hence the key is not in the table. However, searching for KEY=047 that hashes to the address 14, we are in a different situation, because LINK(14)=13 is not null. Hence, some other keys in the table may have also hashed to this address, and so the entire chain must be examined for the presence of 047. Since the key 050 in the node 14 does not match the search key, we have to look at the next key in the chain located at the address 13 to which LINK(14) is pointing. The key 067 in the node 13 does not match the search key 047, either, and it is the end of the list since LINK(13)=0. This, finally, points to address 00, whose link is (always) null. Hence, 047 is not present in the table.

As an example of a successful search, let us try to find KEY=050. It hashes to the address 17 with the key 543, different from 050. But it is not the end of story: LINK(17) = 14 is not null telling us that the next comparison should be made with HKEY(14) = 050. At this point, the search key is found, the list need not be traversed any further, and the process of searching terminates successfully.

After this walk-through, it should not take a Certified SAS Programmer to schedule *hash searching:*

STEP 1. Hash: H = MOD(KEY,HSIZE) + 1.
STEP 2. If LINK(H) = . then search terminates unsuccessfully.
STEP 3. Traverse the list. If KEY = HKEY(H) then search terminates successfully.
STEP 4. Else examine LINK(H). If LINK(H) = 0 then KEY is not found. Stop.
STEP 5. Next link. Set H = LINK(H) and go to step 3.

Now we can finally give a solution to the matching problem using coalesced chain hashing. An additional array parallel to the hash table and links, HSAT, is used to pull the satellites from the lookup file SMALL. If we do not need S_SAT it may be omitted along with the corresponding instructions.

```
** Coalesced Linked List Chaining **;

data match (keep=key s_sat l_sat);
   array hkey (0:&hsize) _temporary_;
   array link (0:&hsize) _temporary_;
   array hsat (0:&hsize) _temporary_;
   ** load and link hash table using keys from SMALL;
   do until (eof1);
      set small end=eof1;
      h = mod(key,&hsize) + 1;
      found = 0;
      if link(h) > . then do;
         link traverse;
         if found then continue;
         do r=&hsize by -1 until (link(r) = .); end;
         link(h) = r;
         h       = r;
      end;
      link(h) = 0    ;
      hkey(h) = key  ;
      hsat(h) = s_sat;
   end;
   ** search table for key from LARGE, output matches;
   do until (eof2);
      set large end=eof2;
      found = 0;
      h = mod(key,&hsize) + 1;
      if link(h) > . then link traverse;
      if found then do;
         s_sat = hsat(h);
         output;
      end;
   end;
   stop;
   traverse: if key = hkey(h) then found = 1;
             else if link(h) ne 0 then do;
                h = link(h);
                go to traverse;
             end;
run;
```

Since the code intentionally parallels the algorithm above, you should not be surprised to find the GO TO instruction. Those believing that "GO TO" and "structured programming" cannot peacefully coexist, may prefer to rewrite the TRAVERSE block at the expense of an extra comparison at the bottom of the loop as

```
do until (found);
   if hkey(h) = key then found = 1;
   else if link(h) = 0 then leave;
   else h = link(h);
end;
```

Now that the coalesced chaining routine is ready, it can be tested using the same sample files as have been used for key-indexing. The program was tested for the load factors 0.5 and 0.8 (50 and 20 per cent sparse table). The results shown in Table 1 generally corroborate the conjectures made earlier:

1. The sparser the table, the faster the search, but it consumes proportionally more memory.
2. If the hash table is relatively sparse, its lookup time does not depend on the number of keys in the table.
3. It runs somewhat slower than key-indexing, but still 2 to 3 times faster than even SQLXJHSH and is much easier on memory than the rest of the methods tested.
4. Just like with key-indexing and bitmapping, hashing needs neither sorting nor removing duplicates.

Judging from the test results (see the "Benchmarking" section below), chaining performs very well, with the added benefit of not being too sensitive to the sparsity of the table (load factor). However, it requires an extra array to hold the links. The link array is as large as the hash table itself, so if SMALL is actually not quite small, the additional memory burden can be significant. In a different class of collision resolution policies collectively called *open addressing,* memory utilization is improved by doing away with the links altogether. We shall discuss two such methods: Linear probing and double hashing.

3. Open Addressing with Linear Probing

The main idea behind open addressing can be described as follows. Just like in the case of coalesced chaining, keys are stored in the hash table itself. Suppose we have a key KEY to be loaded in the table. First, let us hash it using the division method, but straight, i.e. without adding a unity:

```
H = MOD(KEY, &HSIZE);
```

If H points to an empty slot, we simply store the key at this location. If the slot H is occupied, we have a collision. Let us compare the key with the one already sitting at H. If the keys are equal, the current key should be discarded because it is a duplicate, and the next key obtained from the input. Otherwise we have to find a different slot for the current KEY. Let us step down the table one or more times one node at a time. If H becomes less than 0, i.e. we have stepped off the bottom of the table, let us return to its top, and continue to do so in this wrap-around cycle until having encountered either a duplicate key - in which case we just stop and get the next key, or an empty node - in which case we insert the colliding key into it. The method of resolving collisions just described is called linear probing – for the table is being "probed" using a fixed *probe decrement*, C=1, regardless of the key.

Since we have HSIZE+1 nodes in the table, but can only address HSIZE nodes from 0 to HSIZE-1, at least 1 location in the table, the top one, will always remain empty, thus preventing the loop from iterating infinitely. Let us observe how this process works step by step while our 10 test keys are being inserted in the table sized as [00:17] (See the dynamic table Table A2 in the Appemdix):
All the keys up to and including 543, hash uniquely to their very own nodes. However the next key, 532, hashes to the same address 05 as the key 260, already sitting there. According to the plan outlined above, we step down the table until an empty slot is found. This happens at H=03, and so 532 is inserted at this address. The next key, 050, is not too friendly, either, since it claims the same seat, H=16, that is already assigned to 543. The nearest free slot down the table is H=14, and so that is where 050 goes. But the next key, 067, happens to hash to the same H=16 again! Now, to find where to

place this one, we have to travel all the way to H=13, at which point the hash table is loaded and ready to be searched.

As in the case with chaining, the process of loading the table readily suggests the way of looking it up. As an example of an unsuccessful search, let us look for KEY=51. MOD(51,17) yields 0, and address 00 is empty. Hence, 51 is not in the table, period. Searching for KEY=66 is more complex, since it hashes to H=15 occupied by the key 185. Since there is no match, we look at the next key, 050, one node down the table, find a mismatch again, and proceed in this manner all the way to H=12, which is empty. It means that 66 is not in the table, either, for if it had been inserted in the table, it would have been found before an empty node is hit.

As an example of a successful search, let us look for KEY=922. It hashes to H=04, and we have an immediate match. Another successful search for KEY=067 is a bit more laborious, for it hashes to H=15, and we have to step down the table twice until the key is identified in H=13.

From these simple examples, it should be clear how linear probing reduces the number of probes sequential search would require. In the worst case scenario in the example above, linear probing examines 5 keys until it either finds or rejects a search key; but on the average, with the load factor 10/17, the number of comparisons will be close to 2 per search, hit or miss. Sequential search, on the other hand, would require, on the average, 8 probes for a hit and 17 for a miss. Now it is time to translate all these verbal speculations into SAS:

```
** Hashing by Open Addressing with Linear Probing **;

data match (keep=key s_sat l_sat);
   array hkey (0:&hsize) _temporary_;
   array hsat (0:&hsize) _temporary_;
   ** load table with keys from SMALL;
   do until (eof1);
      set small end=eof1;
      do h=mod(key,&hsize) by -1 until (hkey(h)=. or hkey(h)=key);
         if h < 0 then h = &hsize-1;
      end;
      hkey(h) = key  ;
      hsat(h) = s_sat;
   end;

   ** search table for each key from LARGE and output matches;
   do until (eof2);
      set large end=eof2;
      do h=mod(key,&hsize) by -1 until (hkey(h) = .);
         if h < 0 then h = &hsize-1;
         if hkey(h) = key then do;
            s_sat = hsat(h);
            output;
            leave;
         end;
      end;
   end;
   stop;
run;
```

The main advantage of this scheme, as it is evident from the code above, is its profound simplicity. In fact, none of existing hashing methods is simpler or more straightforward than the linear probing. And, if the table is sparse enough, it performs quite well, too! As a rule of thumb, the linear probing will do the hashing job just right if about half of all nodes in the table are left empty, i.e. with the load factor of about 0.5. However, as the table gets fuller, its performance deteriorates the quicker, the fuller the table is. With load factors above 0.9, the only good things we can say about the linear probing is that it is simple and it works, albeit slowly but surely.

The reason why linear probing exhibits such a behavior in a crowded table lies in the phenomenon called *primary clustering*. When looking for an unoccupied node for a colliding key, we fill out the very first empty location we come across. Therefore the groups of adjacent occupied addresses tend to aggregate, forming clusters of keys. Worse still, the clusters can bridge together forming bigger clusters. (For instance, consider what happened to the clusters 970, 971 and 922, 260 in our test table above.) Hence, if the table is not quite sparse, we will eventually have to travel through almost the entire table before finding an empty location to either insert a key or stop the loop in the case of an unsuccessful search.

One apparent way to alleviate the problem of primary clustering is to try stepping through the table using more than one node at a time. It turns out to be a very good and sound idea. Complemented with another good and sound idea, it leads to the open addressing method called *double hashing* that eliminates primary clustering entirely. Therefore, it would allow achieving the same speed of search with a less sparse table resulting in a superior memory utilization.

4. Open Addressing with Double Hashing

So, as suggested above, let us try stepping down the table using some probe decrement C > 1. However, the value of C must be chosen rather carefully. With linear probing, it is guaranteed by virtue of C=1 that in the wraparound process of probing the table, each node can be examined, and examined exactly once. What kind of value should C > 1 have to retain the same fundamental property? It follows from the number theory that if the probe decrement C and the hash table size HSIZE are *relatively prime*, this property holds. Now remember, we have chosen the table size prime in order to minimize the collisions. Therefore, selecting C as any integer between 1 and HSIZE-1 inclusively will make C and HSIZE relatively prime.

However, there is one more important consideration helping choose C even wiser. Namely, if we could make C depend on the key in a random yet deterministic manner, it would help spread diversify the probing sequences belonging to different keys, and hence distribute the keys even more evenly in the table. MOD function, as we know, possesses quite good randomizing capabilities (which is why it is used as a hash function in the first place). Therefore, if we compute C as

```
C = 1 + MOD(KEY, HSIZE-2),
```

it will both distribute the values of C among the keys pseudo-randomly and guarantee that any C value obtained this way and HSIZE are relatively prime. Indeed, C can result in nothing else but some integer between 1 and HSIZE-2, and since HSIZE is prime, C and HSIZE will always by relatively prime. In practice, such a choice for C has been proven to work satisfactorily in most cases.

In essence, what we are doing is hashing the key the second time to obtain the probe decrement, which is why this method of resolving collisions is called double hashing. Of course, the second hashing is an extra computation, but it is not too expensive, and it is situated outside the inner loop of the routine. Therefore, we should not expect a lot if computational overhead, all the more that eliminating primary clustering turns out to be much more important from the standpoint of performance.

With the exception of C > 1, the basic linear probing algorithm remains intact. Like before, if in the process of decrementing H, it is found that H < 0, that is, we have fallen off the bottom of the table, we wrap around it; only in this case, instead of returning right to &HSIZE-1, we shall return to H+&HSIZE. For example, if HSIZE=17, C=5, and we have found that H=-2, we shall wrap around the table to $-2+17=15^{th}$ array item. Let us see, using the set of our experimental keys, what kind of hash table this process will compile, starting with an empty table and inserting one key at a time. The dynamic table A3 created by this process is shown in the Appendix.

Comparing the final state of the table with that compiled by linear probing, we clearly see that it is much more uniform, with the clusters of keys well separated from each other, and with no cluster containing more than 3 keys. It means that no matter what key we are looking for, no search will require more than 3 comparisons between keys in the worst case scenario.

While theoretically, double hashing is significantly more involved that linear probing, amending the program for linear probing in order to accommodate double hashing boils down to a single line of code preceding the main hash, and a subtle change in the way to wrap around (below, all the changes to the linear probing routine are shown in upper case):

```
** Open Addressing with Double Hashing **;

data match (keep=key s_sat l_sat);
   array hkey (0:&hsize) _temporary_;
   array hsat (0:&hsize) _temporary_;
   ** load table with keys from SMALL;
   do until (eof1);
      set small end=eof1;
      C = 1 + MOD(KEY,&HSIZE-2);
      do h=mod(key,&hsize) by -C until (hkey(h)=. or hkey(h)=key);
         if h < 0 then H ++ &HSIZE;
      end;
      hkey(h) = key  ;
      hsat(h) = s_sat;
   end;
   ** search table for each key from LARGE and output matches;
   do until (eof2);
      set large end=eof2;
      C = 1 + MOD(KEY,&HSIZE-2);
      do h=mod(key,&hsize) by -C until (hkey(h) = .);
         if h < 0 then H ++ &HSIZE;
         if hkey(h) = key then do;
            s_sat = hsat(h);

            output;
            leave;
         end;
      end;
   end;
   stop;
run;
```

Let us take a look at the performance Table 1. With a 50 per cent sparse table, double hashing runs just a tad slower than coalesced chaining with 20 percent sparsity, but on the positive side, it uses less memory. So, double hashing is quite fast; it even loads an equally sparse table somewhat faster than the chaining because it does not have to worry about the links. The fact that a searching method based on stepping through the table before an empty node is found works so well, may seem surprising. However, this is a direct result of the double hashing probing methodology. In fact, independent experiments (corroborating theoretical conclusions) show that if the table is no more than half full, double hashing makes on the average no more than 2 comparisons per miss, and no more than 1.3 comparisons per hit.

IV. HASHING WITH NON-NATURAL KEYS

As the test results show, hashing performs admirably by any account regardless of the collision resolution policy being used. However, even though hashing schemes we have discussed impose no limitations on the range of keys, they have been developed under the assumption that the keys are integers. Now it is time to remove this restriction as well.

The reason it is possible to do is rooted in the fact that in its final stage, hashing is strictly comparison-based, which effectively renders the nature of keys non-critical. Both hashes and traversals are used merely to minimize the number of comparisons necessary to carry out a search, yet the final hit-or-miss decision - if a hash address is not empty - is made by comparing some keys in the table to the search key. Therefore, in order to be able to operate on keys of any type, we only have to figure out how to hash a key if it is not a non-negative integer. For the hash function to remain uniform and fast, it is critical to adhere to a few simple rules:

- Hashing process should involve as many key characters as possible.
- String operations and conversions must be minimized.

Let us consider a number of distinct practical situations.

1. Fractional Signed Keys

In this case, we can simply rescale each key before hashing by multiplying it by a suitable integer constant and adding another constant to the result if necessary. For instance, if our keys are in the decimal form X.Y, multiplying each key by Y would suffice. If, in addition, they can be negative, we would simply add an integer Z known to exceed the largest absolute value a negative key can assume. So, the entire change to the programs above needed to accommodate fractional signed keys would be using

```
MOD ( KEY*Y + Z, HSIZE)
```

in the hashing formulae instead of the straight modulo. It will not cause any noticeable deterioration in performance, since in SAS this kind of computation is quite fast.

2. Digital Strings

First of all, since digital strings are character variables (consisting of digits only), the hash table itself will have to be declared as a character array of appropriate expression length, for example:

```
ARRAY HKEY(0:&HSIZE) $12 _TEMPORARY_;
```

Hashing a digital string is a simple matter of using the INPUT function and an appropriate numeric informat. For example, if the keys were 16-digit account numbers stored in a character variable, we could simply choose

```
MOD (INPUT(KEY,16.), HSIZE)
```

as our hash function. Another way to hash a digital string is to apply the same methods that are used for hashing character variables in general (see below).

3. Generic Character Keys

Numerous techniques have been developed to hash arbitrary character keys well [2, 3, 4]. Almost all of them are based on breaking a character key apart and then involving the individual bytes into a sort of computation resulting in an integer in the range [0:HSIZE-1]. Some of these methods, for instance, universal hashing, actually guarantee to hash *any* input evenly. However, they are based on the assumption that the process of extracting individual bytes from a string is very fast. Unfortunately, this is exactly what is slow in SAS. We would be much better off converting a character string to an integer in a single shot, and PIBw. informat is just the tool:

```
MOD (INPUT(LEFT(KEY),PIBw.).
```

Generally, the wider is the informat width, the better, because the wider it is, the more key information is involved in the hashing process. However, selecting the informat too wide may result in a large integer rendering the result produced by MOD function incorrect. Experimentally, it has been found that under NT, the maximum allowable width, 8, works fine. Under OS/390, it should not exceed 7, and under HP-UNIX, 6 is the limit. The method has an extra advantage of avoiding the slow SUBSTR function, for it automatically chops the number of characters from the beginning of KEY equal to the informat width. Note that we use PIBw. instead of S370FPIBw.. First, it is faster. Secondly, with hashing, the order of bytes does not matter: We only want to use as many key bytes as possible to minimize collisions. The LEFT function may help by squeezing leading blanks to the right. If a key is longer than the practical informat width, the trick still works, provided that the input characters distinguish the keys well. However, if they have a good chance of being identical, they can be selected from a different portion of the key.

4. Composite Keys

This situation arises quite often. A natural inclination is to concatenate the components and hash the result. Principally, there is nothing wrong about it; however, there are two pitfalls. First, in the context of hashing, where computing a hash function fast is paramount, concatenation is slow. Second, the components may concatenate into an integer lying beyond SAS integer precision. Third, too large a value can cause the MOD function to return a no-sense result, for instance, a remainder greater than the divisor.

Consider a (real-life) situation when records are uniquely identified by two numeric variables, a 16-digit ID and 9-digit MEM, while particular ID can point to multiple accounts. Concatenating the keys as ID || MEM and hashing the result would have the effect of scrambling the entire MEM. All keys with the same ID would then hash to the same address regardless of MEM and lead to multiple collisions and horrible performance. Luckily, it can be avoided since we are not interested in the value of the key itself, but only in its remainder modulo HSIZE. Hence, Horner's algorithm can be used to hash the components separately and then combine the results in the final address. The outcome is the same as if we had enough integer precision to store the combined key accurately. For the ID and MEM, it means that the hash function can be computed in the form:

```
MOD(MOD(ID,HSIZE)*1E9 + MEM, HSIZE) .
```

If the partial keys are longer or the range is wider, they can be split further, and Horner's rule can be applied to the components once again. Of course, in order for this method to work, the parts of the key must be kept in parallel hash arrays, and loaded and tested separately. If, for instance, ID and MEM were hashed by chaining, the HKEY declaration would have to be replaced with

```
ARRAY HID  (0:&HSIZE) _TEMPORARY_;
ARRAY HMEM (0:&HSIZE) _TEMPORARY_;
```

The instruction

```
HKEY(H)=KEY
```

would become

```
HID  (H) = ID    ;
HMEM (H) = MEMNO ;
```

Also, in the TRAVERSE subroutine, the instruction

```
IF KEY=HKEY(H)
```

would transform into the following:

```
IF ID = HID(H) AND MEMNO = HMEM(H);
```

Similar modifications could be done if the open addressing methods were used.

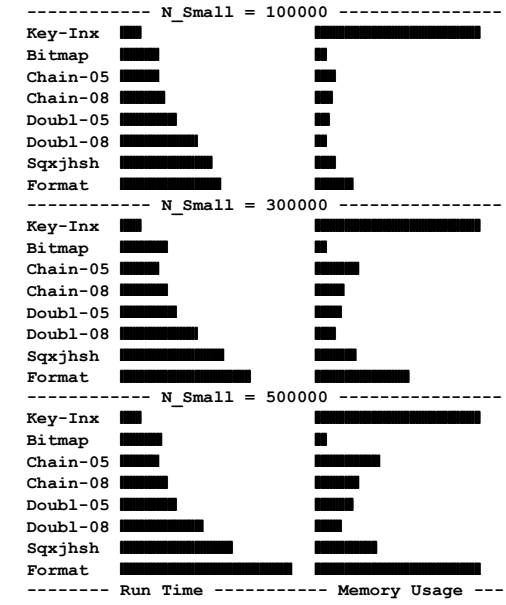
V. BENCHMARKING

Each technique presented above operates best in its own "area of expertise" defined by the number of lookup keys and key range. To compare them to each other and two SAS-supplied methods, SMALL and LARGE were created with random integer keys in [0:8E6] range where all methods could work within the system imposed memory limit of 70 MB. To include bitmapping into the comparison group, the satellite S_SAT was omitted from SMALL. The input was prepared in such a way that hits and misses were equally likely to occur. LARGE with fixed N_LARGE=2E6 was then matched against SMALL with varying number of records in batch on S/390 G5 R36 Enterprise Server running SAS Version 6.09E. For key-indexing, bitmapping, and hashing LOAD represents the time needed to load a table from SMALL (<EOF1> loop). In the case of formatting, LOAD is the time required to unduplicate SMALL and compile the format. For MERGE, it is the time needed to sort the files. The value LF= is the load factor used for the run. LOAD, SEARCH, and TOTAL are given in CPU seconds, MEMORY – in kilobytes.

Table 1. Benchmarking.

N_Small	Method	Load	Search	Run	Memory
100,000	Key-Inx	0.42	12.18	12.60	65261
	Bitmap	0.36	22.64	23.00	3925
	Chain-05	0.31	21.78	22.09	5997
	Chain-08	0.34	26.28	26.62	4829
	Doubl-05	0.28	32.78	33.06	4445
	Doubl-08	0.33	47.91	48.24	3961
	Sqxjhsh	0.00	52.66	52.66	5881
	Format	6.27	51.92	58.19	10866
	Merge	19.74	46.16	65.90	3276
300,000	Key-Inx	0.67	12.07	12.74	65261
	Bitmap	0.99	26.17	27.16	3925
	Chain-05	0.87	21.47	22.34	12093
	Chain-08	0.95	26.03	26.98	8637
	Doubl-05	0.80	31.24	32.04	7493
	Doubl-08	0.97	47.03	48.00	5769
	Sqxjhsh	0.00	58.38	58.38	11401
	Format	18.71	55.19	73.90	26199
	Merge	20.91	46.63	67.54	3267
500,000	Key-Inx	0.92	12.09	13.01	65261
	Bitmap	1.59	26.56	28.15	3925
	Chain-05	1.37	21.60	22.97	18033
	Chain-08	1.53	25.30	26.83	12357
	Doubl-05	1.29	31.90	33.19	10465
	Doubl-08	1.57	42.17	43.74	7625
	Sqxjhsh	0.00	64.49	64.49	16921
	Format	29.77	67.26	97.03	57289
	Merge	21.59	47.27	68.85	3267

The same benchmarking information might be digested better if presented in a more visual form. On the chart below, the left half of bars represents relative run-times, and the right half shows relative memory utilization.



Note that the run times exhibited by key-indexing, bitmapping, and hashing, in agreement with their direct-addressing nature, are virtually independent from the number of lookup keys. For key-indexing and bitmapping, memory usage is always fixed since the number of keys has no effect on the universe of keys they embrace. Hashing uses memory strictly proportional to the number of keys in the table and sparsity of the table.

VI. APPLICATIONS

1. Subsetting

Subsetting used above as a sample problem is an important but only one of many tasks to which direct-addressing based methods can be applied successfully. However, before discussing other applications, we have to make a few final observations about subsetting, all the more that it has been used as our proving grounds. From the test results, it follows that when it comes to one-time subsetting, direct-addressing methods result in lookup speeds unmatched even by methods written in the underlying software and specifically designed for searching. As an icing on the cake, hashing is significantly more memory-efficient than formatting and SQL. The latter is extremely important when the number of keys in SMALL grows beyond a couple of million. Hash memory is strictly proportional to the number of lookup keys and can be accurately estimated beforehand. Contrary to that, the amount of memory used by formats or SQL seems to grow uncontrollably after a certain threshold has been reached.

On a different note, we should exercise caution dragging satellites from SMALL through the memory. If there is more than one satellite, one may be tempted to create a separate parallel satellite array for each, but this is not always the right thing to do. Remember, character temporary SAS arrays are allocated in 8-byte multiples per item (unless you are running V8.1). If we have four 8-byte character satellites, a separate array can be declared as $8 for each with 100 per cent memory utilization. However, if we have four 2-byte satellites and create 4 parallel arrays $2 each, it will waste gobs of memory, for SAS will allocate the arrays with 8 bytes per item, anyway. So, in this case, we will be much better off memory-wise allocating one array as $8, stringing the satellites together in the load phase, and unstringing them into separate variables just prior to outputting a record.

2. Dynamic DATA Step Data Dictionaries

Let us take a look at key-indexing and hashing from a different, more philosophical, standpoint. The key-indexed and hash tables we have used to facilitate direct address and hybrid searching can be viewed as *some abstract data type* (*ADT*) in memory, that allows to efficiently perform certain operations on its *entries*. The ADT used in key-indexing and hashing is simply called *a table*. The entries contain keys and maybe some satellite information. There are two operations we have learned how to perform in the process of solving our sample problem: *Insert and search*. Many kinds of ADTs other than a hash and key-indexed table can facilitate these operations. A simple sequentially searched array, binary searched sorted array, AVL tree are just a few ADT examples.

The difference between various ADTs lies in the time necessary to insert an entry or search the entries given a key. For example, a plain array requires O(1), i.e. constant time, independent from the number of entries N, to insert a new entry - we simply append it to the right. However, searching such a structure occurs in O(N) time, i.e. proportional to N. If the ADT is a sorted array, we need O(N) time to insert an entry because it is necessary to shift a number of items proportional to N to free a node for the new key keeping the table sorted. In exchange, searching an ordered array, as we know from Part 1, occurs only in O(log(N)), or even O(loglog(N)) time. Yet another ADT, an AVL tree, facilitates both operations in O(log(N)) time as its worst case.

From these examples, it is clear what kind of advantage key-indexing and hashing offer: If a hash table is sparse enough, they support both insert and search operations in constant time O(1), because, as we have seen before, it takes practically the same time to search the table or to insert a new key, no matter how many keys the table may contain.

As a side note, from this standpoint, the difference between key-indexing and hashing is merely superficial. A key-indexed table is, in effect, nothing else but an infinitely sparse hash table, and the hash function used to access it is simply constant.

The fact that hashing supports searching (and thus retrieval and update) in constant time makes it ideal for implementing DATA step dynamic data dictionaries. Imagine that in the course of DATA step processing, we need to memorize certain key elements and their attributes as we go, and at different points in the program, ask and answer questions like the following:

1. Has the current key already been used before?
2. If it is new, how to insert it in the table, along with its attribute, in such a way that the question 1 could be answered as fast as possible in the future?
3. How to access a key element in the most speedy fashion and update its satellite datum?
4. If the key is no longer needed, how to delete it?

If the "key element" satisfies the conditions making key-indexing applicable (for instance, it is a SAS date), there is no better tool for the job. All the actions are performed in O(1) time and do not get any simpler:

1. See if the node whose value equals KEY contains a missing value.
2. Fill the node with the attribute.
3. Overwrite the attribute already in the node.
4. Move a missing value to the node.

If the keys are not limited-range integers, we will have to organize a hash table using either of the collision resolution policies given in the text. In both programs, the body of the first DO UNTIL(EOF) loop constitutes nothing else but a ready-to-go combined hash search-and-insertion. That answers questions 1 and 2, or 1 and 3. The second DO UNTIL(EOF) loop is a pure hash search, and answers question 1 itself.

A practical application of these principles immediately coming to mind is obtaining frequency counts in the case of a huge number of distinct levels of a categorical variable, when FREQ or SUMMARY either run out of memory or take too long to run. To compute frequencies without sorting, we must be able to maintain a table in memory allowing to immediately locate the value coming with the next record and add a unity to its count.

The following question was asked in SAS-L: "I have an unsorted SAS data set with almost 100 million records. It has a numeric variable FLDR_ID that can be any integer number from -500,000 to +500,000. How to create a file with frequencies, cumulative frequencies, percents and cumulative percents for all values of FLDR_ID having only 50 MB of RAM?" The problem with the "standard" approaches (FREQ or SUMMARY) is that there are too many discrete values of the categorical variable, and both procedures, if applied "head-on", either run out of memory or seem to run endlessly. From the standpoint of direct addressing, the key FLDR_ID is a restricted-range integer, and therefore for the purpose of the data dictionary, key-indexing should be here right at home. This was realized by Ian Whitlock and the author:

```
data freq (keep=fldr_id freq cfreq pcnt cpcnt);
   array f (-500001:500000) _temporary_;
   do until(end);
      set ids end=end nobs=nobs;
      if fldr_id = . then fldr_id = lbound(f);
      f(fldr_id) ++ 1;
   end;
   ptot = 1/nobs * 100;
   do i=lbound(f) to hbound(f);
      if f(i) = . then continue;
      freq =  f(i);
      cfreq ++ freq;
      pcnt  = freq * ptot;
      cpcnt ++ pcnt;
      if i > lbound(f) then fldr_id = i;
      else fldr_id = .;
      output;
   end;
run;
```

The program uses 12 MB of memory and runs an order of magnitude faster than either FREQ or SUMMARY (provided that they do not run out of memory in the process).

3. Stable Sortless Unduplication

While discussing hashing, we saw that as an attempt is made to load the next key into a hash table, the search-and-insert subroutine first determines whether the key has already been inserted, and if it has, goes to the next record. As this occurs very fast, the search-and-insert subroutine can be successfully used to remove duplicates from a file without sorting.

Speaking of the latter, for a SAS programmer, "duplicate removal" almost instantly rings "PROC SORT NODUPKEY" or "SELECT DISTINCT", depending on the prior exposure and taste preferences. It is an interesting phenomenon. We have, in effect, accustomed to using the side effects of two very time-consuming procedures just to kick out records with repeating keys. Of course, in the situation when a file has to be both sorted and unduplicated, PROC SORT is just the tool for the job. However, if sorting is not needed, a lot of extra work is done for no reason. What is more, consider a situation when not only we need to delete the duplicates from a file, but also retain the original order of its records, in other words, unduplicate the file in a *stable* manner. Should we decide to sort with NODUPKEY, we would be looking at at least 3 steps:

1. Add a sequence variable, say SEQ, to the file.
2. Sort the file with NODUPKEY EQUALS options by the key.
3. Re-sort the file by SEQ, and drop SEQ from the output.

Not only it does not look efficient, it does not make a whole lot of sense. Imagine that we have to remove duplicate cards from a deck; would we sort the deck first? Probably not! We would most likely take the cards off the deck one by one and memorize which cards have been taken out so far. If a card is "new", it goes face up to the output deck; if it is "old", it goes to the waste basket. At the end, the output will contain no duplicates and have the same relative order as input. All along in this process, we are using our human memory to keep track of the "keys" having been already used. Getting back to real files, a direct-address-based dictionary table can play the same role, providing both the quickest way to memorize "used" keys and establish whether the current key has already been used. Of course, the table must have a sufficient memory capacity, so we have to exercise a good judgement choosing between key-indexing, bitmapping, or hashing.

As an example, let us consider unduplicating a file similar to SMALL (how "small", depends on the range of keys and number of records) having 1,000,000 records, say. Assume that KEY has 16 digits, so neither key-indexing nor bitmapping can be used. However, a 50% sparse open-addressed hash table can be deployed at the expense of about 30 MB of memory. (It is not a small change, but with "usual" PC memories steadily creeping towards 1 GB, such memory usage can be considered tolerable.) Moreover, with 50% of nodes guaranteed to be empty, we can use linear probing, the simplest collision resolution method, with great deal of confidence. The plan (paralleling the playing card analogy above) is plain:

Step 1. Read a record from SMALL.
Step 2. Search for the key associated with the record in the hash table.
Step 3. If the key is found, it is a duplicate. Go to step 1.
Step 4. Otherwise insert the key in the table, output the record and go to step 1.

In the language of the SAS DATA step, it does not get any simpler, either:

```
** Sortless Stable Unduplication with Linear Probing;

data nodup (keep=key l_sat);
   array hkey (0:2000003) _temporary_;
   set large;
   do h=mod(key,2000003) by -1 until (hkey(h) = .);
      if h < 0 then h = 2000003;
      if hkey(h) = key then delete;
   end;
   hkey(h) = key;
run;
```

That is all it takes. Of course, the number 2000003 is not just arbitrary – it is the first prime number greater than 2000000, the "target" hash table size. But what about performance? On the same real computer the rest of the tests for this paper has been done, this step finishes the task in 3.1 CPU seconds. This compares quite favorably with PROC SORT EQUALS NODUPKEY (4.3 CPU seconds, and of course more for two extra steps if the stable output is required), and SQL with DISTINCT (11.2 CPU seconds).

4. Other Applications

It is impossible to embrace all conceivable applications of direct addressing methodology in one paper, so let us superficially mention just two more directions.

The author has participated in a "fuzzy matching" project, where the records from multi-million files with insufficient and redundant key information had to be linked using probabilistic matching. The linkage was essentially done in two stages. The first stage, using multiple composite redundant keys, identified probable matches, which were then scored pair-wise in the second stage. In both stages, key-indexing and hashing techniques were used to boost performance. They successfully supplanted "large" formats and SAS indexes, and as a result, the matching process was able to finish in about 1/5 of the original run-time on the same UNIX server where the original programs were run.

In this paper, only memory-resident direct-addressing methods have been considered. But what if we have so many distinct keys that none of the methods above will work just because of sheer memory limitations? Is it possible to apply the direct-addressing techniques, working so well in the high-speed memory, to some form of disk searching? The answer to this question is "yes". In fact, using a hybrid disk/memory hashing methodology, a plain SAS data set can be organized in such a way that the speed of accessing it randomly will exceed that of SAS index several times. Moreover, because of the intrinsic properties of hashing, the performance of such a lookup table does not depend on the distribution of the search keys. However, it is a topic for another paper.

CONCLUSION

Key-indexing is an in-memory lookup technique based strictly on direct addressing into an array with no comparisons between keys made. Its area of applicability is limited to integer keys falling in a limited range defined by available memory resources. However, when applicable, key-indexed search exhibits unmatched performance, and is the most straightforward way of implementing an ADT where all operations, such as search, insert, retrieve, update, delete, and enumerate are done in constant, O(1), time.

Bitmapping does not deviate a bit from the key-indexing philosophy, but uses available memory resources smarter by indexing keys directly into the bits, rather than 8-byte elements, of a numeric array. This way, bitmapping can address a much larger universe of integer keys than pure key-indexing. Both techniques have the advantage of working very fast with unlimited number of keys falling into their workable range. For instance, for keys restricted to 8 digits, up to 100 million integer keys can be in effect "stored" and subsequently extremely rapidly searched in a bitmap occupying only about 12 MB of real storage (RAM).

Hashing helps direct addressing work on keys of any type and range by bringing serial search and collision resolution policies into the equation. A bit slower that pure direct addressing, hashing searches times faster than SAS formats and SQL, and uses significantly less memory. Massive data processing applications like a data warehouse or production list management system are examples of the fields where the unmatched speed and efficiency of direct-addressing methods can be utilized. Compared to "traditional" techniques, they can successfully supplant formats and SQL in eliminating costly table joins, and tremendously accelerate the processes of data extraction, scrubbing, and validation, based on a large predetermined set of keys. The larger the data, the bigger advantage direct addressing can offer. Finally, direct-addressing searching methods are just additional, free programming tools, and can be used by any SAS programmer interested in efficiency and performance.

Key-indexing, bitmapping, and hashing are cool. They allow operating in the niches where "standard" approaches may run out of memory or take a frustrating time to run. The author encourages other SAS users to use these tools, modify them, tweak them, improve the code, and discover new areas of application. Karsten M. Self wrote once after having tried hashing in a real-world application: "Hash rocks, Dude!" Needless to say, the author eagerly agrees.

REFERENCES

1. D. E.Knuth, *The Art of Computer Programming*, **2**.
2. D. E.Knuth, *The Art of Computer Programming*, **3**.
3. R. Sedgewick, Algorithms in C, Parts 1-4.
4. T. A. Standish. Data Structures, Algorithms and Software Principles in C.

ACKNOWLEDGEMENTS

Thanks to Karsten M. Self, Ian Whitlock, F. Joseph Kelley, Sigurd Hermansen, and Base SAS R&D team for their enthusiastic support of direct-addressing methods in SAS, valuable discussions full of ideas, wit, and vigor, and giving the author an opportunity to apply the techniques to solve practical problems. The author gratefully acknowledges the contribution of the individuals who have, directly or indirectly, encouraged the author and supported his efforts of making direct addressing an accepted and practically used DATA step philosophy:

Michael V. Dorfman	Paul Gorell	David Pider
Eugenia P. Kravchenko	Steven Kleiman	Robert Workman
Doris H. Bogar	Alex V. Martchenko	David Cassell
Victor P. Dorfman	Thomas Mendicino	Jim Groeneveld
Vera Voloshin	Alex L. Voloshin	Diana Noble
Koen Vyverman	Vladimir A. Kirillov	Gerard Pauline
Benjamin Guralnik	Yuri Katsnelson	Ray Pass
Gennady Taratut	Michael A. Raithel	Art Carpenter
Jane King	Michael Rhoads	Shiling Zhang
Jay Melesky	Dianne Rhodes	Ronald J. Fehd
Bob Abelson	Don Stanley	Thomas Zicafoose
Ashiru Babatunde	Mark Terjeson	Christoph Edel
Peter Crawford	William W. Viergever	John Whittington
Colin Earle	Gregg Snell	Paul Kent
Peter Lund	Rick Aster	Kathy Y. Knorozova
Viacheslav V. Tsiolko	Igor A. Soloshenko	Michael M. Begun

AUTHOR CONTACT INFORMATION

Paul M. Dorfman
10023 Belle Rive Blvd. 817, Jacksonville, FL 32256
(904) 564-1931 (h) / (904) 954-8533 (o)
sashole@bellsouth.net
paul.dorfman@citicorp.com
paul_dorfman@hotmail.com

APPENDIX

Table A1. Inserting the sample keys into a hash table with collision resolution using coalesced linked list chaining.

H -->	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	
HKEY	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	**185**	.	KEY=185
LINK	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	**00**	.	
HKEY	.	.	.	**971**	.	.	.	.	.	.	.	.	.	.	.	.	185	.	KEY=971
LINK	.	.	.	**00**	.	.	.	.	.	.	.	.	.	.	.	.	00	.	
HKEY	.	.	.	971	.	.	.	.	.	**400**	.	.	.	.	.	.	185	.	KEY=400
LINK	.	.	.	00	.	.	.	.	.	**00**	.	.	.	.	.	.	00	.	
HKEY	.	.	.	971	.	.	**260**	.	.	400	.	.	.	.	.	.	185	.	KEY=260
LINK	.	.	.	00	.	.	**00**	.	.	00	.	.	.	.	.	.	00	.	
HKEY	.	.	.	971	.	**922**	260	.	.	400	.	.	.	.	.	.	185	.	KEY=922
LINK	.	.	.	00	.	**00**	00	.	.	00	.	.	.	.	.	.	00	.	
HKEY	.	.	**970**	971	.	922	260	.	.	400	.	.	.	.	.	.	185	.	KEY=970
LINK	.	.	**00**	00	.	00	00	.	.	00	.	.	.	.	.	.	00	.	
HKEY	.	.	970	971	.	922	260	.	.	400	.	.	.	.	.	.	185	**543**	KEY=543
LINK	.	.	00	00	.	00	00	.	.	00	.	.	.	.	.	.	00	**00**	
HKEY	.	.	970	971	.	922	**260**	.	.	400	.	.	.	.	**532**	185	543		KEY=532
LINK	.	.	00	00	.	00	**15**	.	.	00	.	.	.	.	**00**	00	00		Collision @ 06
HKEY	.	.	970	971	.	922	260	.	.	400	.	.	.	**050**	532	185	**543**		KEY=050
LINK	.	.	00	00	.	00	15	.	.	00	.	.	.	**00**	00	00	**14**		Collision @ 17
HKEY	.	.	970	971	.	922	260	.	.	400	.	.	**067**	**050**	532	185	**543**		KEY=067
LINK	.	.	00	00	.	00	15	.	.	00	.	.	**00**	**13**	00	00	**14**		Collision @ 17

Table A2. Inserting the sample keys into a hash table with collision resolution by open addressing with linear probing.

H --→	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	
KEY=185	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	**185**	.	.	
971	.	.	**971**	.	.	.	.	.	.	.	.	.	.	.	.	185	.	.	
400	.	.	971	.	.	.	.	.	.	**400**	.	.	.	.	.	185	.	.	
260	.	.	971	.	.	**260**	.	.	.	400	.	.	.	.	.	185	.	.	
922	.	.	971	.	**922**	260	.	.	.	400	.	.	.	.	.	185	.	.	
970	.	**970**	971	.	922	260	.	.	.	400	.	.	.	.	.	185	.	.	
543	.	970	971	.	922	260	.	.	.	400	.	.	.	.	.	185	**543**	.	
532	.	970	971	**532**	922	260	.	.	.	400	.	.	.	.	.	185	543	.	Collision @ 05
050	.	970	971	532	922	260	.	.	.	400	.	.	.	.	**050**	185	543	.	Collision @ 16
067	.	970	971	532	922	260	.	.	.	400	.	.	.	**067**	050	185	543	.	Collision @ 16

Table A3. Inserting the sample keys into a hash table with collision resolution by open addressing with double-hashing.

H --→	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	
KEY=185	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	**185**	.	.	
971	.	.	**971**	.	.	.	.	.	.	.	.	.	.	.	.	185	.	.	
400	.	.	971	.	.	.	.	.	.	**400**	.	.	.	.	.	185	.	.	
260	.	.	971	.	.	**260**	.	.	.	400	.	.	.	.	.	185	.	.	
922	.	.	971	.	**922**	260	.	.	.	400	.	.	.	.	.	185	.	.	
970	.	**970**	971	.	922	260	.	.	.	400	.	.	.	.	.	185	.	.	
543	.	970	971	.	922	260	.	.	.	400	.	.	.	.	.	185	**543**	.	
532	.	970	971	.	922	260	.	.	.	400	.	.	.	.	**532**	185	543	.	Collision @ 05; C=8
050	.	970	971	.	922	260	.	.	.	400	**050**	.	.	.	532	185	543	.	Collision @ 16; C=6
067	.	970	971	.	922	260	.	.	**067**	400	050	.	.	.	532	185	543	.	Collision @ 16; C=8

Advanced Methods to Introduce External Data into the SAS® System

Andrew T. Kuligowski – Nielsen Media Research

ABSTRACT / INTRODUCTION

The SAS® System has numerous capabilities to store, analyze, report, and present data. However, those features are useless unless that data are stored in or can be accessed by the SAS System. This presentation will provide a brief introduction to many of the different methods that can be used to pass data into the SAS System, with special emphasis on those methods for use on the Personal Computer. Topics will include the menu-driven SAS Import Wizard, DDE, ODBC, and others - as many as time and space will allow! The goal of this presentation is to provide information that will be useful to all users of the SAS System. Some topics are tailored to the novice, while others will be more applicable to the experienced user. Please note that some information, by its very nature, will only be applicable to select operating systems. This presentation will be tailored to Version 8 of the SAS System, however special note will be made of those topics that have changed between Versions 6 and 8.

SAS IMPORT WIZARD

Often, the best tool for a job is the simplest one. To illustrate that point, the first topic we will discuss is the *SAS Import Wizard.* Available since Release 6.12 of the SAS System, the SAS Import Wizard is a menu-driven system to define and import external data into the SAS System. It is typical in appearance and in approach to Wizards available in other Windows products, such as Microsoft Excel. The Import Wizard is accessed by selecting **Import Data** from the **File** pulldown menu on the SAS toolbar. (Those who have not yet upgraded to Version 8 will find the selection is called **Import** under Version 6.12.)

The first screen requires the user to specify the type of file to be imported. The first choice is a "standard file format"; a pull-down menu allows various options such as dBASE, LOTUS, Excel, or delimited files. (NOTE: Some of these options are only available if the site has licensed SAS/ACCESS to PC File Formats.) The other choice, "user-defined file format", provides an interface to the *External File Interface.* (The External File Interface, which permits the user to specify the details of an external file via menus, will not be discussed in this presentation.) **See Figure A for a sample of this screen – this and subsequent screen prints will illustrate the IMPORT of an Excel 2000 file.**

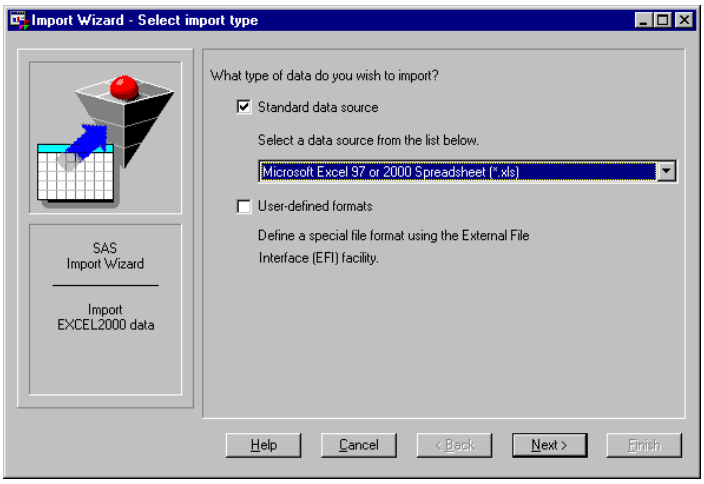


Figure A – SAS Import Wizard "Import Type" Screen

The SAS Import Wizard now displays a screen prompting the user to "Select File". The file name can be manually typed in the space provided, or the *Browse* button can be selected in order to search for the file. **See Figure B for a sample of this screen.**

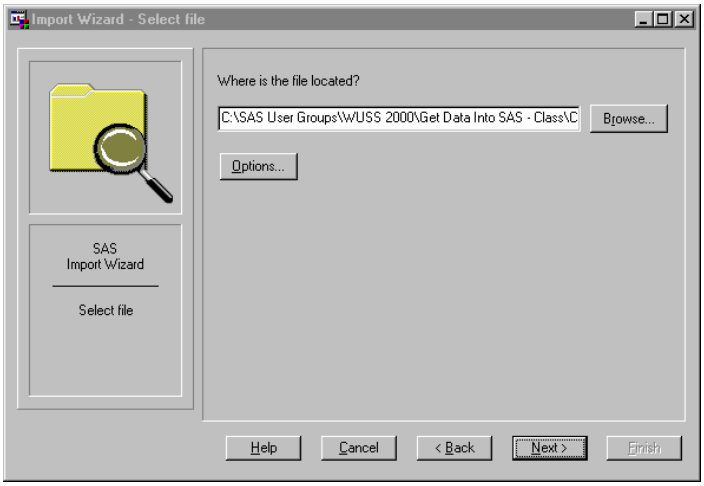


Figure B - SAS Import Wizard "Select File" Screen

This screen also has an Options button. This button causes a separate pop-up window to be displayed, which can be used to select any options that are specific to the type of file selected on the first screen. **See Figure C for a sample of this screen.**

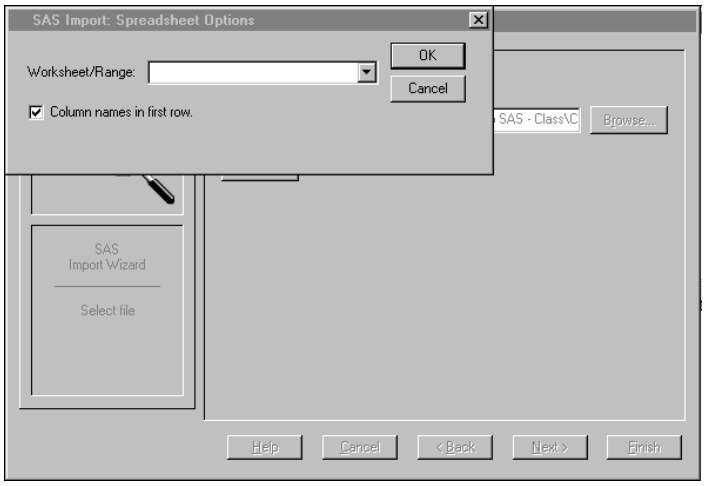


Figure C – SAS Import Wizard "Select File – Options" Screen

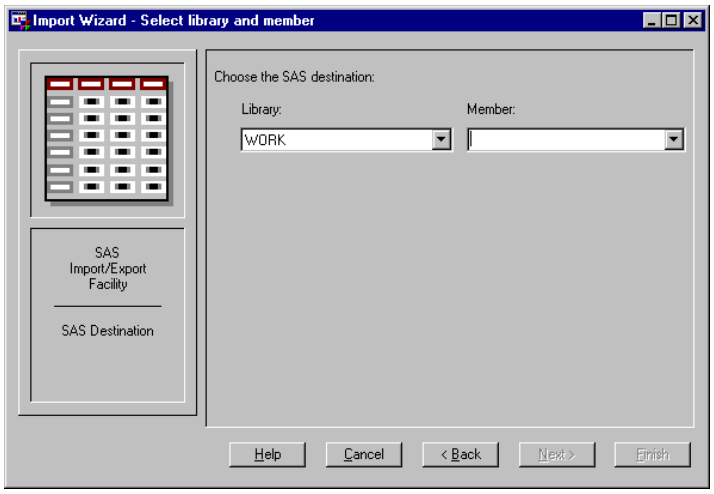


Figure D – SAS Import Wizard "Library & Member" Screen (Initial)

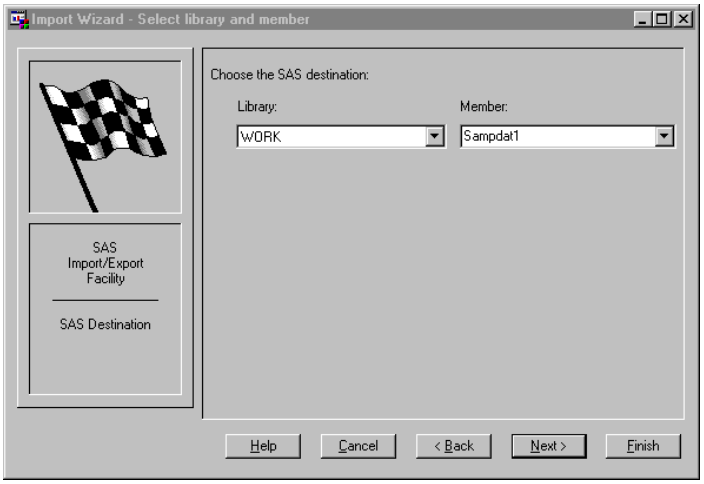


Figure E – SAS Import Wizard "Library & Member" Screen (Completed)

Next, the user is prompted to choose the "SAS Destination", better known as the Library and Member. Again, the user can manually enter the SAS library and member into which the data is to be stored, or they can use pull-down menus to select from those which are already known to the SAS session. **See Figures D and E for samples of this screen.**

If "standard file format" was originally selected and the screen was correctly and completely filled out, please note that the picture on the left side of the screen changes from a tabular image to a checkered flag when entry is finished. Normally, a checkered flag alludes to a finish line or completion – in fact under Version 6.12, this *was* the final screen of the SAS Import Wizard. However, under Version 8, an additional screen allows the user to store the generated code in a file, allowing for reuse and/or modification. **See Figure F for an example of this screen.**

At this point, the SAS System processes the request. Success is indicated by a simple message in the SASLOG, advising that the SAS dataset is now available for use :

```
NOTE: [SASdsn] was successfully
      created
```

At this point, it is possible for those users who are still using Version 6.12 of the SAS System to save their generated code, although the process is slightly more complicated than entering in values on a menu as with Version 8. The process to save generated SAS code under Version 6.12 is as follows:

- Toggle to the PROGRAM window in SAS Display Manager.
- RECALL (F4 by default) your previously submitted code.

The generated code will be echoed in the Program Window, and it can now be saved to an external file.

It should be noted that the code generated under Version 6.12 will look radically different than the code generated under Version 8. This is because Version 8 uses **PROC IMPORT**, which was not available under Version 6.12. **Refer to Figures G and H for examples of code generated under both versions of SAS.**

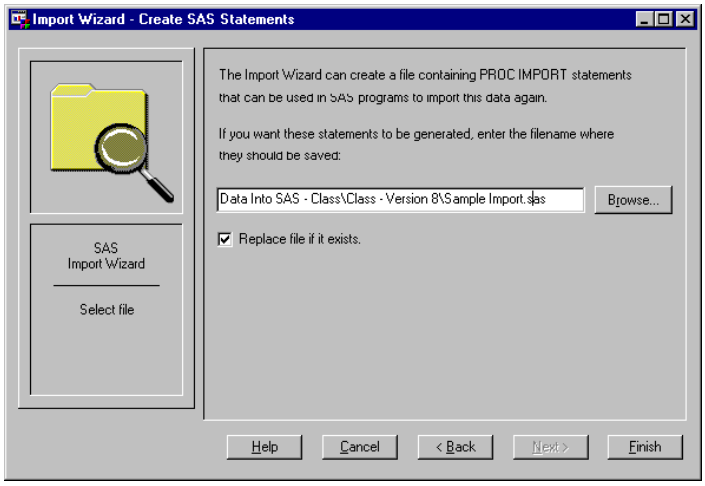


Figure F – SAS Import Wizard "Create SAS Statements" Screen

```
PROC ACCESS DBMS=EXCEL;
  CREATE WORK._IMEX_.ACCESS;
  PATH='C:\SAS Class\Sample
        Data\NHL 1999-2000 Scoring
        Leaders.xls';
  GETNAMES YES;
  SCANTYPE=YES;
  CREATE WORK._IMEX_.VIEW;
  SELECT ALL;
RUN;
DATA WORK.tempdata;
  SET WORK._IMEX_;
RUN;
```

Figure G – SAS Import Wizard Generated Code – Version 6.12

```
PROC IMPORT OUT=WORK.Sampdat1
     DATAFILE="C:\SAS Class\Sample
        Data\NHL 1999-2000 Scoring
        Leaders.xls"
     DBMS=EXCEL2000 REPLACE;
  GETNAMES=YES;
RUN;
```

Figure H – SAS Import Wizard Generated Code – Version 8

The SAS Import Wizard is not always the optimal solution - if it was, this would be an awfully short presentation! There are two major drawbacks to the Import Wizard. First of all, it is not available when running SAS in a mainframe environment. Secondly, it is an interactive tool, which renders it useless for a batch, schedule-oriented environment. However, the SAS Import Wizard can be the easiest, quickest, and therefore best solution for the one-time-only processing of an external file during an interactive SAS session. It also has great utility as a code generator, saving time for the user to invest in other areas.

PROC IMPORT

As mentioned earlier, PROC IMPORT (along with the corresponding PROC EXPORT) is a new addition to the SAS System. It provides a simple-to-code method that facilitates the transfer of data from an external source to the SAS System.

It is not necessary to understand the syntax of PROC IMPORT if the user generates the code from the Import Wizard – in general, wizards are designed so that the end user does not need any advanced knowledge of the tool in question. However, since PROC IMPORT can be invoked manually, either from scratch or by including code that was generated by an earlier execution of the Import Wizard – with or without further modification – it is worth briefly reviewing the procedure at this time.

There are two required arguments on the PROC IMPORT statement. The first, **OUT=**, should be familiar to all but the most inexperienced SAS users. The other required "argument" can actually be one of two different arguments, depending on the source of the input data:

- **DATAFILE="filename"** is used to specify most input data sources, such as sequential files or Excel spreadsheets.
- **TABLE="tablename"** is used to specify DBMS tables, such as from Microsoft Access.

There are also two optional arguments. **DBMS=** specifies the type of data to be imported. It is not required in conjunction with DATAFILE=, as long as the filename contains a valid extension associated with the data source such as .XLS. However, it *is* required if the file name does not have a valid extension or if TABLE= was specified instead of DATAFILE=. The other optional argument, **REPLACE**, controls whether or not a preexisting SAS dataset is overwritten by the procedure.

There are a number of statements available for use in conjunction with PROC IMPORT, depending on the data source that is being processed and on whether DATAFILE= or TABLE= is being used on the PROC statement. To conserve space, these options will not be discussed in this paper – the reader is directed to the Version 8 *SAS Procedures Guide* for details.

As with the Import Wizard, the options available to PROC IMPORT are limited to .TXT and .CSV

(or other delimited file) if the users' site has not licensed SAS/ACCESS to PC File Formats.

CSV (Comma Separated Value) FILES

The *CSV,* or *Comma Separated Value* File is a special variety of sequential file, typically used for importing or exporting data from a spreadsheet. Data values are separated by commas, as is implied by the name, and character values are typically surrounded by double quotation marks (").

CSV files can be processed by using the DSD parameter on the INFILE statement. This parameter automatically sets the default delimiter to comma, although this can be overridden by use of the **DELIMITER=** option. The presence of a pair of commas denotes a missing value. The DSD parameter also causes SAS to strip the double quotation marks, if present, from character values before storing them in SAS variables. Please note that character variables are defined with a default length of 8 bytes in this instance. This default length can be overridden by use of the **LENGTH** statement. Do not attempt to specify a format length on the input statement for character variables, as this may cause delimiting commas to be treated as part of the variable's value. **See Figure I for an example of reading a CSV file.**

```
DATA TEMP;
   LENGTH  CITY $ 20.  STATE $ 15. ;
   INFILE  SAMPCSV  DSD ;
   INPUT   YEAR     CONFNAME $
           CITY $   STATE    $;
RUN;
```

Figure I - CSV File

DDE

The next method of obtaining external data that shall be discussed in this presentation is *Dynamic Data Exchange*. Dynamic Data Exchange, or DDE, allows a client application to request information from a server application in a Windows or OS/2 environment. Effective with Release 6.08, the SAS System acts as a client application in this relationship. It can request data from a server application, with the requirement that the server application must be running. (It can also send commands and data to a server application, but that is a topic for another presentation.)

In order to use DDE, a connection must be established between the client application and the server application. This is accomplished by issuing a FILENAME statement with the keyword "DDE". The syntax for this statement, in this context, is:

```
FILENAME fileref DDE 'DDE-triplet' ;
```

The DDE-triplet is a specialized argument, and is made up of three components:

```
application|topic!item
```

Application is the name of the server application, such as Excel. *Topic* is defined as the "topic of conversation"; basically, this is the file to be processed. *Item* is the "item of conversation"; in a spreadsheet, this is the range of cells that is to be included. For example, the DDE-triplet an Excel worksheet would be:

```
Excel|[Book1]Sheet1!R1C1:R250C4
```

Note that the application and topic are separated by a vertical bar (|), while the topic and item are separated by an explanation point (!).

The DDE triplet for an application should be defined in the documentation for that application. However, most people find it easier to let SAS determine the proper DDE triplet. The following is a step-by-step method to obtain the proper DDE triplet for an application, assuming both SAS and that application are active:

- Toggle to your application, and use the standard PC "cut" techniques to store the portion of the client application to be processed in the Windows Clipboard. (For example, use the mouse to highlight the area to be "cut", then select CUT or COPY on the EDIT pop-up menu of most Windows applications.)
- Toggle to your SAS session, and click on the "Options" menu on the Menu Bar in SAS.
- The Options menu will contain an option called "DDE Triplet". Click on it.
- This will display an Information Box, which will contain the DDE-triplet.
- Enter this DDE-triplet into the FILENAME statement of your SAS routine.

If the user is willing to perform a little manual intervention, it is even possible to use DDE without ever knowing the name of the DDE triplet!

- As above, toggle to your application, and use the standard PC "cut" techniques to store the portion of the client application to be processed in the Windows Clipboard.
- Toggle to your SAS session, and replace the FILENAME statement with the following:

```
FILENAME fileref DDE CLIPBOARD ;
```

The SAS routine is now ready to be executed. The weakness in this approach is that the data to be processed must be stored in the Clipboard prior to each invocation of your SAS routine. The benefit is

that there is no need to ever know the DDE-triplet for your application to use DDE. (Please note that this approach will only work if an application is DDE compliant.)

In order to use DDE with the SAS System, the server application must be running while SAS is running. If the server application is not active, then it can be invoked from within the SAS session with the "X" command. However, the SAS options XSYNC and XWAIT must be turned off before issuing this command, or control will not be returned to the SAS session until that external application is closed -- this defeats the purpose of a DDE link! (Of course, the user could also simply toggle over to the Windows Program Manager and manually invoke the application.)

The actual transfer of data from the external application to the SAS System is done via the combination of an **INFILE** and **INPUT** statement. The actual code to accomplish this task looks exactly like the code to read a sequential file into SAS. **See Figure J for an example of reading an Excel 5.0 spreadsheet via SAS.** For further examples covering a number of PC products, please refer to "Technical Support Document #325 - The SAS System and DDE", which is available on the SAS Institute web site.

```
OPTIONS  NOXSYNC NOXWAIT;
X 'C:\EXCEL SASCONF.XLS' ;
FILENAME SASCONF DDE
   'Excel|[Book1]Sheet1!R1C1:R250C4';
DATA CONFSCHD;
   INFILE SASCONF;
   INPUT DAY TIME TITLE AUTHOR;
RUN;
```

Figure J - INPUT from EXCEL using DDE

It should be noted that the use and support of DDE is declining, as more recent technological advances become the tools of choice in the new millennium. However, the reader is encouraged to develop an understanding of DDE for several reasons:

- New DDE applications are still being coded at many sites around the world.
- There are many existing applications that were coded to use DDE; these applications will need maintenance.
- DDE is available with base SAS, and does not require additional product licensing or installation, making it a viable alternative for many sites.

SAS/ACCESS ENGINES

SAS/ACCESS software is available, under separate licenses, for a variety of host systems, covering traditional mainframe, personal computer, and UNIX environments. It provides a method to view and transfer data from several common database management systems (DBMS) and a number of common PC file formats, into the SAS system.

The ACCESS procedure can create *descriptor* files that will provide information about the data stored in the DBMS table or PC file format, and use that information to create a SAS data file. In addition, use of an *interface view engine* will allow SAS to read data from the file formats directly into SAS routines. The interface view engine is used by the SAS SQL procedure to directly access external databases without leaving the SAS session. However, the SQL statements used in the procedure are beyond the scope of this Tutorial.

There are two types of descriptor files created by the ACCESS procedure: an *access descriptor* and a *view descriptor*. Access descriptors provide information regarding the structure of the file to be accessed. This includes data types, table names, and column names, as well as the related SAS dataset information such as variable names and formats. This access descriptor can then be used to create the view descriptor, which will contain criteria to be used to select columns and rows from the selected DBMS table or PC file. The data can be used directly from the view descriptor in the SAS routine, or it can be extracted from the DBMS or PC file into a SAS data file.

The type of DBMS or PC file to be used is specified in the PROC ACCESS statement in the form:

PROC ACCESS DBMS=*filetype*

Filetype can take on many different values. To cite just a few examples, the user can obtain data from DB2®, SYBASE®, and ORACLE® by selecting filetypes *DB2*, *SYBASE*, and *ORACLE*, respectively. In a Windows environment, *XLS*, and *WKn (*where *n* is a valid version number) will allow the transfer of data from Excel or Lotus® spreadsheets, while *DBF* and *DIF* are obviously the filetype for interfacing with .DBF and DIF formatted files, respectively.

The actual access or view descriptor is then created with the following syntax:

CREATE libref.member-name.ACCESS or
CREATE libref.member-name.VIEW

The PROC ACCESS and CREATE statements are followed by a statement that identifies the name of the DBMS, DBF, XLS or other file that will be accessed. In addition, there are other editing statements; these provide information about the structure of the DBMS or PC file being accessed, and select columns to be viewed. **See Figure K for an example of using PROC ACCESS create a view, with a subsequent use of its output.**

```
LIBNAME VWLIB 'c:\confdat\';
PROC ACCESS DBMS=xls;
  CREATE vwlib.states.access;
  PATH 'c:\confdat\state.xls';
     <editing statements omitted>
  CREATE vwlib.states2.view;
     <select, format, and
      subset statements omitted>
RUN;

DATA _NULL_;
  SET vwlib.states2;
  <statements omitted>
RUN;
```

Figure "K" - PROC ACCESS : Creation and use of a View

The SAS view descriptor can be used in any PROC or DATA step just like a SAS data set. It is also possible to use PROC ACCESS to create a SAS dataset from the view descriptor. This is accomplished by issuing PROC ACCESS with the VIEWDESC=*libref.view-descriptor* and OUT=*libref.sas-data-filename* options. **See Figure L for an example of using PROC ACCESS create a view, with a subsequent use of its output.**

```
PROC ACCESS VIEWDESC=vwlib.states2
            OUT=vwlib.stdata ;
RUN;

PROC PRINT DATA=vwlib.stdata;
RUN;
```

Figure "L" - PROC ACCESS : Creation of a SAS Dataset

The process to create and use a SAS/ACCESS view, as described above, is reasonably easy to understand and to use. However, under Version 8, SAS/ACCESS became even easier to employ, by allowing view definitions via the LIBNAME statement! The syntax is straightforward:

```
LIBNAME  libref  enginename
  <engine-specific options>
  <general LIBNAME options>;
```

This process will be described under the ODBC section of this presentation.

It is not possible to fully cover PROC ACCESS in the limited space of this paper -- there are a number of separate manuals dedicated to the topic! For further information, including details of the DBLOAD procedure that will transfer data from SAS to the assorted DBMS and PC products, the reader is directed to the assorted SAS/ACCESS manuals.

PROC DIF and PROC DBF

SAS/ACCESS to PC File Formats, which requires a separate license from Base SAS software, also provides additional interfaces to two traditional types of PC files. *DBF* and *DIF* formats date back to the "ancient" days of PCs in the early 1980s; however, data continues to exist in these formats and one must be prepared to deal with it.

DBF files contain data originally formatted for the dBASE™ database management product, and are accessed through PROC DBF. PROC DBF accepts two options. The first, OUT= should be self-explanatory to anyone with even a modest experience in use of the SAS System. The other, DBn=, is required, and contains the file reference to the DBF file. The *n* in "DBn" refers to the version of dBASE for which the file was created; "2", "3", "4", and "5" are valid version numbers. Please note that all dBASE formatted files are assigned the extension .DBF, regardless of the version of dBASE (or other product) under which they were created. SAS will produce an error message if the version number is incompatible with the file format.

PROC DBF will produce a SAS dataset, with variable names mapped from the original dBASE field names. Field names will be truncated to 8 characters, if necessary, to conform to current SAS variable name restrictions. **See Figure M for examples of PROC DBF.**

```
10   /* This is a dBASE IV format file. */
11   filename sasconf
        'c:\sasconf\confloc.dbf';
12   proc dbf db2=sasconf  out=conf_db2;
RUN;

ERROR: Input file is not a DBASEII file.
WARNING: Data set WORK.CONF_DB2 not replaced
         because new file is incomplete.
NOTE: The SAS System stopped processing this
      step because of errors.
NOTE: The PROCEDURE DBF used 0.44 seconds.

13   proc dbf db4=sasconf  out=conf_db4;
RUN;

NOTE: 7 observations written to the output
      SAS data set.
NOTE: The PROCEDURE DBF used 0.38 seconds.
```

Figure "M" - PROC DBF - Invalid and Correct

DIF files, short for Data Interchange Format, date back to VisiCalc™ and other early PC spreadsheet products. PROC DIF can be used to convert DIF files into SAS datasets. The format for PROC DIF is similar to PROC DBF, with the DIF= option used in place of DBn=. Variable names are assigned COL1 to COLn in the output SAS dataset; it is recommended that the user subsequently reassign them to something more meaningful. . **See Figure N for an example of PROC DIF.**

```
filename sasconf 'c:\sasconf\confloc.dif';
proc dif dif=sasconf  out=conf_dif;
run;
proc print uniform;
run;
```

Figure "N" - PROC DIF

ODBC

Open DataBase Connectivity, or *ODBC*, started off as a standard for the exchange of data between DataBase Management Systems (DBMS) under Microsoft's Windows environment. Since those early days, interfaces to other operating systems and machines, such as the Apple MacIntosh, have been developed. It is necessary to use the SAS/ACCESS Interface to ODBC in order to use ODBC to bring data into the SAS System.

It is important to note that the SAS / ODBC interface does not directly obtain data from an external source, unlike other data sources which are available via SAS/ACCESS. Instead, it interfaces with the ODBC manager, which in turn interfaces with the other external data sources.

In order to use the SAS / ODBC Interface, it is therefore first required to install the appropriate drivers, or "*Data Sources*".. This is done via the ODBC Icon available through the operating system. (Under Windows 98, for example, this could be found in the "ODBC Administrator" from the Program Menu, or the "ODBC Data Sources" icon in the Control Panel.) **See Figures O, P, Q, R, and S for examples of the screens used to define a SAS driver in ODBC with the ODBC Data Sources method. The ODBC Administrator uses the same basic menu s, although the first 2 menus are different.**

It should be noted that under Version 6, the SAS ODBC driver used DDE for product to product communication. As of Version 7, the SAS ODBC driver uses TCP/IP protocol for communication. Version 6 users will need to update their driver settings when upgrading to Version 8.

The SAS / ODBC interface is also unlike the other products in the SAS/ACCESS family, in that PROC ACCESS is not used to bring external data into SAS. Instead, the SAS System utilizes an enhancement to **PROC SQL**, the *SQL Procedure Pass-Through Facility*. (Please note that it is still necessary to license and install SAS/ACCESS to ODBC in order to use the SQL Pass-Through Facility in conjunction with ODBC. It should also be noted that the SQL Pass-Through Facility can also be used with other databases via separate SAS/ACCESS licenses; these will not be discussed in this presentation.)

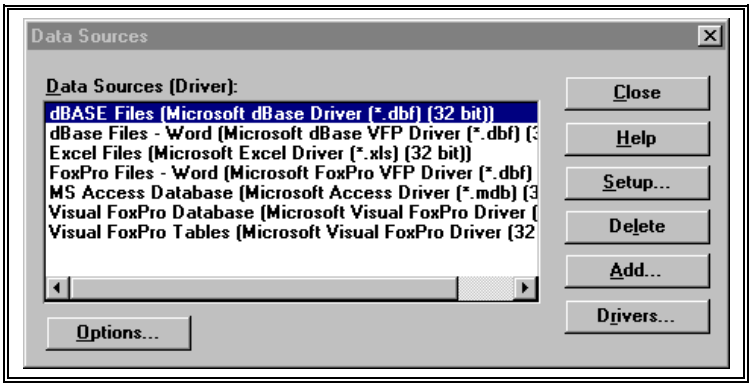


Figure "O" - ODBC Driver Configuration Active Data Sources

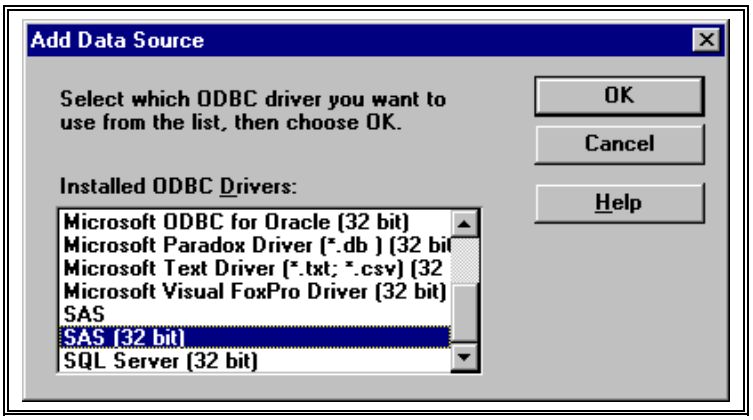


Figure "P" ODBC Driver Configuration - Add Data Source

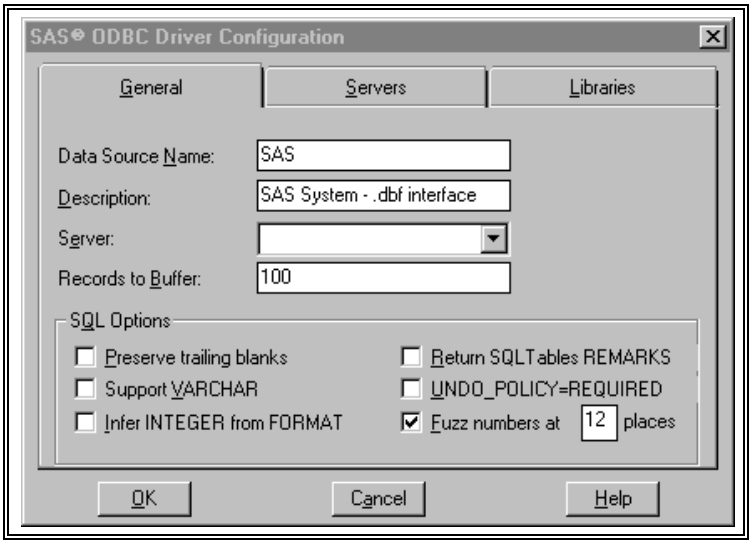


Figure "Q" - ODBC Driver Configuration - General Information

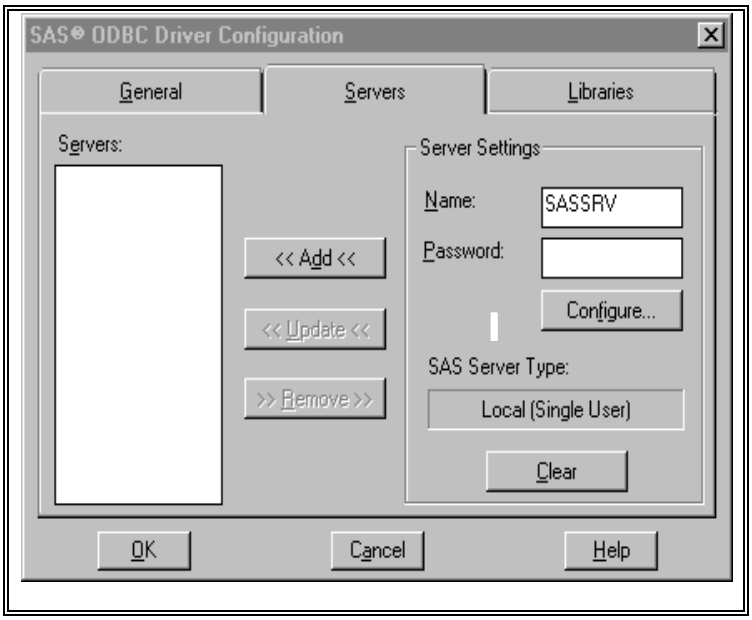


Figure "R" - Windows 3.1 ODBC Driver Configuration Servers (1 of 2)

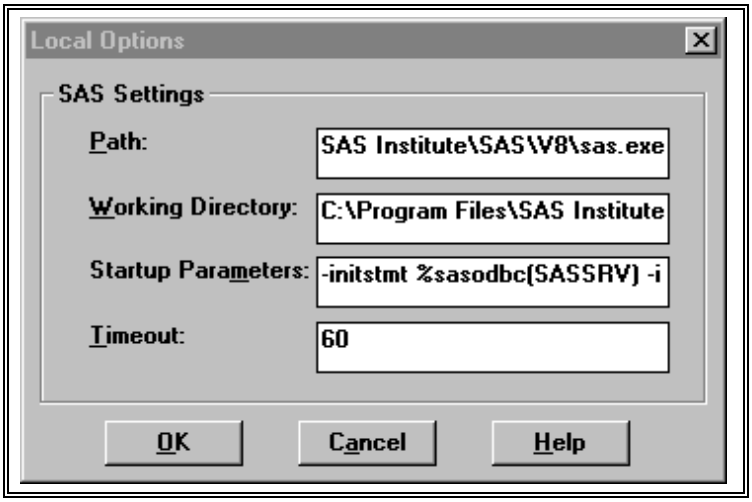


Figure "S" - Windows 3.1 ODBC Driver Configuration Servers (2 of 2)

The SQL Procedure Pass-Through Facility has four statements associated with it; or more correctly, three statements and a "component", which is included within the SQL itself.

The first statement, **CONNECT**, establishes a connection with ODBC. The syntax is:

```
CONNECT TO ODBC <AS alias> <(options)>
```

The alias is optional; however, if used, the word "AS" *must* immediately precede it. The options are used to specify the data source which is to be processed, and must be enclosed within parentheses. **DSN=***data-source* can be used to specify a previously defined data source, as described above. (Note that "DSN" stands for "Data Source Name", and not "Data Set Name".) Alternatively, the word **PROMPT** will walk the user through the process of setting up an ODBC interface at execution time. Note that PROMPT and DSN= are mutually exclusive of each other. **See Figures T and U for an example of the screens displayed to define a dBASE data source. See Figures V and W for the SAS source code to use these screens.** One other option that should be noted is **LOG**, which will cause all warnings and errors generated by the ODBC API (*Application Programming Interface)* to be written to the SASLOG. This may be especially useful during the development phase of an application.

As one would expect, **DISCONNECT** is the opposite of CONNECT. This statement ends the connection with ODBC. The syntax is :

```
DISCONNECT FROM ODBC | alias;
```

using either the word "ODBC" or the alias that was specified in the CONNECT statement. It is optional, as an implicit DISCONNECT is issued when the **QUIT** statement ends PROC SQL. However, it may be desired to explicitly break the connection if more SQL statements are to be issued, or if a different CONNECT is desired.

The **EXECUTE** statement sends non-query SQL statements to ODBC. These are primarily used to update the external database, and as such are outside the scope of this presentation.

CONNECTION TO is a clause that can be inserted in an SQL **SELECT** statement to send an SQL query to the external data source accessed via ODBC. The syntax is:

```
FROM CONNECTION TO ODBC | alias
     (SQL to external source)
```

again using either the word "ODBC" or the alias that was specified in the CONNECT statement. There are a number of special queries that can be used in conjunction with ODBC; these will be omitted from this presentation due to space limitations.

All of these statements and components can be combined with the statements normally associated with PROC SQL to form a successful ODBC query. **See Figure V for an example of PROC SQL using PROMPT. See Figure W for an example of PROC SQL using DSN=.**

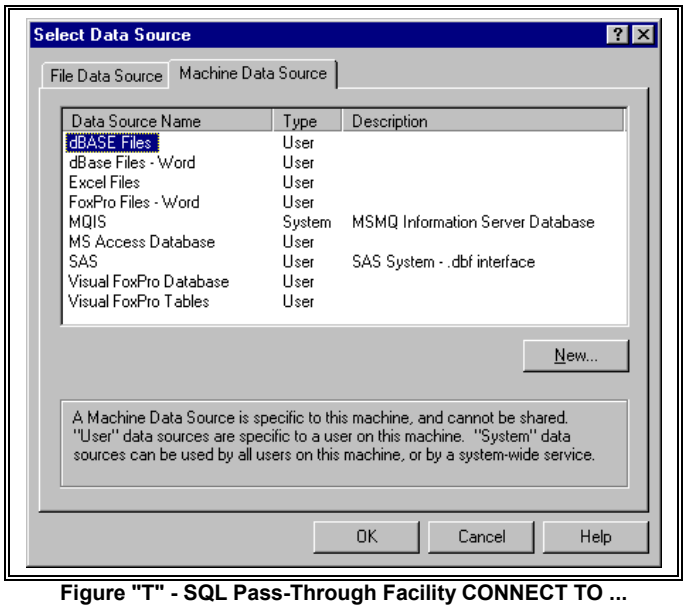


Figure "T" - SQL Pass-Through Facility CONNECT TO ... PROMPT / Data Source

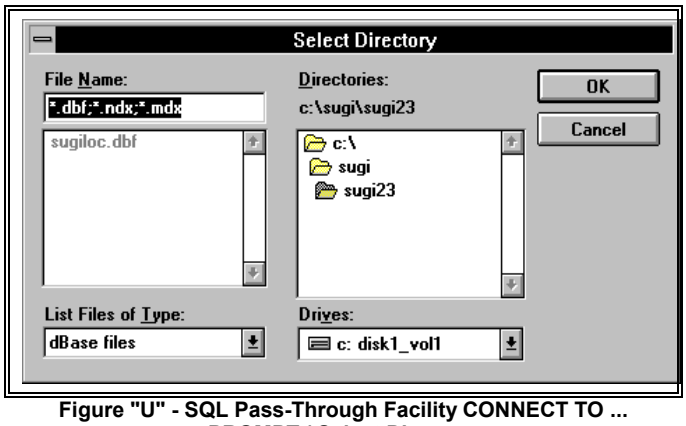


Figure "U" - SQL Pass-Through Facility CONNECT TO ... PROMPT / Select Directory

```
84   /* "feedback" option expands SQL */
85   /* statements in the SASLOG.     */
86   proc sql feedback ;
87      connect to odbc(prompt);
88      create table confloc as
89         select * from connection to odbc
90           (select * from confloc );
NOTE: Statement transforms to:
        select YEAR, CONFEREN, CITY, STATE
          from connection to ODBC
                 /* dbms=ODBC,
                connect options=(prompt) */
                  ( select * from confloc );
NOTE: Table WORK.CONFLOC created, with
      7 rows and 4 columns.
91      disconnect from odbc ;
92      quit ;
NOTE: The PROCEDURE SQL used 7.91 seconds.
```

Figure "V" - SQL Pass-Through Facility using PROMPT

```
218  proc sql ;
219     connect to odbc as sasconf
220              (dsn="confdemo");
221     create table confloc as
222       select * from connection
                  to sasconf
223         (select * from confloc );
NOTE: Table WORK.CONFLOC created, with
      7 rows and 4 columns.
224     disconnect from sasconf ;
225     quit ;
NOTE: The PROCEDURE SQL used 3.35 seconds.
```

Figure "W" - SQL Pass-Through Facility using DSN

The ODBC examples covered so far have been based on Version 6 technology, although they have been run using Version 8. However, under Version 8, it has become even easier for a SAS routine to interface with ODBC by using the LIBNAME statement.

By using the ODBC keyword, the LIBNAME statement can be used to provide a straightforward interface between the external data source and the SAS System:

```
LIBNAME  libref  ODBC
   <ODBC-specific options>
   <general LIBNAME options>;
```

See Figure X for an example of using the LIBNAME statement with ODBC.

```
245 libname odbclib odbc
246         datasrc='SASClass Example';
NOTE: Libref ODBCLIB was successfully
      assigned as follows:
      Engine:        ODBC
      Physical Name: SASClass Example

247 proc print data=odbclib.confloc; run;
NOTE: There were 10 observations read
      from the dataset ODBCLIB.CONFLOC.
NOTE: PROCEDURE PRINT used:
      real time           0.10 seconds
```

Figure "X" – LIBNAME statement with ODBC

CONCLUSION

There are a number of methods to introduce external data into the SAS System. It would be impossible to provide in-depth information on all of them in the limited space of this presentation. It is hoped that the material contained in this paper will serve to stimulate the curiosity of the reader, and that they will continue their education by researching the appropriate manuals and technical papers devoted to the specific topics discussed within this paper. Ultimately, however, it will be through real-life trial and error that true comprehension and retention of this knowledge will be attained.

REFERENCES / FOR FURTHER INFORMATION

Beatrous, Steve, and Clifford, Billy. (1998). "Sometimes You Get What You Want: I/O Enhancements for Version 7". *Proceedings of the Twenty-Third Annual SAS Users Group International Conference*. Cary, NC: SAS Institute, Inc.

Bodt, Mark (1996). "Talking to PC Applications Using Dynamic Data Exchange". *Observations*, Volume 5, No. 3 (Second Quarter 1996). Cary, NC: SAS Institute, Inc.

Boling, John C. (1997). "SAS Data Views: A Virtual View of Data". *Proceedings of the Twenty-Second Annual SAS Users Group International Conference*. Cary, NC: SAS Institute, Inc.

Cody, Ronald (1998). "The INPUT Statement: Where It's @". *Proceedings of the Twenty-Third Annual SAS Users Group International Conference*. Cary, NC: SAS Institute, Inc.

Dickson, Alan, and Pass, Ray (1996). "SELECT ITEMS FROM PROC.SQL Where ITEMS > BASICS". *Proceedings of the Twenty-First Annual SAS Users Group International Conference*. Cary, NC: SAS Institute, Inc.

Gilmore, Jodie (1997). "Using Dynamic Data Exchange with Microsoft Word". *Proceedings of the Twenty-Second Annual SAS Users Group International Conference*. Cary, NC: SAS Institute, Inc.

Heffner, William F. (1998). "DATA Step in Version 7: What's New?". *Proceedings of the Twenty-Third Annual SAS Users Group International Conference*. Cary, NC: SAS Institute, Inc.

Kuligowski, Andrew T., and Roberts, Nancy (1997). "From There to Here: Getting Your Data Into the SAS System". *Proceedings of the Twenty-Second Annual SAS Users Group International Conference*. Cary, NC: SAS Institute, Inc.

Kuligowski, Andrew T. (1998). "An Overview of Techniques to Introduce External Data into the SAS System". *Proceedings of the Sixth Annual Conference of the SouthEast SAS Users Group*. USA.

Kuligowski, Andrew T. (1999). *Course Notes: Turning External Data Into SAS Data.* Dunedin, FL: self-published.

Riba, S. David (1996), *Course Notes: Connecting With Your Data*. Clearwater, FL: JADE Tech, Inc.

Sanders, Roger E. (1998). "Accessing Data from Your PC Using Version 7 of the SAS System". *Proceedings of the Twenty-Third Annual SAS Users Group International Conference*. Cary, NC: SAS Institute, Inc.

SAS Institute, Inc. (1995), *SAS/ACCESS Software for PC File Formats: Reference, Version 6, First Edition.* Cary, NC: SAS Institute, Inc.

SAS Institute, Inc. (1994), *SAS/ACCESS Software for Relational Databases: Reference, Version 6, First Edition.* Cary, NC: SAS Institute, Inc.

SAS Institute, Inc. (1990), *SAS Language: Reference, Version 6, First Edition.* Cary, NC: SAS Institute, Inc.

SAS Institute, Inc. (2000). *SAS OnlineDoc, Version 8.* Cary, NC: SAS Institute, Inc.

SAS Institute, Inc. (1997). *Window by Window: Capture Your Data Using the SAS System.* Cary, NC: SAS Institute, Inc.

Rev up Your Spreadsheets With Some V8 Power

Peter Eberhardt M.A., Fernwood Consulting Group, Inc

ABSTRACT

There is a common need to provide read access to data residing in SAS tables. This tutorial will introduce some components of SAS Integration Technologies, then show how to use these components to get SAS data into a MS Excel spreadsheet – in some cases, without the need for SAS on the user's computer. This tutorial is geared for any SAS programmer who has a good grasp of VBA.

There are few SAS programmers and/or analysts who have not heard the refrain *'Can I have that in an Excel spreadsheet?'*. And, once provided, the additional and inevitable refrain *'Can you run it again with this one change?'*. If you have SAS/Access for PC File Formats, generating the spreadsheets is not too big an issue, assuming you have time to make the change and run the programme. And if you do not have SAS/Access for PC File formats, then you have yet another layer of conversion, additional time, and of course the window of opportunity for errors to creep in. Many people turned to Dynamic Data Exchange (DDE) to try to alleviate these problems. In this paper I will introduce a more robust set of tools aimed at sharing data between SAS and non-SAS applications – SAS Integration Technologies.

INTEGRATION TECHNOLOGIES

New in SAS v8, Integration Technologies is a set of tools that allow you to access the power of the SAS system from a variety of programming environments – Java, Visual Basic etc.. In this paper, we will be looking at using some Windows desktop tools and components to access SAS. In particular, we will look at using Excel VBA as the client programming language, Microsoft's Active Data Objects (ADO) as the data access component, and the SAS Integrated Object Model (IOM) to open SAS to the Excel client.

First, let us briefly examine the SAS Integrated Object Model (for a complete description of SAS Integration Technologies, and the IOM, refer to the SAS web site www.sas.com/rnd/itech/library). Figure 1 depicts the IOM Hierarchy; in this paper we will focus on the Workspace and the ADO/OLE DB components.

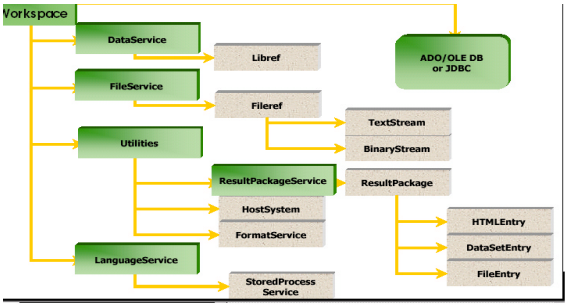


Figure 1.
IOM Hierarchy

The root of the IOM hierarchy is the SAS Workspace object; when instantiated by the Workspace Manager within a client programme, the SAS Workspace object can be thought of as a SAS session. Virtually all of the functionality you would have in a batch SAS session is available to you through the workspace object. The Workspace Manager creates the SAS Workspace objects on an IOM Server. In the Windows environment there are three ways the Workspace Manager can create a SAS Workspace:

- Through local COM if the SAS Server runs on the same machine as the client
- Through DCOM if the SAS Server runs on another machine that supports DCOM
- Through the IOM Bridge for COM if the SAS Server runs on another machine that does not support COM/DCOM functionality (Unix/OS390)

Regardless of how the SAS Workspace is created (COM, DCOM, IOM Bridge), it still offers the same set of services – DataService, FileService, LanguageService, and Utilities.

The LanguageService component provides methods to submit SAS code to the IOM Server as well as retrieve log and list outputs. If you take advantage of the SAS Output Delivery System (ODS) you can use the ResultsPackageService to retrieve collected items.
While the programme is executing, the LanguageService raises events (e.g. step begin, step end) which will allow you to monitor the progress.

The DataService and FileService provide methods to access SAS libraries through librefs or host system files through filerefs. The full range of library and file manipulation tools is available through these services. Microsoft's ADO/OLE DB

data model is used to share data between the client application and the SAS IOM server. The ActiveX Data Object (ADO) model is shown in Figure 2.

MICROSOFT ADO

In order to share data between SAS and Excel we will need to understand a bit about ADO. For the purposes of this paper there are two objects of interest – the Connection object and the Recordset object.

The Connection object provides properties to define the source of the data, and methods to manage the link between the client to the datasource. The Recordset object uses the Connection object to return data to the client. The Fields collection of the Recordset object provides data about the contents of the recordset.

DO YOU HAVE ALL OF YOUR TOOLS?

In order to follow the examples in this paper, you will have to have SAS v8.1 installed; as part of the installation procedure be sure the Integration Technologies components are installed. Microsoft Data Access Components (MDAC) v2.1 or higher should also be installed; normally this will be installed along with SAS Integration Technologies. The examples that follow were developed under Win NT 4.0 sp6 using SAS v8.1 and Excel 97 sr2.

In order to use the SAS IOM within Excel we will need to make sure that Excel has the proper references to the objects. To do this, start Excel and follow these steps

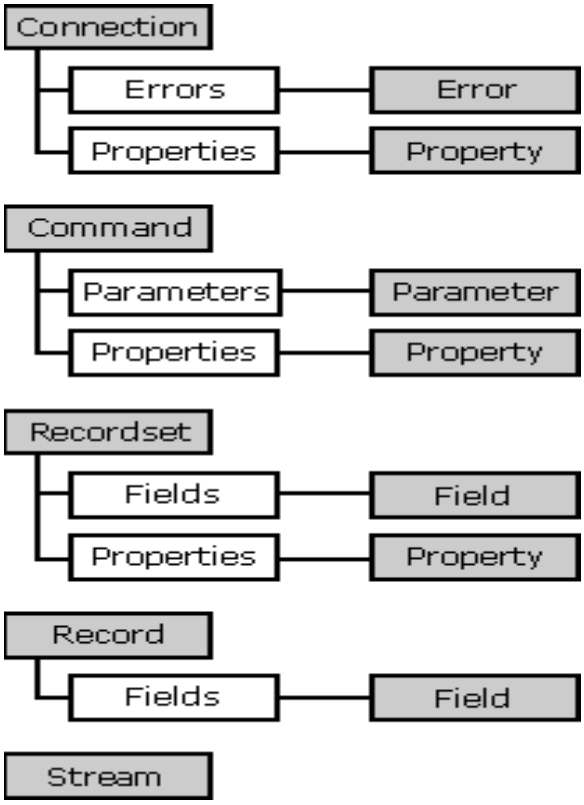


Figure 2.
ADO Hierarchy

from an empty spreadsheet open the Visual Basic Editor (Tools…Macro…Visual Basic Editor)
create a new module (Insert…Module)
add the references to the SAS IOM and the SAS Workspace manager (Tools…References – in the dialogue box, scroll down and select the SAS objects)
add the references to the Microsoft ActiveX Data Objects (Tools…References – in the dialogue box scroll down to Microsoft ActiveX Data Objects and select the highest version available)

DRIVERS, START YOUR ENGINES

Let's start with a simple connection and retrieval of data; in the first case we will retrieve the contents of the table ***sasuser.shoes***. To keep the demonstration as simple as possible and to highlight the IOM components the Excel worksheet will not use common spreadsheet components such as forms and buttons. The first example will simply return the contents of ***sasuser.shoes*** and display the contents in the Excel immediate window. Let us examine each of the lines of code here (see Listing 1). NOTE: watch carefully for code that spans multiple lines. Although I have tried to catch all such spans and add the 'line to be continued' character _ (the underscore), the transfer from Excel

to the word processor may have led to word wrap, and consequently code that will not compile.

First, whenever using VBA modules, always set ***Option Explicit***. This option requires you to declare all variables used. If this option is not set errors in the form of misspelled variables can easily creep into your programme. The next three declarations:

```
Dim swsSAS    As SAS.Workspace
Dim rsSAS     As New ADODB.Recordset
Dim swmWM  As New _
SASWorkspaceManager.WorkspaceManager
```

declare these objects to be global to the module. The use of the ***New*** keyword indicates that the objects should be created when the programme starts. These objects were declared global to the module so they could be available to any function or subroutine within the module. The subroutine ***Test*** does all of the work in this module.

The command:

```
Set swsSAS = _
swmWM.Workspaces.CreateWorkspaceByServer("",_
VisibilityProcess, Nothing, "", "", xmlInfo)
```

creates a SAS workspace (swsSAS) on the local machine. The third parameter (the VBA keyword ***Nothing)*** indicates the SAS server is on the local machine. If the SAS server were to be located on another machine then an appropriate server definition would have to be passed

The command (which should be on one line):

```
cnnIOM.Open "Provider=sas.iomprovider.1; SAS Workspace
ID=" & swsSAS.UniqueIdentifier
```

opens the data connection between the SAS IOM server and the Excel client. The "***Provider= sas.iomprovider.1"*** option indicates the particular SAS dataprovider we will be using (there are three available providers). The "SAS Workspace ID= & swsSAS.UniqueIdentifier" option tells the connection which workspace it is dealing with.

The command:

```
rsSAS.Open "sashelp.shoes", cnnIOM, adOpenDynamic,_
adLockPessimistic, ADODB.adCmdTableDirect
```

opens the SAS table sashelp.shoes for update. The connection object, cnnIOM, identifies where the data reside (a local SAS workspace), the ***adOpenDynamic*** keyword indicates the data are updateable.

A recordset can be positioned in one of three locations, before the first record or the beginning of the file (BOF), after the last record or the end of the file (EOF), or on an active record; if there are no records in the recordset you cannot move to an active record. A common way to test for an empty dataset is to see if both the BOF and EOF properties are true; if both properties are true, there are no records in the recordset. The statement

```
If Not (rsSAS.BOF And rsSAS.EOF) Then
```

then checks if there are any records in the recordset, proceeding only if there are records. There are a number of methods used to navigate a recordset; some of the common ones are:

MoveFirst - move to the first record
MoveLast - move to the last record
MoveNext – move to the next record
MovePrevious – move to the previous record

Within the conditional ***If*** statement, we loop through all of the records in the recordset, listing the contents in the VBA immediate window using the ***Debug.Print*** method.

```
rsSAS.MoveFirst
Do While Not rsSAS.EOF
   Debug.Print rsSAS!region, rsSAS!product,_
rsSAS!subsidiary, rsSAS!stores, rsSAS!sales,_ rsSAS!inventory,
rsSAS!returns
   rsSAS.MoveNext
Loop
```

After looping through all of the records, we close all of the objects we had opened and explicitly destroy them by setting them to ***Nothing***. If objects are not properly closed and destroyed, the memory they occupied is not freed (memory leak) often resulting in your programme slowing drastically and possibly crashing altogether.

FIELDS OF GLORY

The example in Listing 1 allowed us to display the contents of the table ***sasuser.shoes***; however, we had to explicitly identify all of the fields in the recordset. In the next example (Listing 2) we will see how to identify the fields in each record. The following code is the segment which lists the fields:

```
For Each fld In rsSAS.Fields
    Debug.Print fld.Name, fld.Type
Next
```

If you recall from the ADO hierarchy model (Figure 2) the recordset object has a ***Fields*** collection; this code is simply stepping through all of the fields in the collection. The ***For Each …. Next*** construct is a common method to iterate over a collection.

WHAT'S UNDER THE HOOD?

How do you find the options and properties for these objects we are using? If manuals were regularly distributed with software, you could read the manual. Now manuals are replaced with on-line help. Unfortunately, navigating the help files is not always easy or productive. Fortunately there is a way to get the methods and properties of the objects – the VBA object browser. Within the VBA editor select View… Object Browser; the shortcut key id F2 (Figure 3).

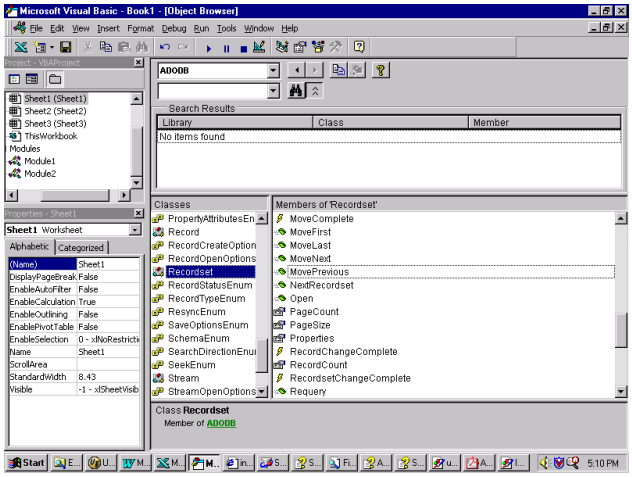


Figure 3
The Object Browser

The example in Figure 3 shows the members (properties and methods) of the recordset object.

LET'S EXCELERATE

Now that we have been able to start SAS from our Excel spreadsheet, let's actually populate a worksheet (Listing 3). This example builds upon the previous one. We will iterate over the ***Fields*** collection to put out column headers (***fld.Name***). In addition we will check for character fields (***fld.Type = adWChar***) and set the column widths to be 2 characters wider than the actual (defined) width (***fld.DefinedSize + 2***). We will also bold the titles and set the cell border to a bottom underline. After displaying the column headers, we then process every record in the recordset as before; within each record we iterate over the ***Fields*** collection and populate the cells with the field value (***fld.Value***). After populating the worksheet, we move to the cell A2.

SUBMIT

Ok, we can read ***sasuser.shoes***. My boss will really find those data useful!! Ok, maybe he will, maybe she won't. But a ***DATA Step***, that we know would be useful. In the next example (Listing 4) we use the ***LanguageService*** to submit a simple data step and return the results. As with the first example, the output will be displayed in the VBA Immediate window. The new piece to the puzzle is the line (the quoted part should all be on one line):

```
swsSAS.LanguageService.Submit _
"data a; do customer=1 to 0;quantity=customer*customer;
pizza='Pepperoni';output;end;run;"
```

This will create a dataset with 10 records and 3 variables. And now with the ability to submit SAS code we can add some real v8 power.

LET'S GET REAL

In order to access some real data we need to use the SAS Workspace DataService to assign a libref to an existing SAS Library (Listing 5). In this example we assign the libref CARD to the directory D:\Cardiac using the command

```
Set libref =
swsSAS.DataService.AssignLibref("card",_"","d:\cardiac", "")
```

Then, in the SAS code in the submit command, we can reference datasets in the specific libref CARD.

THE CHEQUERED FLAG

These examples have been kept simple to highlight a few of the aspects of SAS Integration Technologies. By no means are they exhaustive of the power of SAS Integration Technologies. However, they should start you on your way.

To get started you should have SAS with Integration Technologies installed on your desktop. Once you have the programmes working using a local SAS Server it is a matter of changing only a few options and you are ready to run the same programmes against remote SAS servers. To deploy your applications you need only the SAS Client components and the client application (e.g. Excel) on the desktop and SAS Integration Technologies components on the remote SAS Server.

REFERENCES

For a complete overview of SAS Integration Technologies see www.sas.com/rnd/itech/library

Green, John (1999) Excel 2000 VBA Programmer's Reference
Wrox Press, Birmingham

Jennings, Roger (1999) Database Developer's Guide with Visual Basic® 6
SAMS, Indianapolis IN

About the Author

Peter is SAS Certified Professional V8, SAS Certified Professional V6, and SAS Certified Professional - Data Management V6. In addition his company, Fernwood Consulting Group Inc. is a SAS Quality Partner.

If you have any questions or comments you can contact Peter at:
Fernwood Consulting Group Inc.,
288 Laird Dr.,
Toronto ON M4G 3X5
Canada

Voice: (416)429-5705
e-mail: peter@fernwood.on.ca

LISTINGS

Listing 1

```
Option Explicit ' always set option explicit
Dim swsSAS       As SAS.Workspace
Dim rsSAS        As New ADODB.Recordset
Dim swmWM        As New
SASWorkspaceManager.WorkspaceManager

Public Sub test()
Dim cnnIOM  As New ADODB.Connection
Dim xmlInfo As String

' Create a local SAS workspace.
  Set swsSAS =
swmWM.Workspaces.CreateWorkspaceByServer("",
VisibilityProcess, Nothing, "", "", xmlInfo)

' Open a connection to the workspace
  cnnIOM.Open "Provider=sas.iomprovider.1; SAS
Workspace ID=" & swsSAS.UniqueIdentifier

' Associate the Recordset object with the SAS data set.
  rsSAS.Open "sashelp.shoes", cnnIOM,
adOpenDynamic, adLockPessimistic,
ADODB.adCmdTableDirect
  If Not (rsSAS.BOF And rsSAS.EOF) Then
     rsSAS.MoveFirst
     Do While Not rsSAS.EOF
        Debug.Print rsSAS!region, rsSAS!Product,
rsSAS!subsidiary, rsSAS!stores, rsSAS!sales,
rsSAS!inventory, rsSAS!returns
        rsSAS.MoveNext
     Loop
  End If
  rsSAS.Close
  Set rsSAS = Nothing
cnnIOM.Close
  Set cnnIOM = Nothing
  SwmWM.Workspaces.RemoveWorkspaceByUUID
swsSAS.UniqueIdentifier
swsSAS.Close
  Set swsSAS = Nothing
  Set swmWM = Nothing
End Sub
```

Listing 2

```
Option Explicit ' always set option explicit
Dim swsSAS  As SAS.Workspace
Dim rsSAS   As New ADODB.Recordset
Dim swmWM   As New
SASWorkspaceManager.WorkspaceManager

Public Sub test()
Dim cnnIOM  As New ADODB.Connection
Dim xmlInfo As String
Dim fld     As Field

' Create a local SAS workspace.
  Set swsSAS =
swmWM.Workspaces.CreateWorkspaceByServer("",
VisibilityProcess, Nothing, "", "", xmlInfo)

' Open a connection to the workspace
cnnIOM.Open "Provider=sas.iomprovider.1; SAS
Workspace ID=" & swsSAS.UniqueIdentifier

' Associate the Recordset object with the SAS data set.
  rsSAS.Open "sashelp.shoes", cnnIOM,
adOpenDynamic, adLockPessimistic,
ADODB.adCmdTableDirect

  For Each fld In rsSAS.Fields
    Debug.Print fld.Name, fld.Type
  Next

  rsSAS.Close
  Set rsSAS = Nothing
cnnIOM.Close
  Set cnnIOM = Nothing
  SwmWM.Workspaces.RemoveWorkspaceByUUID
swsSAS.UniqueIdentifier
SwsSAS.Close
  Set swsSAS = Nothing
  Set swmWM = Nothing
End Sub
```

Listing 3

```
Option Explicit ' always set option explicit

Dim swsSAS  As SAS.Workspace
Dim rsSAS   As New ADODB.Recordset
Dim swmWM   As New
SASWorkspaceManager.WorkspaceManager

Public Sub test()
Dim cnnIOM  As New ADODB.Connection
Dim xmlInfo As String
Dim count   As Integer
Dim fld     As Field
Dim row     As Long


' Create a local SAS workspace.
  Set swsSAS =
swmWM.Workspaces.CreateWorkspaceByServer("",
VisibilityProcess, Nothing, "", "", xmlInfo)

' Open a connection to the workspace
cnnIOM.Open "Provider=sas.iomprovider.1; SAS
Workspace ID=" & swsSAS.UniqueIdentifier

' Associate the Recordset object with the SAS data set.
  rsSAS.Open "sashelp.shoes", cnnIOM,
adOpenDynamic, adLockPessimistic,
ADODB.adCmdTableDirect

' SELECT  the first sheet and freeze the panes on the 2nd
line
  Worksheets("sheet1").Activate
  Range("A2").Select
  ActiveWindow.FreezePanes = True
  If Not (rsSAS.BOF And rsSAS.EOF) Then
    Worksheets("sheet1").Activate
    Range("A1").Select
    For Each fld In rsSAS.Fields
      ActiveCell.Value = fld.Name
      ActiveCell.Font.Bold = True
      With ActiveCell.Borders(xlBottom)
        .LineStyle = xlContinuous
        .Weight = xlThin
      End With

      If fld.Type = adWChar Then
        Columns(ActiveCell.Column).ColumnWidth =
fld.DefinedSize + 2
      Else
        Columns(ActiveCell.Column).ColumnWidth = 12
      End If
      col = col + 1
      ActiveCell.Next.Select
    Next
    rsSAS.MoveFirst
    row = 2
    Do While Not rsSAS.EOF
      For Each fld In rsSAS.Fields
        ActiveCell.Value = fld.Value
        ActiveCell.Next.Select
      Next
      row = row + 1
      RsSAS.MoveNext
    Loop
  Range("A2").Select

  End If

  rsSAS.Close
  Set rsSAS = Nothing
  CnnIOM.Close
  Set cnnIOM = Nothing
  SwmWM.Workspaces.RemoveWorkspaceByUUID
swsSAS.UniqueIdentifier
  SwsSAS.Close
  Set swsSAS = Nothing
  Set swmWM = Nothing
End Sub
```

Listing 4

```
Option Explicit ' always set option explict
Dim swsSAS       As SAS.Workspace
Dim rsSAS        As New ADODB.Recordset
Dim swmWM        As New
SASWorkspaceManager.WorkspaceManager

Public Sub test()
Dim cnnIOM As New ADODB.Connection
Dim xmlInfo As String

   ' Create a local SAS workspace.
Set swsSAS =
swmWM.Workspaces.CreateWorkspaceByServer("",
VisibilityProcess, Nothing, "", "", xmlInfo)

   ' Use LanguageService
swsSAS.LanguageService.Submit "data a; do customer=1
to 10;quantity=customer*customer;
pizza='Pepperoni';output;end;run;"

' Open a connection to the workspace
cnnIOM.Open "Provider=sas.iomprovider.1; SAS
Workspace ID=" & swsSAS.UniqueIdentifier

' Associate the Recordset object with the SAS data set.
rsSAS.Open "work.a", cnnIOM, adOpenDynamic,
adLockPessimistic, ADODB.adCmdTableDirect
If Not (rsSAS.BOF And rsSAS.EOF) Then
   rsSAS.MoveFirst
   Do While Not rsSAS.EOF
      Debug.Print rsSAS!CUSTOMER,
rsSAS!QUANTITY, rsSAS!PIZZA,
rsSAS!ORDERDATE
      rsSAS.MoveNext
   Loop
End If
rsSAS.Close
Set rsSAS = Nothing
cnnIOM.Close
Set cnnIOM = Nothing
swmWM.Workspaces.RemoveWorkspaceByUUID
swsSAS.UniqueIdentifier
swsSAS.Close
Set swsSAS = Nothing
Set swmWM = Nothing
End Sub
```

Listing 5

```
Option Explicit ' always set option explict
Dim swsSAS       As SAS.Workspace
Dim rsSAS        As New ADODB.Recordset
Dim swmWM        As New
SASWorkspaceManager.WorkspaceManager

Public Sub test()
Dim cnnIOM As New ADODB.Connection
Dim xmlInfo As String
Dim count   As Long
Dim libref  As SAS.libref
Dim fld     As Field

' Create a local SAS workspace.
   Set swsSAS =
swmWM.Workspaces.CreateWorkspaceByServer("",
VisibilityProcess, Nothing, "", "", xmlInfo)
   Set libref = swsSAS.DataService.AssignLibref("card",
"", "d:\cardiac", "")
   swsSAS.LanguageService.Submit "data a; set
card.revup;run;"

' Open a connection to the workspace
   cnnIOM.Open "Provider=sas.iomprovider.1; SAS
Workspace ID=" & swsSAS.UniqueIdentifier

' Associate the Recordset object with the SAS data set.
   rsSAS.Open "work.a", cnnIOM, adOpenDynamic,
adLockPessimistic, ADODB.adCmdTableDirect
   If Not (rsSAS.BOF And rsSAS.EOF) Then
      For Each fld In rsSAS.Fields
         Debug.Print fld.Name
      Next

      rsSAS.MoveFirst
       Do While Not rsSAS.EOF
   '       Debug.Print rsSAS!CUSTOMER,
rsSAS!QUANTITY, rsSAS!PIZZA,
rsSAS!ORDERDATE
           rsSAS.MoveNext
       Loop
   End If
   rsSAS.Close
   Set rsSAS = Nothing

   swsSAS.DataService.DeassignLibref "card"
   swmWM.Workspaces.RemoveWorkspaceByUUID
swsSAS.UniqueIdentifier
   swsSAS.Close
   Set swsSAS = Nothing
   Set swmWM = Nothing
End Sub
```

Using Functions and Arrays in the SAS® System to Manage and Manipulate Data

Ben Cochran, The Bedford Group, Raleigh, NC

Abstract

Oftentimes, SAS users receive data that is alleged to be cleaned, scrubbed, and ready for an application or placement into a Data Warehouse. Even though the data may be clean, many times further manipulation is needed. This paper examines ways of manipulating data through the use of functions and arrays in the SAS System.

This paper is intended for the beginning and intermediate level SAS user. Functions and arrays are a part of the base SAS System; therefore, no other SAS product will be examined in this paper.

Basic Data Manipulation

Anytime users need to change existing data to meet our application needs, they are manipulating the data. One of the most basic forms of manipulating data is rearranging the order of the observations, or, in other words, sorting it. Sorting data not only rearranges the order of the observations, it allows for **by- group** processing. An example of this is merging two data sets by a common variable.

Another way to manipulate data is through the use of SAS functions.

SAS Functions

A SAS function is a pre-written routine that returns a value. Functions are often categorized by the type of data manipulation performed. These categories include: Data Conversion, Truncation, Arithmetic, Mathematical, Trigonometric, Statistical, Financial, Random Number, Date and Time, and State and Zip.

The general form of a SAS function is:

function-name (argument, argument, …);

The arguments are surrounded by a set of parentheses. The number of arguments depends on the function. Some functions take a specific number of arguments, some functions have a varying number of arguments, some functions do not have any arguments at all. But there is ALWAYS the set of parentheses that follow the function. Each argument is separated from the others by a comma.

The arguments can be: constants, expressions, variables, or other functions. In other words, functions can be nested.

When using functions to manipulate data several methods can be used to do this. Following are some of the more popular ways to use functions.

Creating Variables with Functions

Many times we need to create values from existing data because what we need for a report or other application is not originally in the data. This is done by using an assignment statement which it has the following syntax**:**

new_variable = expression **;**

New variables can be created by using functions as part of the expression. In this case, the assignment statement would look like this**:**

x = function-name(argument, argument, . .)**;**

Manipulating Date Variables:

One of the most common examples of creating new variables is found in the manipulation of DATE values. For example, we may be asked to build a drill-down bar chart in SAS/EIS where

the initial view shows a bar for each year. The length of each bar reflects the number of tests performed by a clinic. Clicking on a bar (YEAR) would then yield a display showing a bar for each quarter within the chosen year. Clicking on a bar (QUARTER) would then yield a display showing a bar for each month within the chosen quarter, etc.

In order to accomplish this task, there needs to be a variable for YEAR, QUARTER, and MONTH in the existing data. Upon examining the data, we find that there is NO variable for YEAR, QUARTER, or MONTH. Looking closer, we see that there is a variable named DATE that contains the SAS date of the test. If we know that there are functions available to us that can create the needed variables, we can use them in a DATA step similar to the one below:

```
data new_data;
      set dates;
      year = year(date);
      quarter = qtr(date);
      month = month(date);
run ;
```

In the above example, notice that the variables being created are named YEAR (from the YEAR function), QUARTER (from the QTR function), and MONTH (from the MONTH function). Function names are not reserve words in the SAS language.

Manipulating Other Numeric Variables:
Suppose we want the highest, lowest, and average cholesterol readings for each patient. The data is in a SAS data set named **CHOLESTEROL** and has the following form:

Pat_id	date1	chol1	date2	chol2	date3	chol3
1201	010600	211.1	070200	201.4	112100	191.1
1212	010400	181.7	063000	188.0	120100	191.8
1222	020100	194.8	071100	195.3	121100	196.0
1261	011800	199.7	062800	201.1	120600	211.1
. . .						

In order to get the desired statistics from this data, write a data step using the **MIN**, **MAX**, and **MEAN** functions. The **MIN** function returns the minimum value from its arguments. The **MAX** function returns the maximum value from its arguments. The **MEAN** function returns the average from its arguments. Notice how these functions are used in the following DATA step:

```
data cholesterol_stats (keep=pat_id high
                                  low average) ;
    set cholesterol;
    high  =  max ( of chol1 - chol3);
    low   =  min (of  chol1 -  chol3);
    average  = mean (of chol1 - chol3);
run;
```

In the above DATA step, the variables HIGH, LOW and AVERAGE are created for the MAX, MIN, and MEAN functions, respectively. In this example, the arguments are NOT separated by commas. Instead, a variable list notation, the single dash ('-'), is used. Also, notice the use of the keyword 'OF' in the argument list. The SAS system interprets this as the variable list starting with CHOL1 and going through CHOL3. If the word 'OF' were not used in this example, the SAS system would interpret the CHOL1 – CHOL3 as a subtraction equation.

Suppose your data was stored in a SAS data set named **CHOLESTEROL2** and has the following structure.

Pat_id	date	chol_reading
1201	010600	211.1
1201	070200	201.4
1201	112100	191.1
1212	010400	181.7
1212	063000	188.0
1212	120100	191.8
1222	020100	194.8
1222	071100	195.3
1222	121100	196.0
. . .		

Suppose we are asked to develop an application that can indicate whether a patient's cholesterol reading has gone up or down from the previous reading. In order to get the desired results from this data, it will have to be manipulated with the **LAG** function as in the DATA step below:

```
data compare;
    set cholesterol2;
    by pat_id;
    last_reading = lag(chol_reading);
    if first.pat_id = 1 then last_reading = . ;
   difference = chol_reading - last_reading ;
run;
```

At compile time, the **LAG** function creates a buffer. During the execution of the DATA step, the value of CHOL_READING is stored in this buffer and then assigned to the variable

LAST_READING in the next execution of the DATA step. But, if the first observation from a PAT_ID is being processed, then there is no previous value for CHOL_READING (for that patient) so this DATA step sets the value of LAST_READING to missing. The resulting data set, **COMPARE** can be seen below:

Pat_id	date	chol_reading	last_reading	difference
1201	010600	211.1	.	.
1201	070200	201.4	211.1	-9.7
1201	112100	191.1	201.4	-10.3
1212	010400	181.7	.	.
1212	063000	188.0	181.7	6.3
1212	120100	191.8	188.0	3.8
1224	020100	194.8	.	.
1224	071100	195.3	194.8	.5
1224	121100	196.0	195.3	.7
. . .				

You can generate **random numbers** from various distributions using random number functions. One of the uses of this type of function is found in applications that generate random samples.

The **PATIENTS** data set contains thousands of observations. The first four observations are:

Pat_id	name	address	city	st
1201	Welch, W. B.	21 East St.	Apex	NC
1212	Coxe, Jan S.	43 King Rd.	Cary	NC
1224	Dow, Sue A.	19 Elm St.	Garner	NC
1261	Moore, Ron	16 Oak Av.	Apex	NC
. . .				

You want to create a sample of 25 randomly chosen patients. You can do this with the following DATA step which uses the **CEIL** and **RANUNI** functions.

```
data sample_of_25;
   do i = 1 to 25;
      rn = ceil(ranuni(0) * total_obs);
      set patients point=rn nobs=total_obs;
      output;
   end;
   stop;
run;
```

First, look at the **NOBS=** option on the SET statement. At compile time, it reads the descriptor portion of the PATIENTS data set, finds out the number of observations in the data set, and stores that number in the variable TOTAL_OBS.

Both the CEIL and the RANUNI functions are used to create the variable RN. First, the RANUNI function returns a randomly generated number between 0 and 1. Then, that randomly generated number is multiplied by the value of TOTAL_OBS. The result is some number that is between something less than 1 and the number of observations in the PATIENTS data set. This number is then fed into the CEIL function that rounds up to the next highest integer. If the previous number is already an integer, the CEIL function has no effect. The result of this is a whole number whose value is between 1 and the number of observations in the PATIENTS data set. This number is then assigned to the variable called RN.

The POINT= option causes the SET statement to read the RNth number in the PATIENTS data set. Then this observation is OUTPUT to the SAMPLE_OF_25 data set. As this process goes through the DO LOOP 25 times, 25 observations are created for the new data set.

The STOP statement is absolutely necessary to prevent an endless loop. (The DATA step will continue to loop because the end of file is never encountered on PATIENTS data set).

Manipulating Character Variables:

The billing department of the clinic wants to use the PATIENTS data set to create mailing labels for patient bills. To do so, the values of the NAME variable need to be manipulated as follows:

Pat_id	name	➔	fullname
1201	Welch, W. B.		Mr. W.B. Welch
1212	Coxe, Jan S.		Ms. Jan S. Coxe
1222	Dow, Sue A.		Ms. Sue A. Dow
1261	Moore, Ron		Mr. Ron Moore
. . .			

In this data set, the PAT_ID ends in a 1 if the gender of the patient is male, or ends in a 2 if the patient is female. PAT_ID is a character variable with a length of 4. Some of values contain four digits, some contain three. So, we need to examine the last digit, if it is a 1, then the courtesy title is Mr., else if the digit is a 2, then the courtesy title is Ms.

The name field needs to be transformed from lastname, firstname, and middle initial to courtesy title, firstname, middle initial, lastname.

The following data step uses the SUSTR, LENGTH, SCAN, and TRIM functions and the CONCATENATION operator to manipulate character variables.

```
data mailing_list;
  set patients;
  if substr(pat_id, length(pat_id)) ='1' then
            title='Mr. ';
  else if substr(pat_id, length(pat_id)) = '2' then
            title='Ms. ';
   f_name_mi = scan(name, 2, ',');
   last_name  = scan(name, 1, ',');
   full_name = title!!trim(f_name_mi)!!' ' !!
        last_name;
run;
```

In this example,

- the **LENGTH** function finds the length of the character string stored in the PAT_ID variable. It returns a number, in this case, a three or a four.
- The **SUBSTR** function has three arguments and creates a substring from its first argument, in this case, PAT_ID. The second argument marks the beginning of the substring that will be created from the first argument. In this case, the substring starts on the last character of PAT_ID. The third argument is optional, and it indicates how long the substring will be. The default is to go to the end of argument 1. In other words, the substring is going to start with the last character of argument 1 and go for a length of one. It will pick the last character from PAT_ID whether it has three or four digits. The value returned by the SUBSTR function is NOT assigned to any variable, but is used to compare with the quoted value, a '1' or a '2'. If it is a '1' then the value of TITLE will be 'Mr. ', if it is a '2', then the value of TITLE will be 'Ms. '.
- The **SCAN** function scans a character string, looking for the nth 'word' separated by a delimiter. A word is a text string between two delimiters. The SCAN function has three arguments. The first is the text to be scanned, the second is the word, and the third specifies the delimiter. In this DATA step the SCAN function is first used to create the variable F_NAME_MI. It scans the variable NAME, looking for the second word separated by a comma. The SCAN function is next used to create the variable LAST_NAME. It scans the variable NAME looking for the first word separated by a comma.
- The **CONCATENATION** operator (!!) is used to create FULL_NAME. It has the effect of 'gluing' together character strings. First, TITLE is glued to the **TRIM** of F_NAME_MI. The TRIM function trims all trailing blanks from the end of a string. Next, a single blank is 'glued' to FULL_NAME. This separates F_NAME_MI from LAST_NAME by a single blank. Finally, LAST_NAME is joined to FULL_NAME.

Variable Conversion

In many applications we must convert one data type to another. The first example illustrates the conversion form character values to numeric values. Recall the **CHOLESTEROL** data set.

Pat_id	date1	chol1	date2	chol2	date3	chol3
1201	010600	211.1	070200	201.4	112100	191.1
1212	010400	181.7	063000	188.0	120100	191.8
1222	020100	194.8	071100	195.3	121100	196.0
1261	011800	199.7	062800	201.1	120600	211.1
. . .						

One of the clinical physicians wants to examine the length of time (in days) between testing. When the PATIENTS data set was created, all the date variables were created as character variables. They need to be converted to numeric so that the number of days between DATE1 and DATE2 can be calculated. The **INPUT** function is used to do such a conversion.

The INPUT function has 2 arguments. The first is the variable to be converted, the second is an informat that states HOW the variable will be converted.

```
data numeric_dates;
    set cholesterol;
    drop chol1 – chol3 date3;
    num_date1=input(date1, mmddyy6.);
    num_date2=input(date2, mmddyy6.);
    span1 = num_date2 – num_date1;
run;
```

In this example, the **INPUT** function first reads the DATE1 variable with the MMDDYY6. informat and stored the numeric results in the variable NUM_DATE1. NUM_DATE2 is

created in a similar manner. View the **NUMERIC_DATES** data set below:

Pat_id	date1	date2	num_ date1	num_ date2	span
1201	010600	070200	14615	14793	178
1212	010400	063000	14613	14791	178
1222	020100	071100	14641	14802	161
1261	011800	062800	14627	14789	162
. . .					

The next example illustrates numeric to character conversion. Recall the **COLESTEROL2** data set.

Pat_id	date	chol_reading
1201	010600	211.1
1201	070200	201.4
1201	112100	191.1
1212	010400	181.7
1212	063000	188.0
1212	120100	191.8
1222	020100	194.8
1222	071100	195.3
1222	121100	196.0
. . .		

When this data set was created, PAT_ID was created as a numeric variable. Remember its values are either a 3 or 4 digit number. As a numeric variable in a SAS data set, it takes up eight bytes of storage. If we convert it to a character variable, it takes up only four bytes of storage. The following DATA step illustrates how to do this conversion.

```
data character_id(drop=x_id);
    set cholesterol2(rename=(pat_id=x_id));
    pat_id = put(x_id, $4.);
run;
```

The PAT_ID variable in the CHOLESTEROL2 data set is renamed to X_ID. The PUT function reads the value of X_ID and then writes it to a new PAT_ID using the $4. format. The PUT function always creates a character variable. The new data set, CHARACTER_ID looks just like CHOLESTEROL2. The only difference is that PAT_ID is a character variable in CHARACTER_ID while PAT_ID is a numerical variable in COLESTEROL2.

Data Transformation

Sometimes the data that you are working with is in the shape of the **CHOLESTEROL** data set (one observation per PAT_ID).

Pat_id	date1	chol1	date2	chol2	date3	chol3
1201	010600	211.1	070200	201.4	112100	191.1
1212	010400	181.7	063000	188.0	120100	191.8
1222	020100	194.8	071100	195.3	121100	196.0
1261	011800	199.7	062800	201.1	120600	211.1
. . .						

The application that you are developing needs data in the 'shape' of the **CHOLESTEROL2** data set (one observation per cholesterol reading).

Pat_id	date	chol_reading
1201	010600	211.1
1201	070200	201.4
1201	112100	191.1
1212	010400	181.7
1212	063000	188.0
1212	120100	191.8
1222	020100	194.8
1222	071100	195.3
1222	121100	196.0
. . .		

The following DATA step will transpose the data from one observation per PAT_ID to one observation per cholesterol reading.

```
data transpose1(keep=pat_id date chol_reading);
    set cholesterol;
    array dates {3} date1 date2 date3;
    array chols {3} chol1 chol2 chol3;
    do i = 1 to 3;
       date = dates{i};
       chol_reading = chols{i};
       output;
     end;
run;
```

In this example, the array statements collect three variables under each array. When the DO loop iterates the first time, the value of the first variable in the DATES array (the variable DATE1) is assigned to the variable DATE. Next, the value of the first variable in the CHOLS array (the variable CHOL1) is assigned to the variable CHOL_READING. Then the output statement writes out the first observation.

When the DO loop iterates the second time, the value of the second variable in the DATES array (the variable DATE2) is assigned to the variable DATE. Next, the value of the second variable in the CHOLS array (the variable CHOL2) is assigned to the variable CHOL_READING. Then the output statement writes out the second observation.

When the DO loop iterates the third time, the value of the third variable in the DATES array (the variable DATE3) is assigned to the variable DATE. Next, the value of the third variable in the CHOLS array (the variable CHOL3) is assigned to the variable CHOL_READING. Then the output statement writes out the third observation. Notice that the OUTPUT statement executes three times for every single time the SET statement executes.

To transpose the data from one observation per cholesterol reading to one observation per PAT_ID use the following DATA step.

```
data transpose2(keep=pat_id date chol_reading);
   array dates {3} $ date1 date2 date3;
   array chols {3} chol1 chol2 chol3;
   do i = 1 to 3;
      set cholesterol2;
      dates{i} = date;
      chols{i} = chol_reading;
   end;
   output;
run;
```

First of all, notice in this example that the SET statement executes three times for every single time the OUTPUT statement executes. In other words, for every three observations read, only one is written.

Next, notice the '$' in the ARRAY statement. This is an example of an ARRAY statement that creates the variables DATE1, DATE2, and DATE3. In this DATA step, the ARRAY statement appears before the SET statement, meaning that without the '$' on the ARRAY statement, the date variables would be created as numeric. In the CHOLESTEROL2 data set, all the date variables are character. The '$' just keeps SAS from doing an automatic numeric to character conversion.

Conclusion

The SAS System is very robust and has well over a hundred functions and other ways to manipulate data. Only a few of them are presented in this paper. For a full discussion of all the functions available, a user would have to consult the literature published by SAS Institute, Inc. The best place to start would be the SAS Language Reference Guide, or the on-line documentation available for Version 8 of the SAS System.

The author can be reached at the following location:

Ben Cochran
The Bedford Group
3216 Bedford Avenue
Raleigh, NC 27607
919.831.1191

bedford.group@mindspring.com

The Bedford Group is a SAS Institute Quality Partner and SAS Certified Professional for Version 8. They specialize in SAS consulting and training.

Trademark Information

Changes and Enhancements to PROC MEANS In Version 8 of the SAS® System

Andrew H. Karp
Sierra Information Services, Inc.
Sonoma, California USA

PROC MEANS (and is "sister," PROC SUMMARY) have been BASE SAS Software procedures for a long time. Most SAS Software users have found their ability to rapidly analyze and summarize the values of numeric variables to be essential tools in their programs and applications. A number of new features have been added to PROC MEANS in The Nashville Release (Version 8) of the SAS System which will are discussed in this paper

Background

This paper assumes the reader is already familiar with core PROC MEANS/SUMMARY capabilities. As a reminder, with the release of Version 6 of the SAS System in 1991 PROC MEANS and PROC SUMMARY became identical procedures, with just some very differences that are documented in the BASE SAS Software documentation. For the purposes of this paper we can treat them as identical procedures. The core difference is that by default PROC MEANS sends the results of its "work" to our Output Window and that PROC SUMMARY, by default, creates a SAS data set. All of the examples in this paper show PROC MEANS syntax, which you can easily switch to PROC SUMMARY if you want.

The core function of these procedures is to analyze the values of numeric variables in SAS data sets (or SAS views to data stored in other RDBMS products). For both PROCs, only numeric variables can be placed in the VAR statement. Variables places in the VAR statement are considered *analysis variables.* If you put the names of character variables in the VAR statement, PROC MEANS/SUMMARY will not execute and an error will be shown in your SASLOG.

Variables placed in the CLASS or BY Statement are considered *classification variables*, and may be either character or numeric. Starting in Version 6, you could use the CLASS statement to request analyses at the different levels of an unsorted classification variable by using the CLASS, instead of the BY Statement. This feature remains in Version 8, and enhancements to it are discussed below.

New Statistics Available in Version 8

Prior to Version 8, the only BASE SAS procedure that could calculate the values of *quantile statistics* such as the median (50th percentile) was PROC UNIVARIATE. In Version 8, PROCs MEANS and SUMMARY (as well as PROC TABULATE) can also analyze and report the values of quantile statistics.

Consider the following PROC MEANS task, which analyzes a data set containing electric consumption data from a public utility.

```
proc MEANS NOPRINT
data=electric.elec_V8;
class rate_schedule;
var total_revenue;
output out=new2 sum=median_REV
mean=total_REV p50=mean_REV;
run;
```

The OUTPUT Statement directs the placement, in a temporary SAS data set, of new variables containing the results of the analylses requested elsewhere in the PROC MEANS Task. The *Statistics Keyword* **P50** requests the fiftieth percentile or the median, of the analysis variable be placed in a variable called mean_REV in output data set new2.

This PROC MEANS task will run without errors. Mean_REV is a valid SAS variable name, even though what we are doing is storing the median in a variable called "mean_REV." If you look at the OUTPUT Statement again you will see that the sum will be stored in "Median_REV" and the mean will be stored in "Total_REV," all valid SAS variable names in Version 8. (Remember, in V8 we can use up to 32 characters for variables names,)

You would probably not notice your mistake until sometime later in the project when you--or

perhaps your boss--realize that the values associated with the variable names simply don't "make sense." A new Version 8 feature, discussed below, will keep you from ever making this mistake again. It's called the AUTONAME option.

The AUTONAME Option

This handy option goes in the OUTPUT statement. When you use it, PROC MEANS will automatically give names to the variables in the output SAS data sets. Here is an example, again using the electric consumption data set.

```
proc means noprint
data=x.electricity;
class division serial;
var kwh1 rev1;
output out=new4 mean(kwh1) =
             sum(rev1) =/autoname;
run;
```

Output data set NEW4 contains the classification variables DIVISION and SERIAL, the automatically generated variables _TYPE_ and _FREQ_ (about which more later), and the variables KWH1_mean and REV1_mean. The AUTONAME option automatically appended the statistics keyword to the name of the analysis variable, with an underscore symbol between the analysis variable and the statistics keyword. Using this new option will prevent you from giving variables in our output data sets the "wrong" names!

The CHARTYPES Option

This V8 addition to PROC MEANS dramatically simplifies creation of multiple output SAS data sets in a single use of PROC MEANS.

Users can code multiple OUTPUT statements in a single PROC MEANS task and then use a WHERE clause data set option to limit the output of observations to each data set. This capability reduces--and often eliminates--the need to run PROC MEANS several times to create different output data sets. You can usually do everything you need to do in one invocation of the procedure.

Here is an example. A catalog retail firm has a SAS data set containing order records and an analyst wants to create several output SAS data sets, using PROC MEANS, containing analyses at different combinations of the values of four classification variables. Consider the following PROC MEANS task:

```
proc means noprint
data=order.orderfile2;class mailcode
dept_nbr segment status;var itmprice
itm_qty;output out=new  sum=;run;
```

With four variables in the CLASS statement, the temporary SAS data set, NEW, created by the OUTPUT statement, will contain the sum of analysis variables ITMPRICE and ITM_QTY at all possible combinations of the classification variables. There will be sixteen values of the SAS-generated variable _TYPE_ in output data set NEW, representing what I like to call a "complete" analysis of the numeric analysis variables at all possible combinations of the classification variables.

Suppose the analyst wanted to create three separate output SAS data sets, containing the sum of ITMPRICE and ITM_QTY. The separate data sets would contain the desired analyses

By MAILCODE and SEGMENT
By MAILCODE, SEGMENT and STATUS
By DEPT_NBR and SEGMENT

One approach would be to run PROC MEANS three separate times, with different classification variables in the CLASS Statement. When working with very large data sets, this approach, while giving the desired results, wastes computing resources as the source data set will be read three separate times.

Another approach would be to create "one big data set," containing the "complete" analysis, as shown above, and then use a data step to read each observation and output some of them to various subset data sets. Using the previous example, the observations in temporary data set NEW would be tested in a data step and those meeting the condition of interest would be output to new data sets. While this approach still yields the desired outcome, what the analyst would have to do is create a big data set and then test each of its observations to find the ones she wants to put in the smaller data sets. Here is an example:

```
Data A B C:
  SET NEW;
   IF _TYPE_ = '1001'B then OUTPUT A;
ELSE IF _TYPE_ = '1011'B
```

```
      then OUTPUT B;
ELSE IF _TYPE_ = '0110'B then
OUTPUT 'C';
RUN;
```

This data step shows an underutilized feature of the SAS Programming Language, which is called the "bit-testing facility." By using it, we don't have to know the numeric value of _TYPE_, we just need to know the position of the classification variables in the CLASS statement. In this context, a number one means "I want this variable," and a number zero means "I don't want this variable." You can check for yourself that 1001 in base 2 is equal to nine (9) in base 10.

A third approach would be to determine the numeric values of the desired values of _TYPE_ and put them in the output statement as WHERE clause conditions.

The new **CHARTYPES** option makes all of this unnecessary. This option, placed in the PROC MEANS statement, converts the (default) numeric values of _TYPE_ to a character variable containing zeros and ones. The length of this variable is equal to the number of variables in the CLASS (or BY) statement. Keep in mind, that even though this variable contains zeros and ones, it is a character variable.

Using the **CHARTYPES** option makes it much easier to create multiple output data sets in a single application of PROC MEANS. Our marketing analyst could make the three data sets she wants by submitting the following PROC MEANS task:

```
proc means noprint data=order.orderfile2
chartypes;class mailcode dept_nbr segment
status;var itmprice itm_qty;output
out=one(where=(_type_ = '1001'))
       sum=;output out=two(where=(_type_ =
'1011'))
       sum=;output out=three(where=(_type_ =
'0110'))
       sum = ;run;
```

By using the **CHARTYPES** option, the analyst easily creates the three desired data sets in a single use of PROC MEANS. As with the previously-described bit-testing facility in the SAS Programming Language, a one (1) means "I want this variable" and a zero (0) meams "I don't want this variable." Again this is a character, not a numeric variable, and in order to use it effectively you need to know the ordering of the variables in your CLASS statement.

The DESCENDTYPES Option

By default, observations in data sets created by PROC MEANS are ordered in ascending values of the variable _TYPE_. So, _TYPE_ = 0 will be first, followed by _TYPE_ = 1, and so forth, with the highest value of _TYPE_ at the bottom.

The new **DESCENDTYPES** option, which is placed in the PROC MEANS statement, instructs the procedure to order observations in the output data sets it creates in descending value of _TYPE_.

This option is very handy if you want the observation with _TYPE_ = 0 at the bottom of your data set, rather than at the top.

The TYPES Statement

This Version 8 enhancement to PROC MEANS should not be confused with the _TYPE_ variable discussed previously.

By default, PROC MEANS will analyze the numeric analysis variables at all possible combinations of the values of the classification variables. With the **TYPES** statement, only the analyses specified in it are carried out by PROC MEANS. This new feature can save you both programming and processing time.

The next example shows how the **TYPES** statement is used to restrict the operation of PROC MEANS to analyzing the values of the analysis variables to just the combination(s) of classification variables in the CLASS or BY Statement. In PROC MEANS task that follows, the marketing analyst working with the previous-discussed order history file wants to create a single output data set containing analyses of ITMPRICE and ITM_QTY at the following combinations of the CLASS variables

- Overall (_TYPE_ = 0)
- SEGMENT and STATUS
- MAILCODE and SEGMENT
- MAILCODE and DEPT_NBR and SEGMENT

The **TYPES** Statement shown below limits the execution of the PROC MEANS task to just the combinations of the classification variables specified in this statement. Because only only output data set is requested in this task, temporary data set A will contain all of the analyses requested by **TYPES** statement.

```
proc means noprint
data=order.orderfile2 ;class mailcode
dept_nbr segment status;types ()
segment * status      mailcode *
segment      mailcode * dept_nbr *
segment;var itmprice itm_qty;output
out=a sum = ;run;
```
More complex implementations of the **TYPES** statement are documented in the PROC MEANS chapter in the SAS Procedures documentation.

Multiple CLASS Statements

Multiple CLASS Statements are now permitted in PROC MEANS. In previous releases of SAS System software the values of classification variables were either portrayed in the Output Window, or had their values stored in output data sets, in "sort order." New options in the CLASS statement permit user control over how the levels of the classification variables are portrayed.

When using multiple CLASS statements, how you order them in the PROC MEANS task is very important. If you have two CLASS statements, for example, the values of the classification variables in the *second* CLASS statement will be *nested within* the values of the variables in the *first* class statement.

The next PROC MEANS task shows how two CLASS statements were used to analyze some electrical utility data stored in a SAS data set. The first CLASS statement requests an analysis by the values of REGION. The DESCENDING option to the right of the slash in the first CLASS statement instructs PROC MEANS to analyze the data in DESCENDING order of the values of REGION.

The second CLASS statement requests that the data be analyzed by the values of the classification variable TRANSFORMER. Since no options are specified in the second CLASS statement, the values of TRANSFORMER will be portrayed in "sort order," within the descending values of REGION.

```
proc means
data=electric.elec_v8 noprint nway;
class region/descending;
class transformer;
var total_revenue ;
output out=c sum= mean= /autoname;
run;
```

Additional CLASS Statement Options

There are several other useful options now available in the CLASS Statement. A complete list is found on pages 636-638 of the Version 8 SAS Procedures Guide within Chapter 24 (The Means Procedure). Of these, we will discuss the DESCENDING. EXCLUSIVE, GROUPINTERNAL, MISSING, MLF, ORDER= and PRELOADFMT options.

- ***The DESCENDING Option***

This option, discussed above, orders the levels of the CLASS variables in descending order.

- ***The EXCLUSIVE Option***

This option, used in conjunction with the **PRELOADFMT** option excludes from analysis all the combinations of CLASS variables that are not in the preloaded range of user-defined formats (see the **PRELOADFMT** option, below

- ***The GROUPINTERNAL Option***

This option, which saves computing resources when your numeric classification variables contain discrete values, tells PROC MEANS to NOT apply formats to the class variables when it groups the values to create combinations of CLASS variables.

- ***The MISSING Option***

This option instructs PROC MEANS to consider as valued values missing values for variables in the CLASS statement. If you do not use the MISSING option, PROC MEANS will *exclude observations with a missing CLASS variable value from the analysis.* A related PROC MEANS option, **COMPLETETYPES,** is discussed later on in this paper.

- ***The MLF Option***

The new MLF option permits PROC MEANS to use the primary and secondary format labels to create subgroup combinations when a mulitlabel format is assigned to variable(s) in the CLASS statement. For more information on the new mutilabel formats now available in Version 8, please consult the PROC FORMAT documentation.

- ***The ORDER= Option***

This option specifies the order PROC MEANS will group the levels of the classification variables in the output it generates. (See above for a discussion of the **DESCENDING** option in the CLASS Statement.) The arguments to the **ORDER=** option are: **DATA, FORMATTED, FREQ** and **UNFORMATTED.** Notice that DESCENDING is not an argument to this option,

but is a "standalone" option within the CLASS statement.

- ***The PRELOADFMT Option***

This option instructs the SAS System to pre-load formats in memory before executing the statements in PROC MEANS. In order to use it, you must also specify either the **COMPLETETYPES, EXCLUSIVE or ORDER=DATA** options. Using **PRELOADFMT** in conjunction with, for example, the **COMPLETETYPES** option creates and outputs all combinations of class variables even if the combination does not occur in the input data set. By specifying both **PRELOADFMT** in the CLASS statement and **COMPLETETYPES** option in the PROC MEANS statement, your output will include all combinations of the classification variables, including those with no observations.

The COMPLETETYPES and EXCLUSIVE Options

As discussed above, the **COMPLETETYPES** option instructs the procedure to create all possible combinations of the values of the classification variables, even if that combination does not exist in the data set. It is most often used with CLASS statement options such as **PRELOADFMT.** The **EXCLUSIVE** option is used in conjunction with the **CLASSDATA=** option (also new to Version 8, and discussed below) to include any observations in the input data set whose combination of classification variable values are not in the **CLASSDATA=** data set.

The CLASSDATA= Option

Starting in Version 8 you can specify the name of a data set (temporary or permanent) containing the desired combinations of classification variables that PROC MEANS is to use to filter or to supplement in the input data set. This option is particularly useful if you have many classification variables, and many values of the classification variables, and you only want analyses for some of these combinations. See the PROC MEANS documentation for additional details on how the **CLASSDATA=** option is utilized.

The WAYS Statement

This new PROC MEANS statement specifies the number of ways that the procedure is to create unique combinations of the CLASS statement variables.

We can see how the **WAYS** Statement with the following example. Returning to the customer order file, suppose the marketing analyst wants to create an output data set containing all two-day analyses of the classification variables. (That is, at each unique combination of the CLASS statement variables taken two at a time.) The following PROC MEANS task creates temporary SAS data set B, which contains the desired analysis.

```
proc means noprint
data=order.orderfile2 ;class mailcode
dept_nbr segment; types segment *
status      mailcode * segment
mailcode * dept_nbr;var itmprice
itm_qty;output out=b sum = ;run;
```

The same results would obtain if the analyst used the **WAYS** statement rather than the **TYPES** statement. The following PROC MEANS task, utilizing the **TYPES** statement, requests analyses of the numeric analysis variables at all two-way combinations of the CLASS statement variables.

```
proc means noprint
data=order.orderfile2 ;class mailcode
dept_nbr segment;
WAYS 2;var itmprice itm_qty;output
out=b sum = ;run;
```

Identifying Extreme Values of Analysis Variables using the IDGROUP Option

This new OUTPUT statement option combines and extends the features of the **ID** statement, the **IDMIN** option in the PROC MEANS statement and the **MAXID** and **MINID** options in the Output statement so that you can create an output data set containing variables identifying multiple extreme values of the analysis variables,

Here is an example utilizing the IDGROUP option on the electrical utility data set shown previously.

```
proc means
data=electric.elec_v8 noprint nway;
class transformer;
var total_revenue ;
output out=c
  idgroup (max(total_revenue)
out[2] (total_revenue)=maxrev)
  idgroup (min(total_revenue)
out[2] (total_revenue)=minrev)
```

```
sum= mean= /autoname;
```

In this example, output temporary data set C will contain the SUM and MEAN of analysis variable TOTAL_REVENUE. Using the **AUTONAME** option, discussed above, the names of these variables in the output data set will be TOTAL_REVENUE_SUM and TOTAL_REVENUE_MEAN. The **IDGROUP** options instruct PROC MEANS to also determine the two largest (MAX) and smallest (MIN) values of analysis variable TOTAL_REVENUE, at each value of the single classification variable in the CLASS statement, and output those values in to variables called MINREV_1, MINREV_2, MAXREV_1 and MAXREV_2. The **OUT[2]** argument within each **IDGROUP** option specifies the number of extreme values to output. You can use select from 1 to 100 extreme values to output.

Before using this new feature, you should carefully read the PROC MEANS documentation regarding the **MAXID** and **MINID** syntax. If you are not careful, you can create output variables with the same name! If that occurs, only the first variable with the same name will appear in the output data set. You can avoid this potential outcome by using the previously discussed **AUTONAME** option.

Additional Changes and Enhancements to PROC MEANS

There are a number of other changes and enhancements to PROC MEANS. This paper has shown you what, in my experience, I feel the most important and useful new features have been added in Version 8. You can obtain a list of all enhancements to PROC MEANS from the SAS Institute web site. The complete URL for this topic is:

http://www.sas.com/service/library/onlinedoc/v8/whatsnew/tw5508/z1335349.htm

Acknowledgements

Thanks to Robert Ray of SAS Institute's BASE Information Technology group for his insights in to PROC MEANS and many of the enhancements added to it in Version 8. Also, thanks to the many people who have attended my "Summarizing and Reporting Data Using the SAS System" seminar who have made comments or asked questions that challenged me to learn more about PROC MEANS.

Author contact

Andrew H. Karp
President
Sierra Information Services, Inc.
19229 Sonoma Highway PMB 264
Sonoma, California 94115 USA
707 996 7380
SierraInfo@AOL.COM
www.SierraInformation.com

Copyright

Anyone Can Learn PROC TABULATE

Lauren Haworth, Genentech, Inc., San Francisco

➢ ABSTRACT

SAS® Software provides hundreds of ways you can analyze your data. You can use the DATA step to slice and dice your data, and there are dozens of procedures that will process your data and produce all kinds of statistics. But odds are that no matter how you organize and analyze your data, you'll end up producing a report in the form of a table.

This is why every SAS user needs to know how to use PROC TABULATE. While TABULATE doesn't do anything that you can't do with other PROCs, the payoff is in the output. TABULATE computes a variety of statistics, and it neatly packages the results in a single table.

Unfortunately, TABULATE has gotten a bad rap as being a difficult procedure to learn. This paper will prove that if you take things step by step, anyone can learn PROC TABULATE.

This paper is based on Version 8, but all of the examples save the last one will also work in Version 6.

➢ INTRODUCTION

This paper will start out with the most basic one-dimensional table. We will then go on to two-dimensional tables, tables with totals, and finally, three-dimensional tables. By the end of this paper, you will be ready to build most basic TABULATE tables.

➢ ONE-DIMENSIONAL TABLES

To really understand TABULATE, you have to start very simply. The simplest possible table in TABULATE has to have three things: a PROC TABULATE statement, a TABLE statement, and a CLASS or VAR statement. In this example, we will use a VAR statement. Later examples will show the CLASS statement.

The PROC TABULATE statement looks like this:

```
PROC TABULATE DATA=TEMP;
```

The second part of the procedure is the TABLE statement. It describes which variables to use and how to arrange the variables. This first table will have only one variable, so you don't have to tell TABULATE where to put it. All you have to do is list it in the TABLE statement. When there is only one variable, you get a one-dimensional table.

```
PROC TABULATE DATA=TEMP;
  TABLE RENT;
RUN;
```

If you run this code as is, you will get an error message because TABULATE can't figure out whether the variable RENT is intended as an *analysis* variable, which is used to compute statistics, or a *classification* variable, which is used to define categories in the table.

In this case, we want to use rent as the analysis variable. We will be using it to compute a statistic. To tell TABULATE that RENT is an analysis variable, you use a VAR statement. The syntax of a VAR statement is simple: you just list the variables that will be used for analysis. So now the syntax for our PROC TABULATE is:

```
PROC TABULATE DATA=TEMP;
  VAR RENT;
  TABLE RENT;
RUN;
```

The result is the table shown below. It has a single column, with the header RENT to identify the variable, and the header SUM to identify the statistic. There is just a single table cell, which contains the value for the sum of RENT for all of the observations in the dataset TEMP.

Rent
Sum
264514.00

➢ ADDING A STATISTIC

The previous table shows what happens if you don't tell TABULATE which statistic to use. If the variable in your table is an analysis variable, meaning that it is listed in a VAR statement, then the statistic you will get by default is the sum. Sometimes the sum will be the statistic that you want. Most likely, sum isn't the statistic that you want.

To specify the statistic for a PROC TABULATE table, you modify the TABLE statement. You list the statistic right after the variable name. To tell TABULATE that the statistic MEAN should be applied to the variable RENT, you use an asterisk to link the variable name to the statistic keyword. The asterisk is a TABULATE *operator*. Just as you use an asterisk as an operator when you want to multiply 2 by 3 (2*3), you use an asterisk when you want to apply a statistic to a variable.

```
PROC TABULATE DATA=TEMP;
  VAR RENT;
  TABLE RENT*MEAN;
RUN;
```

The output with the new statistic is shown below. Note that the variable name at the top of the column heading has remained unchanged. However, the statistic name that is shown in the second line of the heading now says

"Mean." In addition, the value shown in the table cell has changed from the sum to the mean.

Rent
Mean
1335.93

➢ ADDING ANOTHER STATISTIC

Each of the tables shown so far was useful, but the power of PROC TABULATE comes from being able to combine several statistics and/or several variables in the same table. TABULATE does this by letting you specify a series of "tables" within a single large table. We're going to add a "table" showing the number of observations to our table showing the mean rent.

The first part of our combined table is the code we used before to compute mean rent.

```
PROC TABULATE DATA=TEMP;
  VAR RENT;
  TABLE RENT*MEAN;
RUN;
```

Next, we can add similar code to our TABLE statement to get the number of observations. To add this statistic to the first table, all you do is combine the code for the mean ("RENT*MEAN") with the code you would used to get the number of observations ("RENT*N"). The code for the two "tables" is combined by using a space between the two statements. The space operator tells TABULATE that you want to add another column to your table.

```
PROC TABULATE DATA=TEMP;
  VAR RENT;
  TABLE RENT*N RENT*MEAN;
RUN;
```

The resulting table is shown below.

Rent	Rent
N	Mean
198.00	1335.93

Note that the additional statistic is shown as an additional column in the table. When SAS is creating a one-dimensional table, additional variables, statistics, and categories are always added as new columns.

➢ USING PARENTHESES

While we're just building a simple table, other tables can get complex in a hurry. To keep your table code easy to read, it's helpful to simplify it as much as possible. One thing you can do is use parentheses to avoid repeating elements in row or column definitions.

For example, instead of defining the table statement like this:

```
TABLE RENT*N RENT*MEAN;
```

It can be defined like this:

```
TABLE RENT*(N MEAN);
```

The resulting table is shown below.

Rent	
N	Mean
198.00	1335.93

Note that not only is the table definition easier to read, but the table headings have also been simplified. Now the "Rent" label is not repeated over each column, but rather is listed once as an overall heading. Using parentheses will simplify both your table definitions and your output.

➢ ADDING A CLASSIFICATION VARIABLE

After seeing the tables we've built so far in this chapter, you're probably asking yourself, "Why use PROC TABULATE? Everything I've seen so far could be done with a PROC MEANS."

One answer to this question is classification variables. By specifying a variable to categorize your data, you can produce a concise table that shows values for various subgroups in your data. For example, wouldn't it be more interesting to look at mean rent if it were broken down by city? [1]

To break down rent by city, we will use city as a classification variable. Just as we used a VAR statement to identify our analysis variable, we use a CLASS statement to identify a classification variable. By putting the variable CITY in a CLASS statement, we are telling TABULATE that the variable will be used to identify categories of the data.

The other thing we have to do to our code is tell TABULATE where to put the classification variable CITY in the table. We do this by again using the asterisk operator. By adding another asterisk to the end of the TABLE statement, and following it with the variable name CITY, TABULATE knows that CITY will be used to categorize the mean values of RENT.

```
PROC TABULATE DATA=TEMP;
  CLASS CITY;
  VAR RENT;
  TABLE RENT*MEAN*CITY;
RUN;
```

[1] The sample data in this paper is from an informal survey of apartment rents in three cities: Portland, Oregon, where the author used to live; San Francisco, where the author currently lives; and Long Beach, since that's where the paper will be presented.

The resulting table is shown below. Now the column headings have changed. The variable name Rent and the statistic name Mean are still there, but under the statistic label there are now three columns. Each column is headed by the variable label "City" and the category name Portland," "San Francisco," and "Long Beach." The values shown in the table cells now represent subgroup means.

Rent		
Mean		
City		
Portland	San Francisco	Long Beach
931.44	2282.22	1116.00

➢ TWO-DIMENSIONAL TABLES

You probably noticed that our example table is not very elegant in appearance. That's because it only takes advantage of one dimension. It has multiple columns, but only one row. It is much more efficient to build tables that have both rows and columns. You can fit more information on a page, and the table looks better, too.

The easiest way to build a two-dimensional table is to build it one dimension at a time. First, we'll build the columns, and then we'll add the rows.

For this first table, we'll keep things simple. This is the table we built in a previous example. It has two columns: one showing the number of observations for RENT and another showing the mean of RENT.

```
PROC TABULATE DATA=TEMP;
  VAR RENT;
  TABLE RENT*(N MEAN);
RUN;
```

This table is shown below.

Rent	
N	Mean
198.00	1335.93

To turn this table into a two-dimensional table, we will add another variable to the TABLE statement. In this case, we want to add rows that show the N and MEAN of RENT for different sizes of apartments.

To add another dimension to the table, you use a comma as an operator. All you do is put a comma between the row variable(s) and the column variable(s).

If a TABLE statement has no commas, then it is assumed that the variables and statistics are to be created as columns. If a TABLE statement has two parts, separated by a comma, then TABULATE builds a two-dimensional table using the first part of the TABLE statement as the rows and the second part of the TABLE statement as the columns.

So to get a table with rent as the columns and number of bedrooms as the rows, we just need to add a comma and the variable BEDROOMS. Since we want to add BEDROOMS as a row, we list it before the rest of the TABLE statement. If we wanted to add it as a column, we'd add it to the end of the TABLE statement.

```
PROC TABULATE DATA=TEMP;
  VAR RENT;
  CLASS BEDROOMS;
  TABLE BEDROOMS, RENT*N RENT*MEAN;
RUN;
```

This table is shown below.

	Rent	Rent
	N	Mean
Bedrooms		
1 Bedroom	92.00	1180.35
2 Bedrooms	106.00	1470.96

By the way, there is a limitation on which variables you can use in a two-dimensional table. You can't have a cross-tabulation of two analysis variables. A two-dimensional table must have at least one classification variable (i.e., you must have a CLASS statement). If you think about it, this makes sense. A table of mean rent by mean bedrooms would be meaningless, but a table of mean rent by categories of bedrooms makes perfect sense.

➢ ADDING CLASSIFICATION VARIABLES ON BOTH DIMENSIONS

The previous example showed how to reformat a table from one to two dimensions, but it did not show the true power of two-dimensional tables. With two dimensions, you can classify your statistics by two different variables at the same time.

To do this, you put one classification variable in the row dimension and one classification in the column dimension. The previous example had bedrooms displayed in rows, and rent as the column variable. In this new table, we will add city as an additional column variable. Instead of just displaying the mean rent for each number of bedrooms, we will display the statistic broken down city.

So in the following code, we leave BEDROOMS as the row variable, and we leave RENT in the column dimension. The only change is to add CITY to the column dimension using the asterisk operator. This tells TABULATE to break down each of the column elements into categories by city.

```
PROC TABULATE DATA=TEMP;
  VAR RENT;
  CLASS BEDROOMS CITY;
  TABLE BEDROOMS, RENT*CITY*MEAN;
RUN;
```

This table is shown below. Notice how the analysis variable RENT remains as the column heading, and MEAN remains as the statistic, but now there are additional column headings to show the three categories of CITY.

	Rent		
	City		
	Portland	San Francisco	Long Beach
	Mean	Mean	Mean
Bedrooms			
1 Bedroom	800.08	2063.83	955.23
2 Bedrooms	1050.29	2483.81	1242.92

➢ Adding Another Classification Variable

The previous example showed how to add a classification variable to both the rows and columns of a two-dimensional table. But you are not limited to just one classification per dimension. This next example will show how to display additional subgroups of the data.

In this case, we're going to add availability of off-street parking as an additional row classification. The variable is added to the CLASS statement and to the row dimension of the TABLE statement. It is added using a space as the operator, so we will get rows for number of bedrooms followed by rows for parking availability.

```
PROC TABULATE DATA=TEMP;
  VAR RENT;
  CLASS BEDROOMS CITY PARKING;
  TABLE BEDROOMS PARKING,
     RENT*CITY*MEAN;
RUN;
```

In the results shown below, you can see that we now have two two-dimensional "mini-tables" within a single table. First, we have a table of rent by number of bedrooms and city, and then we have a table showing rent by parking availability and city.

	Rent		
	City		
	Portland	San Francisco	Long Beach
	Mean	Mean	Mean
Bedrooms			
1 Bedroom	800.08	2063.83	955.23
2 Bedrooms	1050.29	2483.81	1242.92
Parking			
No	781.75	1785.30	1060.54
Yes	995.59	2613.50	1154.83

This ability to stack multiple mini-tables within a single table can be a powerful tool for delivering large quantities of information in a user-friendly format.

➢ Nesting the Classification Variables

So far, all we have done is added additional "tables" to the bottom of our first table. We used the space operator between each of the row variables to produce stacked tables.

By using a series of row variables, we can explore a variety of relationships between the variables. In previous table, we could see how rent varies by number of bedrooms and city, and we could see how rent varies by parking availability and city, but we could not see how the number of bedrooms and parking availability interacted to affect rent for each city.

The power of TABULATE comes from being able to look at combinations of categories within a single table. In the following example, we will build a table to look at rent by city for combinations of number of bedrooms and parking availability.

This code is the same as we used for the last example. The only change is that in the row definition, the asterisk operator is used to show that we want to nest the two row variables. In other words, we want to see the breakdown of rents by parking availability within each category of number of bedrooms.

```
PROC TABULATE DATA=TEMP;
  VAR RENT;
  CLASS BEDROOMS CITY PARKING;
  TABLE BEDROOMS*PARKING,
     RENT*CITY*MEAN;
RUN;
```

As you can see in the table below, this code produces nested categories within the row headings. The row headings are now split into two columns. The first column shows number of bedrooms and the second shows parking availability. It is now easier to interpret the interaction of the two variables.

		Rent		
		City		
		Portland	San Francisco	Long Beach
		Mean	Mean	Mean
Bedrooms	Parking			
1 Bedroom	No	707.00	1586.10	896.00
	Yes	843.04	2405.07	984.85
2 Bedrooms	No	856.50	1984.50	1151.94
	Yes	1127.80	2795.88	1324.80

You can also reverse the order of the row variables to look at number of bedrooms by parking availability, instead of parking availability by number of bedrooms. All you do is move PARKING so that it comes before BEDROOMS. TABULATE always produces the nested rows in the order the variables are listed on the TABLE statement.

➢ ADDING TOTALS TO THE ROWS AND COLUMNS

As your tables get more complex, you can help make your tables more readable by adding row and column totals. Totals are quite easy to generate in TABULATE because you can use the ALL variable. This is a built in classification variable supplied by TABULATE that stands for "all observations." You do not have to list it in the CLASS statement because it is a classification variable by definition.

The following code produces a table similar to the previous example, but with the addition of row totals. The statistic is changed to N so that you can see how the totals work.

```
PROC TABULATE DATA=TEMP;
  CLASS BEDROOMS CITY;
  TABLE BEDROOMS, (CITY ALL)*N;
RUN;
```

As you can see from the table below, the table now has row totals.

	City			
	Portland	San Francisco	Long Beach	All
	N	N	N	N
Bedrooms				
1 Bedroom	38.00	24.00	30.00	92.00
2 Bedrooms	42.00	26.00	38.00	106.00

Not only can you use ALL to add row totals, but you can also use ALL to produce column totals. What you do is list ALL as an additional variable in the row definition of the TABLE statement. No asterisk is needed because we just want to add a total at the bottom of the table.

```
PROC TABULATE DATA=TEMP;
  CLASS BEDROOMS CITY;
  TABLE BEDROOMS ALL, CITY*N;
RUN;
```

The resulting table is shown below. Now there are overall totals for each city.

	City		
	Portland	San Francisco	Long Beach
	N	N	N
Bedrooms			
1 Bedroom	38.00	24.00	30.00
2 Bedrooms	42.00	26.00	38.00
All	80.00	50.00	68.00

➢ THREE-DIMENSIONAL TABLES

Now that you have mastered two-dimensional tables, let's add a third dimension. You may be asking yourself: Three dimensions? How do you print a table shaped like a cube?

Actually, a three-dimensional table is not shaped like a cube. It looks like a two-dimensional table, except that it spans multiple pages. A one-dimensional table just has columns. A two-dimensional table has both columns and rows. A three-dimensional table is just a two-dimensional table that is repeated across multiple pages. You print a new page for each value of the page variable.

The hardest part about three-dimensional tables is making sense of the TABLE statement. So the best way to start is with the first two dimensions: the rows and columns. Once you've got that set up correctly, it's relatively easy to add the page variable to expand the table to multiple pages.

For our example, we're going to build another table of rent by city and number of bedrooms, and then we're going to add parking availability as the page variable. We'll end up with two pages of output, the first page will be for apartments without off-street parking, and the second page will be for apartments with off-street parking.

Ignoring the third dimension for now, let's build the basic table. This table has rows showing the number of bedrooms, and columns showing rent by city. The code is as follows:

```
PROC TABULATE DATA=TEMP;
  CLASS BEDROOMS CITY;
  VAR RENT;
  TABLE BEDROOMS,
     (CITY ALL)*RENT*MEAN;
RUN;
```

At this point you should run the code and look the table over carefully to be sure you've got exactly what you want to see in your final table.

	City			
	Portland	San Francisco	Long Beach	All
	Rent	Rent	Rent	Rent
	Mean	Mean	Mean	Mean
Bedrooms				
1 Bedroom	800.08	2063.83	955.23	1180.35
2 Bedrooms	1050.29	2483.81	1242.92	1470.96

The only difference between this table and our final three-dimensional table is that right now, the table is showing results for both categories of parking availability combined. In the final tables, each page will have the results for just one of the two categories.

Assuming the two-dimensional table looks correct, we'll go on to adding the third dimension. When we converted a one-dimensional table to a two-dimensional table, we added the new dimension with a comma operator in the TABLE statement. To change a two-dimensional table to a three-dimensional table, we just add the new variable PARKING to the existing TABLE statement with a comma to separate it from the row and column definitions.

You might think that the following code is the correct way to add the third dimension to the table statement:

```
TABLE BEDROOMS,
  (CITY ALL)*RENT*MEAN, PARKING;
```

However, what this would generate is a table with CITY as the rows, PARKING as the columns, and BEDROOMS as the pages. In order to add the third dimension to a table, you add it at the beginning of the table statements. Remember that this was also true when we added the second dimension to the table. We added rows to the columns by adding a row definition before the column definition.

The correct code for our table is:

```
PROC TABULATE DATA=TEMP;
  CLASS BEDROOMS CITY PARKING;
  VAR RENT;
  TABLE PARKING, BEDROOMS,
     (CITY ALL)*RENT*MEAN;
RUN;
```

The resulting tables are shown below. To save space, both tables are shown on a single page. In reality, TABULATE puts a page break between each of the tables, and the table for apartments without off-street parking would appear on a second page. Notice that each table was automatically given a title that defines the category it represents.

Parking No

	City			
	Portland	San Francisco	Long Beach	All
	Rent	Rent	Rent	Rent
	Mean	Mean	Mean	Mean
Bedrooms				
1 Bedroom	707.00	1586.10	896.00	1040.78
2 Bedrooms	856.50	1984.50	1151.94	1271.45

Parking Yes

	City			
	Portland	San Francisco	Long Beach	All
	Rent	Rent	Rent	Rent
	Mean	Mean	Mean	Mean
Bedrooms				
1 Bedroom	843.04	2405.07	984.85	1254.78
2 Bedrooms	1127.80	2795.88	1324.80	1591.88

➢ MAKING THE TABLE PRETTY

The preceding examples have shown how to create basic tables. They contained all of the needed information, but they were pretty ugly. The next thing you need to learn is a few tricks to clean up your tables.

For example, look at the following code and table:

```
PROC TABULATE DATA=TEMP;
  CLASS BEDROOMS CITY;
  VAR RENT;
  TABLE BEDROOMS,
     (CITY ALL)*RENT*MEAN;
RUN;
```

	City			
	Portland	San Francisco	Long Beach	All
	Rent	Rent	Rent	Rent
	Mean	Mean	Mean	Mean
Bedrooms				
1 Bedroom	800.08	2063.83	955.23	1180.35
2 Bedrooms	1050.29	2483.81	1242.92	1470.96

Notice how the totals column is titled "All." We can make this table more readable by changing that title to "Overall."

To do this, we just attach the label to the keyword in the TABLE statement with an equal sign operator. This operator is used to apply labels to variables and statistics. The new code reads as follows:

```
PROC TABULATE DATA=TEMP;
  CLASS BEDROOMS CITY;
  VAR RENT;
  TABLE BEDROOMS,
     (CITY ALL='Overall')*RENT*MEAN;
RUN;
```

This code produces the following output:

	City			
	Portland	San Francisco	Long Beach	Overall
	Rent	Rent	Rent	Rent
	Mean	Mean	Mean	Mean
Bedrooms				
1 Bedroom	800.08	2063.83	955.23	1180.35
2 Bedrooms	1050.29	2483.81	1242.92	1470.96

Another thing that could be improved about this table is getting rid of the excessive column headings. This table is four lines deep in headings. For starters, we can get rid of the label "City." With values like "San Francisco," "Portland," and "Long Beach," it's obvious that this table is referring to cities. We don't need the extra label.

To get rid of it, we attach a blank label. A blank label is two quotes with a single blank space in between. The blank label is added the same way we added the "Overall" label in the last example. The blank label is attached to the variable using an equal sign.

```
PROC TABULATE DATA=TEMP;
  CLASS BEDROOMS CITY;
  VAR RENT;
  TABLE BEDROOMS,
     (CITY=' ' ALL='Overall')*
     RENT*MEAN;
RUN;
```

The revised output is shown below.

	Portland	San Francisco	Long Beach	Overall
	Rent	Rent	Rent	Rent
	Mean	Mean	Mean	Mean
Bedrooms				
1 Bedroom	800.08	2063.83	955.23	1180.35
2 Bedrooms	1050.29	2483.81	1242.92	1470.96

This looks much better, but there are still too many column headings. We can get rid of two more. Notice how each column is headed by "RENT" and "MEAN."

We can make these labels go away by setting them to a blank label as in the previous example. Also, the row heading "Number of Bedrooms" is not needed, since the categories of "1 Bedroom" and "2 Bedrooms" are self-explanatory. So the variable BEDROOMS is also assigned a blank label. The revised code is shown below.

```
PROC TABULATE DATA=TEMP;
  CLASS BEDROOMS CITY;
  VAR RENT;
  TABLE BEDROOMS=' ',
     (CITY=' ' ALL='Overall')*
     RENT=' '*MEAN=' ';
RUN;
```

Now we have a nice simple table, which is shown below.

	Portland	San Francisco	Long Beach	Overall
1 Bedroom	800.08	2063.83	955.23	1180.35
2 Bedrooms	1050.29	2483.81	1242.92	1470.96

However, we also have a problem. If we take the Rent and Mean labels away, then there is no label in the table to describe the analysis variable or statistic.

We could add a title above the table to hold this information, but there's a better way. Notice how in all of our tables there's a big empty box above the rows and to the left of the column headings. This space is available to us. The following code uses the BOX= option to hold a label describing our table.

```
PROC TABULATE DATA=TEMP;
  CLASS BEDROOMS CITY;
  VAR RENT;
  TABLE BEDROOMS,
     (CITY=' ' ALL='Overall')*
     RENT=' '*MEAN=' '
     / BOX='Average Rent';
RUN;
```

The output is shown below:

Average Rent	Portland	San Francisco	Long Beach	Overall
Bedrooms				
1 Bedroom	800.08	2063.83	955.23	1180.35
2 Bedrooms	1050.29	2483.81	1242.92	1470.96

At this point the table is looking pretty good. There's just one more thing we should do to the table. Since the numbers being reported in the table are rents, it would make the table easier to read if they were formatted with dollar signs. Also, we can round these off to even dollars, that's enough precision for this table.

To change the format of the table cells, we use the FORMAT= option on the TABLE statement. You can use this to apply any valid SAS format. The revised code calls for values to be formatted with dollar signs and commas, and eliminates the display of decimal spaces. The width of 12 was chosen because the widest column label ("Long Beach") is 10 characters wide, and we want an extra space on either side of the label. When you specify a format for the table values, you are also specifying the width of the column that holds that value.

```
PROC TABULATE DATA=TEMP
     FORMAT=DOLLAR12.;
  CLASS BEDROOMS CITY;
  VAR RENT;
  TABLE BEDROOMS=' ',
     (CITY=' ' ALL='Overall')*
     RENT=' '*MEAN=' '
     / BOX='Average Rent';
RUN;
```

The revised output is shown below:

Average Rent	Portland	San Francisco	Long Beach	Overall
1 Bedroom	$800	$2,064	$955	$1,180
2 Bedrooms	$1,050	$2,484	$1,243	$1,471

➢ CREATING HTML OUTPUT

Once you know how to create TABULATE tables, it is a simple matter to turn them into HTML pages that can be posted on your web site.

Using Version 6, all you have to do is the HTML conversion macros that come with your SAS software. Then, you just add a macro call before and after your TABULATE code and SAS will generate the HTML output for you.

The Version 6 code is as follows:

```
%TAB2HTM (CAPTURE=ON, RUNMODE=B);
OPTIONS FORMCHAR='8283848586878889 8A8B8C'X;
PROC TABULATE DATA=TEMP
     FORMAT=DOLLAR12.;
  CLASS BEDROOMS CITY;
  VAR RENT;
  TABLE BEDROOMS=' ',
     (CITY=' ' ALL='Overall')*
     RENT=' '*MEAN=' '
     / BOX='Average Rent';
RUN;
%TAB2HTM(CAPTURE=OFF, RUNMODE=B,
  OPENMODE=REPLACE, HTMLFILE=SAMPLE.HTML);
```

The output is shown below:

Average Rent	Portland	San Francisco	Long Beach	Overall
1 Bedroom	$800	$2,064	$955	$1,180
2 Bedrooms	$1,050	$2,484	$1,243	$1,471

Under Version 8, the process of outputting a TABULATE table to HTML is nearly identical. Instead of calling the macro before and after your code, you call on the Output Delivery System (ODS).

```
ODS HTML BODY='SAMPLE.HTML';
PROC TABULATE DATA=TEMP
     FORMAT=DOLLAR12.;
  CLASS BEDROOMS CITY;
  VAR RENT;
  TABLE BEDROOMS=' ',
     (CITY=' ' ALL='Overall')*
     RENT=' '*MEAN=' '
     / BOX='Average Rent';
RUN;
ODS HTML CLOSE;
```

The output is shown below:

Average Rent	Portland	San Francisco	Long Beach	Overall
1 Bedroom	$800	$2,064	$955	$1,180
2 Bedrooms	$1,050	$2,484	$1,243	$1,471

You can see that the result is similar to the Version 6 table, but it uses more colors, and is more stylish.

➢ CHANGING THE STYLE

If you're using Version 8, you also have the option of changing the look of your results. By adding a STYLE= option to your ODS statement, you can change from the default style (shown above) to one of the following styles. The resulting output is shown below the code for each style

```
ODS HTML BODY='SAMPLE.HTML'
STYLE=XXX;
```

The following examples show a few of the styles that ship with Version 8.

STYLE=BARRETTSBLUE

Average Rent	Portland	San Francisco	Long Beach	Overall
1 Bedroom	$800	$2,064	$955	$1,180
2 Bedrooms	$1,050	$2,484	$1,243	$1,471

STYLE=BRICK

Average Rent	Portland	San Francisco	Long Beach	Overall
1 Bedroom	$800	$2,064	$955	$1,180
2 Bedrooms	$1,050	$2,484	$1,243	$1,471

STYLE=BROWN

Average Rent	Portland	San Francisco	Long Beach	Overall
1 Bedroom	$800	$2,064	$955	$1,180
2 Bedrooms	$1,050	$2,484	$1,243	$1,471

STYLE=D3D

Average Rent	Portland	San Francisco	Long Beach	Overall
1 Bedroom	$800	$2,064	$955	$1,180
2 Bedrooms	$1,050	$2,484	$1,243	$1,471

STYLE=MINIMAL

Average Rent	Portland	San Francisco	Long Beach	Overall
1 Bedroom	$800	$2,064	$955	$1,180
2 Bedrooms	$1,050	$2,484	$1,243	$1,471

STYLE=STATDOC

Average Rent	Portland	San Francisco	Long Beach	Overall
1 Bedroom	$800	$2,064	$955	$1,180
2 Bedrooms	$1,050	$2,484	$1,243	$1,471

➢ CONCLUSIONS

At this point, you should be comfortable with the basics of producing a table using PROC TABULATE. You should be able to produce a simple table with totals, be able to clean it up a bit, and be able to create HTML output.

This should be enough to get you going producing tables with your own data. And now that you're more comfortable with the procedure, you should be able to use the TABULATE manual and other books and papers to learn more advanced techniques.

➢ ACKNOWLEDGEMENTS

SAS is a registered trademark or trademark of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

➢ CONTACTING THE AUTHOR

Please direct any questions or feedback to the author at:
info@laurenhaworth.com

The Utter "Simplicity?" of the TABULATE Procedure - The Final Chapter?

Dan Bruns, Tennessee Valley Authority, Chattanooga, TN

IN THE BEGINNING

Well, here we are again TABULATE fans. I believe I have exhausted this topic (to DEATH some folks say), so I thought I would put it to rest in this FINAL CHAPTER with a paper on the truly advanced features of the TABULATE procedure. The problem is these advanced features are anything but simple. In this tutorial we look at some simpler advanced features, like FORMCHAR, column and row titling, and formatting, and then the one that is really a bear to understand – percentages (PCTN and PCTSUM). Some of the new Version 8 features will also be covered.

The output from a CONTENTS procedure below is just so you know a little about the dataset we will be working with.

CONTENTS PROCEDURE

Data Set Name:	SASDATA.CLASS	Observations:	27
Member Type:	DATA	Variables:	5
Engine:	V60x	Indexes:	0
Created:	9:14 Wednesday, Sep 19	Observation Length:	48
Last Modified:	11:40 Tuesday, Feb 5	Deleted Observations:	0
Data Set Type:		Compressed:	NO
		Reuse Space:	NO

-----Alphabetic List of Variables and Attributes-----

#	Variable	Type	Len	Pos	Format	Label
4	DATE	Num	8	32	DATE5.	Class Date
2	LOC	Char	1	25		Location
1	NAME	Char	25	0		
3	ORG	Char	6	26		Org
5	SCORE	Num	8	40	5.1	Final Exam Score

SOME BASICS

Here are few basic examples and their totally different looking outputs by simply changing where and how the variables are coded. If these are beyond your current proficiency with TABULATE, see my Beginning Tutorial paper in the SUGI 16 and 20 proceedings and my Advanced Tutorial paper in the SUGI 21 proceedings and hang on to your hat because I'm starting from here and assuming you understand this much.

```
PROC TABULATE  DATA=CLASS ;
     CLASS ORG LOC DATE;
     VAR   SCORE;
     TABLE  ORG, LOC*SCORE*(N MEAN)*F=5.1;
```

	Location					
	A		B		C	
	Final Exam Score		Final Exam Score		Final Exam Score	
	N	MEAN	N	MEAN	N	MEAN
Org						
Energy	4.0	84.4	2.0	96.4	.	.
Mgt S	2.0	73.4	1.0	85.4	7.0	84.8
Power	4.0	89.0	2.0	70.7	3.0	79.8

Here you see a column for the count (N) and mean of SCORE for each location.

```
TABLE  ORG*LOC,
         SCORE*(N MEAN MAX PCTN)*F=5.1;
```

		Final Exam Score			
		N	MEAN	MAX	PCTN
Org	Location				
Energy	A	4.0	84.4	93.0	16.0
	B	2.0	96.4	100.0	8.0
Mgt S	A	2.0	73.4	99.4	8.0
	B	1.0	85.4	85.4	4.0
	C	7.0	84.8	98.3	28.0
Power	A	4.0	89.0	99.1	16.0
	B	2.0	70.7	90.0	8.0
	C	3.0	79.8	93.6	12.0

Here are the same numbers from the previous output. Location was simply moved from the column expression to the row expression.

```
TABLE  ORG LOC,
         SCORE*(N MEAN MAX PCTN)*F=5.1;
```

	Final Exam Score			
	N	MEAN	MAX	PCTN
Org				
Energy	6	88.4	100.0	24.0
Mgt S	10	82.6	99.4	40.0
Power	9	81.9	99.1	36.0
Location				
A	10	84.0	99.4	40.0
B	5	83.9	100.0	20.0
C	10	83.3	98.3	40.0

Here are two tables in one: the N, MEAN, MAX, and PCTN statistics in the column expression allows you to use the row expression to see a summary by two different variables (ORG and LOC) in one table.

```
TABLE  ORG ALL,
         (LOC ALL)*SCORE*(N MEAN)*F=5.1;
```

	Location							
	A		B		C		ALL	
	Final Exam Score		Final Exam Score		Final Exam Score		Final Exam Score	
	N	MEAN	N	MEAN	N	MEAN	N	MEAN
Org								
Energy	4	84.4	2	96.4	.	.	6	88.4
Mgt S	2	73.4	1	85.4	7	84.8	10	82.6
Power	4	89.0	2	70.7	3	79.8	9	81.9
ALL	10	84.0	5	83.9	10	83.3	25	83.7

What in the world happened in this last example?

There's no variable named ALL in the dataset?

That's right, but ALL is kind of like a built-in class variable that can be specified accumulate totals for the entire row and/or column. In the above example it was used in the row expression to produce a set of totals after the ORG rows. If you placed it before the ORG variable (i.e. ALL ORG) you would get the totals as the first row of the table. The use of ALL in the column expression caused it to produce a column after the LOC columns. Also notice since it was grouped with LOC and then nested, the column contains the totals for all locations using the same statistics.

You may not be able to tell from this example, but TABULATE computes true statistics (i.e. MEAN above). That means it **DOES NOT** add-up the means from the tables and then divide by the number of table entries; it accumulates each observations value and divides by the number of observations.

TITLES AND LABELS

You can see that to have TABULATE put descriptive titles or labels for the variables you simply need to assign meaningful labels to them. You can either do this in earlier steps that create the dataset or with a LABEL statement in the PROC step. But what about the statistics and ALL? Simply attach a descriptive label to ANY variable or statistic right in the TABLE statement. Follow it with an equals sign (=) and a quoted label ('This is a Label') just like you do in a LABEL statement. Or if you want to use a certain label for every use of the statistic, use the KEYLABEL statement which looks exactly like the LABEL statement except you use the statistic's name instead of a variable name. Here is an example of doing both.

```
TABLE
       ORG ALL,
       (LOC ALL='Row Totals')
           *SCORE*(N MEAN)*F=5.1
       / BOX='SESUG 2K';
KEYLABEL
       N='Count'
       MEAN='Mean'
       ALL='Total' ;
```

SESUG 2K	Location							
	A		B		C		Row Totals	
	Final Exam Score		Final Exam Score		Final Exam Score		Final Exam Score	
	Count	Avg	Count	Avg	Count	Avg	Count	Avg
Department								
Energy	4	84.4	2	96.4	.	.	6	88.4
Mgt S	2	73.4	1	85.4	7	84.8	10	82.6
Power	4	89.0	2	70.7	3	79.8	9	81.9
Total	10	84.0	5	83.9	10	83.3	25	83.7

The above example has another tables option specified (BOX=) that specifies what to put in the upper-left corner box of the table.

In the following example we added the MISSING and NOSEPS options to the PROC statement to have TABULATE treat missing values as a valid category (which it does not do by default) and remove the separation lines between the rows. I also specified some table options: BOX=SCORE to label the upper-left box with the SCORE variable's label; and MISSTEXT='None' to label missing values in the tables with the text 'None' instead of the standard period.

```
PROC TABULATE  DATA=CLASS
            MISSING  NOSEPS ;
    CLASS ORG LOC DATE;
    VAR   SCORE;
/*   TITLES & LABELS */
TABLE
    ORG ALL='--- Totals ---',
    (LOC ALL='Row Totals')
         *(SCORE*MEAN=' '*F=5.1)
    / BOX=SCORE  ROW=FLOAT
           MISSTEXT='None';
```

Final Exam Averages	Location				Row Totals
	A	B	C	ls	
	Final Exam Aver-ages	Final Exam Aver-ages	Final Exam Aver-ages	Final Exam Aver-ages	Final Exam Aver-ages
Department					
Energy	None	84.4	96.4	None	88.4
Mgt S	None	73.4	85.4	84.8	82.6
Power	None	89.0	70.7	79.8	81.9
--- Totals ---	None	84.0	83.9	83.3	83.7

Notice that since the MEAN label was blank and the ROW=FLOAT was specified, that no space was wasted for it. Now as one final farewell to labeling, a table that doesn't look like a table.

```
PROC TABULATE  DATA=CLASS
          MISSING  NOSEPS
          FORMCHAR='                ';
    CLASS ORG LOC DATE;
    VAR   SCORE;
    TABLE
        ORG ALL='--- Totals ---',
        (LOC ALL='Row Totals')
            *(SCORE=' '*MEAN=' '*F=6.1)
        / BOX=SCORE  ROW=FLOAT
              MISSTEXT='None';
```

Final Exam Averages	Location				Dept Totals
	A	B	C		
Department					
	None	None	None	None	None
Energy	None	84.4	96.4	None	88.4
Mgt S	None	73.4	85.4	84.8	82.6
Power	None	89.0	70.7	79.8	81.9
--- Totals ---	None	84.0	83.9	83.3	83.7

By simply adding the FORMCHAR= option to the PROC statement and specifying 16 blanks, you remove all the lines from around the table. If you have access to a laser printer you can also use characters that form "solid" lines around your table. This example is the specification needed on an OS/390 mainframe (MVS).

```
PROC TABULATE  DATA=CLASS
    MISSING  NOSEPS
     FORMCHAR='FABFACCCBCEB8FECABCB
               BB4E7E4F60AFE04C6E40'X ;
    CLASS ORG LOC DATE;
    VAR   SCORE;
    TABLE
        ORG ALL='--- Totals ---',
        (LOC ALL='Row Totals')
            *(SCORE=' '*MEAN=' '*F=6.1)
        / BOX=SCORE  ROW=FLOAT
              MISSTEXT='None';
```

SUBTOTALING

The only real trick to doing subtotaling is the nesting of ALL in the row expression.

```
TABLE
   ORG*(LOC ALL='Loc Subtotal') ALL='Org Total',
   SCORE='Average Final Exam Score'
        *MEAN=' '*F=6.1
   /RTS=25 BOX=SCORE
           ROW=FLOAT MISSTEXT='None';
```

Final Exam Averages		Avg
Department	Location	
	A	None
	**Subtotal	None
Energy	Location	
		None
	A	84.4
	B	96.4
	**Subtotal	88.4
Mgt S	Location	
	A	73.4
	B	85.4
	C	84.8
	**Subtotal	82.6
Power	Location	
	A	89.0
	B	70.7
	C	79.8
	**Subtotal	81.9
Dept Total		83.7

Above we see the nesting of (LOC ALL) in ORG. That tells TABULATE to concatenate an ALL row after all the LOC rows for each ORG value.

PERCENTAGES

In its simplest form the PCTN or PCTSUM is just another statistic like N or MEAN you can request.

```
PROC TABULATE  DATA=CLASS   FORMAT=6.1 ;
     CLASS ORG LOC DATE;
     VAR   SCORE;
     TABLE  ORG, (LOC ALL)*(N*F=3.0 PCTN);
```

	Location							
	A		B		C		ALL	
	N	PCTN	N	PCTN	N	PCTN	N	PCTN
Org								
Energy	6	22.2	2	7.4	.	.	8	29.6
Mgt S	2	7.4	1	3.7	7	25.9	10	37.0
Power	4	14.8	2	7.4	3	11.1	9	33.3

Unless specified, the percentage is computed based on all the observations in the dataset. Notice that the PCTN under the ALL column does not add up to 100 due to rounding.

To specify how the percentage is computed you simply attach a denominator specification to PCTN or PCTSUM using the inequality signs less-than (<) and greater-than (>). ***The real trick to understanding how the denominator specification works is to remember you are telling TABULATE what the denominator is to divide into the N or SUM value, or <u>what you are dividing by.</u>***

```
TABLE  ORG, (LOC ALL)*(N*F=3.0 PCTN<ORG>);
```

	Location							
	A		B		C		ALL	
	N	PCTN	N	PCTN	N	PCTN	N	PCTN
Org								
Energy	6	50.0	2	40.0	.	.	8	29.6
Mgt S	2	16.7	1	20.0	7	70.0	10	37.0
Power	4	33.3	2	40.0	3	30.0	9	33.3

The above example shows the row expression in the denominator specification. Notice that none of the counts (N) have changed but the PCTN values have because the denominator has changed from the entire dataset (27 observations) to all the observations for ORG within that columns (LOC) value. Observe that since PCTN is nested in LOC that the denominator specification is saying to divide each cell under that location by the total number of observations that are in that location. So why do you specify the row expression? Because that is simply telling TABULATE which number of observations to total. So, in the above example, we see that location A cells are divided by 12, the total of all the ORG observations in that location. For location B, we see each cell is divided by 5, the total of all the ORG observations in that location. And for location C, we see each cell is divided by 10, the total of all the ORG observations in that location. And for the ALL column we see each cell is divided by the total of all the ORG observations in all the locations.

Here is a handy rule-of-thumb:

To get percentages by column, use the row expression; to get percentages by row, use the column expression.

```
TABLE  ORG,
       (LOC ALL)*(N*F=3.0 PCTN<LOC ALL>);
```

	Location							
	A		B		C		ALL	
	N	PCTN	N	PCTN	N	PCTN	N	PCTN
Org								
Energy	6	75.0	2	25.0	.	.	8	100.0
Mgt S	2	20.0	1	10.0	7	70.0	10	100.0
Power	4	44.4	2	22.2	3	33.3	9	100.0

Notice in the above example that the entire column expression is coded as the denominator specification. If you don't, strange results or even errors can occur. As before, you are simply telling TABULATE which number of observations to total. So, in the above example, we see that organization 'Energy' cells are divided by 8, the total of all the LOC observations in that organization. For organization 'Mgt S' we see each cell is divided by 10, the total of all the LOC observations in that organization. For organization 'Power' we see each cell is divided by 9, the total of all the LOC observations in that organization. And for the ALL column we see each cell is divided by the total of all the LOC observations in that organization, thus the 100 percent.

```
TABLE  ORG,
      (LOC ALL)*SCORE*
           (SUM*F=5.1 PCTSUM<LOC ALL>);
```

	Location							
	A		B		C		ALL	
	Final Exam Score		Final Exam Score		Final Exam Score		Final Exam Score	
	SUM	PCTSUM	SUM	PCTSUM	SUM	PCTSUM	SUM	PCTSUM
Org								
Energy	337.6	63.7	192.8	36.3	.	.	530.4	100.0
Mgt S	146.7	17.8	85.4	10.3	593.5	71.9	825.6	100.0
Power	356.2	48.3	141.3	19.2	239.4	32.5	736.9	100.0

This example is just to show that the PCTSUM works in the same way. (The summing of exams scores doesn't seem to make much sense, but it is the only numeric variable in the dataset.)

The following examples show that the same rules apply for nesting variables and using ALL.

```
PROC TABULATE  DATA=CLASS
               FORMAT=6.1 NOSEPS;
     CLASS ORG LOC DATE;
     VAR   SCORE;
     TABLE  ORG*DATE,
           (LOC ALL)*(N*F=3.0 PCTN<ORG*DATE>);
```

		Location							
		A		B		C		ALL	
		N	PCTN	N	PCTN	N	PCTN	N	PCTN
Org	Class								
Energy	Date								
	07APR	1	8.3	.	.	.	.	1	3.7
	03MAY	1	8.3	1	20.0	.	.	2	7.4
	22JUN	1	8.3	.	.	.	.	1	3.7
	12OCT	3	25.0	1	20.0	.	.	4	14.8
Mgt S	07APR	1	8.3	.	.	.	.	1	3.7
	03MAY	.	.	1	20.0	.	.	1	3.7
	22JUN	.	.	.	.	4	40.0	4	14.8
	12OCT	1	8.3	.	.	3	30.0	4	14.8
Power	07APR	1	8.3	2	40.0	.	.	3	11.1
	03MAY	1	8.3	.	.	.	.	1	3.7
	22JUN	1	8.3	.	.	3	30.0	4	14.8
	12OCT	1	8.3	.	.	.	.	1	3.7

```
TABLE  ORG*DATE,
      (LOC ALL)*(N*F=3.0 PCTN<LOC ALL>);
```

		Location							
		A		B		C		ALL	
		N	PCTN	N	PCTN	N	PCTN	N	PCTN
Org	Class								
Energy	Date								
	07APR	1	100.0	.	.	.	.	1	100.0
	03MAY	1	50.0	1	50.0	.	.	2	100.0
	22JUN	1	100.0	.	.	.	.	1	100.0
	12OCT	3	75.0	1	25.0	.	.	4	100.0
Mgt S	07APR	1	100.0	.	.	.	.	1	100.0
	03MAY	.	.	1	100.0	.	.	1	100.0
	22JUN	.	.	.	.	4	100.0	4	100.0
	12OCT	1	25.0	.	.	3	75.0	4	100.0
Power	07APR	1	33.3	2	66.7	.	.	3	100.0
	03MAY	1	100.0	.	.	.	.	1	100.0
	22JUN	1	25.0	.	.	3	75.0	4	100.0
	12OCT	1	100.0	.	.	.	.	1	100.0

```
TABLE  ORG*DATE ALL,
      (LOC ALL)*
           *(N*F=3.0 PCTN<ORG*DATE ALL>);
```

		Location							
		A		B		C		ALL	
		N	PCTN	N	PCTN	N	PCTN	N	PCTN
Org	Class								
Energy	Date								
	07APR	1	8.3	.	.	.	.	1	3.7
	03MAY	1	8.3	1	20.0	.	.	2	7.4
	22JUN	1	8.3	.	.	.	.	1	3.7
	12OCT	3	25.0	1	20.0	.	.	4	14.8
Mgt S	07APR	1	8.3	.	.	.	.	1	3.7
	03MAY	.	.	1	20.0	.	.	1	3.7
	22JUN	.	.	.	.	4	40.0	4	14.8
	12OCT	1	8.3	.	.	3	30.0	4	14.8
Power	07APR	1	8.3	2	40.0	.	.	3	11.1
	03MAY	1	8.3	.	.	.	.	1	3.7
	22JUN	1	8.3	.	.	3	30.0	4	14.8
	12OCT	1	8.3	.	.	.	.	1	3.7
ALL		12	100.0	5	100.0	10	100.0	27	100.0

The same rules apply for denominator specifications that are NOT the entire expression.

```
TABLE  ORG*DATE,
      (LOC ALL)*(N*F=3.0 PCTN<DATE>);
```

		Location							
		A		B		C		ALL	
		N	PCTN	N	PCTN	N	PCTN	N	PCTN
Org	Class								
Energy	Date								
	07APR	1	16.7	.	.	.	.	1	12.5
	03MAY	1	16.7	1	50.0	.	.	2	25.0
	22JUN	1	16.7	.	.	.	.	1	12.5
	12OCT	3	50.0	1	50.0	.	.	4	50.0
Mgt S	07APR	1	50.0	.	.	.	.	1	10.0
	03MAY	.	.	1	100.0	.	.	1	10.0
	22JUN	.	.	.	.	4	57.1	4	40.0
	12OCT	1	50.0	.	.	3	42.9	4	40.0
Power	07APR	1	25.0	2	100.0	.	.	3	33.3
	03MAY	1	25.0	.	.	.	.	1	11.1
	22JUN	1	25.0	.	.	3	100.0	4	44.4
	12OCT	1	25.0	.	.	.	.	1	11.1

With the denominator specification of DATE, TABULATE will use the total number of observations for all dates in that column as the denominator. But since DATE is nested within ORG, it will only use those observations that belong to that ORG. So, in the above example the total number of observations for location A in ORG Energy is 6, which becomes the denominator for computing PCTN for all those dates. You can also see the total number of observations for location B in ORG 'Mgt S' is 1, which becomes the denominator for computing PCTN for all those dates, thus the 100 percent on 03MAY. The total number of observations for the ALL column in ORG 'Power' is 9, which becomes the denominator for computing PCTN for all those dates.

```
TABLE  ORG*DATE,
      (LOC ALL)*(N*F=3.0 PCTN<ORG>);
```

		Location							
		A		B		C		ALL	
		N	PCTN	N	PCTN	N	PCTN	N	PCTN
Org	Class								
Energy	Date								
	07APR	1	33.3	.	.	.	.	1	20.0
	03MAY	1	50.0	1	50.0	.	.	2	50.0
	22JUN	1	50.0	.	.	.	.	1	11.1
	12OCT	3	60.0	1	100.0	.	.	4	44.4
Mgt S	07APR	1	33.3	.	.	.	.	1	20.0
	03MAY	.	.	1	50.0	.	.	1	25.0
	22JUN	.	.	.	.	4	57.1	4	44.4
	12OCT	1	20.0	.	.	3	100.0	4	44.4
Power	07APR	1	33.3	2	100.0	.	.	3	60.0
	03MAY	1	50.0	.	.	.	.	1	25.0
	22JUN	1	50.0	.	.	3	42.9	4	44.4
	12OCT	1	20.0	.	.	.	.	1	11.1

With the denominator specification of ORG, TABULATE will use

the total number of observations for all organizations in that column as the denominator. But since ORG is nested with DATE, it will only use those observations that belong to that DATE. So, in the above example the total number of observations for location A with a date of 07APR is 3, which becomes the denominator for computing PCTN for that date in every ORG in that location, thus the 33.3 percent for each one with a count of 1. The total number of observations for location A with a date of 12OCT is 5, which becomes the denominator for computing PCTN for that date in every ORG in that location, thus the 20 percent for each one with a count of 1. The total number of observations for location B with a date of 03MAY is 2, which becomes the denominator for computing PCTN for that date in every ORG in that location, thus the 50 percent for each one with a count of 1. The total number of observations for the ALL column with a date of 03MAY is 4, which becomes the denominator for computing PCTN for that date in every ORG, thus the 25 percent for each one with a count of 1.

The ALL in the denominator specification gave me a real hard time at first until I discovered it is really only needed to satisfy the table expression expansion. Typically ALL is used to do some sort of totaling and is thus concatenated not nested. So, all (ha! ha!) you have to do is include it in your denominator as shown below.

```
TABLE  ORG*DATE ALL,
       (LOC ALL)*(N*F=3.0 PCTN<ORG ALL>);
```

	Location							
	A		B		C		ALL	
	N	PCTN	N	PCTN	N	PCTN	N	PCTN
Org Energy / Class Date								
07APR	1	33.3	.	.	.	.	1	20.0
03MAY	1	50.0	1	50.0	.	.	2	50.0
22JUN	1	50.0	.	.	.	.	1	11.1
12OCT	3	60.0	1	100.0	.	.	4	44.4
Mgt S 07APR	1	33.3	.	.	.	.	1	20.0
03MAY	.	.	1	50.0	.	.	1	25.0
22JUN	.	.	.	.	4	57.1	4	44.4
12OCT	1	20.0	.	.	3	100.0	4	44.4
Power 07APR	1	33.3	2	100.0	.	.	3	60.0
03MAY	1	50.0	.	.	.	.	1	25.0
22JUN	1	50.0	.	.	3	42.9	4	44.4
12OCT	1	20.0	.	.	.	.	1	11.1
ALL	12	100.0	5	100.0	10	100.0	27	100.0

If you leave it out of the denominator specification, you will get the messages:

ERROR: PCTN base is not in table.
ERROR: A PCTN crossing has no denominator.

Where the ALL gets real complicated is when you nest the ALLs in groupings, then you will need to expand the "crossings" as the SAS manuals indicate to be sure you get the proper denominator.

To get a better feel for the use of percentages, let's use the subtotaling example from earlier and add a subtotal percentage.

```
TABLE
   ORG*(LOC ALL='Loc Subtotal') ALL='Org Total',
   SCORE*(SUM*F=6.1 PCTSUM<LOC ALL>)
   / RTS=25 BOX=SCORE
     ROW=FLOAT MISSTEXT='None';
```

Final Exam Score		Final Exam Score	
		SUM	PCTSUM
Org	Location		
Energy	A	337.6	63.7
	B	192.8	36.3
	Loc		
	Subtotal	530.4	100.0
Mgt S	Location		
	A	146.7	17.8
	B	85.4	10.3
	C	593.5	71.9
	Loc		
	Subtotal	825.6	100.0
Power	Location		
	A	356.2	48.3
	B	141.3	19.2
	C	239.4	32.5
	Loc		
	Subtotal	736.9	100.0
Org Total		2092.9	100.0

VERSION 8 FEATURES

Until I can fully understand (and try!) all the new features of TABULATE in version 8, I will only discuss some of the more interesting ones here. Following are most of the new features. Many of them are in response to the SASware Ballot.

The PROC TABULATE statement supports these new options:

- CLASSDATA= - specifies a data set that contains the combinations of class variable values to include in analysis.
- CONTENTS= - allows you to name the link in the HTML table of contents that points to the ODS output of the first table produced.
- EXCLNPWGT - excludes observations with nonpositive weights from the analysis.
- EXCLUSIVE - excludes from the analysis all class variable combinations that are not in the CLASSDATA= data set.
- NOTRAP - disables trapping mathematical errors due to overflow.
- OUT= - names the output data set.
- QMARKERS= - specifies the default number of markers to use for the P2 (fixed space) quantile estimation method.
- QMETHOD - specifies the method to process the input data to compute quantiles.
- QNTLDEF= - specifies the mathematical definition used to compute quantiles.
- TRAP - enables trapping mathematical errors due to overflow.

TABULATE now supports multiple CLASS statements and the following new options:

- ASCENDING - specifies to sort the class variable levels in ascending order.
- DESCENDING - specifies to sort the class variable levels in descending order.
- EXCLUSIVE - excludes from the analysis all class variable values that are not found in the preloaded range of user-defined formats.
- GROUPINTERNAL - specifies not to apply formats to the class variables when TABULATE sorts the values to create combinations of class variables.
- MISSING - considers missing values as valid class variable levels.
- MLF - enables TABULATE to use the primary and secondary format labels for a given range or overlapping ranges to create the subgroup combinations when a multilabel format is assigned to a class variable.
- ORDER= - specifies the sort order for the levels of the class variables in the output.
- PRELOADFMT - specifies to preload all the formats for the

class variables.

- MLF - allows you to make use of multiple labels when a multilabel format is assigned to a class variable in PROC FORMAT.

TABULATE also supports multiple VAR statements.

In the TABLE statement, the following options have been enhanced:

- CONDENSE - prints multiple logical pages on a physical page.
- CONTENTS= - allows you to name the link in the HTML table of contents that points to the ODS output of the table produced using the TABLE statement.
- NOCONTINUED - suppresses the printing of the "(Continued)" continuation message for tables that span physical pages.

PROC TABULATE supports these new statistics that PROC MEANS and SUMMARY also now supports:

- COLPCNT
- COLPCTSUM
- MEDIAN
- P1
- P5
- P10
- P90
- P95
- P99
- PAGEPCTN
- PAGEPCTSUM
- Q1
- Q3
- QRANGE
- REPPCTN
- REPPCTSUM
- ROWPCTN
- ROWPCTSUM

The first thing I had to try was some of the new statistics.

```
PROC TABULATE  DATA=CLASS   OUT=CLASSOUT
          FORMAT=5.1 NOSEPS;
    CLASS ORG LOC DATE;
    VAR   SCORE;
    TABLE  ORG*DATE all,
    SCORE*(LOC ALL)*(N*F=2. MEDIAN*F=5.1
         COLPCTN*F=3. REPPCTN*F=3. )
    / RTS=15 CONDENSE NOCONTINUED ;
    RUN;
```

	Final Exam Score													
	Location													
	A				B				C				All	
		Medi-	Co-lP-	Re-pP-		Medi-	Co-lP-	Re-pP-		Medi-	Co-lP-	Re-pP-		Medi-
N	an	ctN	ctN	N	an	ctN	ctN	N	an	ctN	ctN	N	an	
Org Energy	Class Date													
	03MAY	1	10	4	1	92.8	20	4	.	.	.	.	2	90.9
	22JUN	1	10	4	.	.	.	.	.	.	.	.	1	69.9
	12OCT	2	20	8	1	100.0	20	4	.	.	.	.	3	93.0
Mgt S	07APR	1	10	4	.	.	.	.	.	.	.	.	1	99.4
	03MAY	.	.	.	1	85.4	20	4	.	.	.	.	1	85.4
	22JUN	.	.	.	.	.	.	.	4	87.7	40	16	4	87.7
	12OCT	1	47.3	10	4	.	.	.	3	86.8	30	12	4	76.7
Power	07APR	1	99.1	10	4	70.7	40	8	.	.	.	.	3	90.0
	03MAY	1	93.5	10	4	.	.	.	.	.	.	.	1	93.5
	22JUN	1	70.1	10	4	.	.	.	3	81.2	30	12	4	75.7
	12OCT	1	93.5	10	4	.	.	.	.	.	.	.	1	93.5
All		10	91.0	100	40	90.0	100	20	10	84.0	100	40	25	88.9

Org Energy	Class Date	Final Exam Score All
	03MAY	8 8
	22JUN	4 4
	12OCT	12 12
Mgt S	07APR	4 4
	03MAY	4 4
	22JUN	16 16
	12OCT	16 16
Power	07APR	12 12
	03MAY	4 4
	22JUN	16 16
	12OCT	4 4
All		100 100

It is amazing that after all these years of SAS programmers trying….**and trying….*and trying*** to figure out how to make the @!#$%^**& TABULATE denominator specification work right, ***they make a statistic to do it for you!***

Let's see what the output dataset looks like.

```
PROC CONTENTS  DATA=CLASSOUT;
        RUN;
```

The CONTENTS Procedure

Data Set Name:	SASUSER.CLASSOUT	Observations:	31
Member Type:	DATA	Variables:	10
Engine:	V8	Indexes:	0
Created:	8:53 Friday, October 13, 2000	Observation Length:	72
Last Modified:	8:53 Friday, October 13, 2000	Deleted Observations:	0
Protection:		Compressed:	NO
Data Set Type:		Sorted:	NO
Label:			

-----Alphabetic List of Variables and Attributes-----

#	Variable	Type	Len	Pos	Format	Label
5	_PAGE_	Num	8	8		Page for Observation
6	_TABLE_	Num	8	16		Table for Observation
4	_TYPE_	Char	3	63		Type of Observation
3	DATE	Num	8	0	DATE.	Class Date
2	LOC	Char	1	62		Location
1	ORG	Char	6	56		Org
8	SCORE_Median	Num	8	32		
7	SCORE_N	Num	8	24		
10	SCORE_PctN_000	Num	8	48		
9	SCORE_PctN_010	Num	8	40		

```
PROC PRINT  DATA=CLASSOUT;
        RUN;
```

Obs	ORG	LOC	DATE	_TYPE_	_PAGE_	_TABLE_	SCORE _N	SCORE_Median	SCORE_P10	SCORE_P90
1	Energy	A	03MAY80	111	1	1	1	88.90	10	4
2	Energy	B	03MAY80	111	1	1	1	92.80	20	4
3	Energy	A	22JUN80	111	1	1	1	69.90	10	4
4	Energy	A	12OCT80	111	1	1	2	89.40	20	8
5	Energy	B	12OCT80	111	1	1	1	100.00	20	4
6	Mgt S	A	07APR80	111	1	1	1	99.40	10	4
7	Mgt S	B	03MAY80	111	1	1	1	85.40	20	4
8	Mgt S	C	22JUN80	111	1	1	4	87.65	40	16
9	Mgt S	A	12OCT80	111	1	1	1	47.30	10	4
10	Mgt S	C	12OCT80	111	1	1	3	86.80	30	12
11	Power	A	07APR80	111	1	1	1	99.10	10	4
12	Power	B	07APR80	111	1	1	2	70.65	40	8
13	Power	A	03MAY80	111	1	1	1	93.50	10	4
14	Power	A	22JUN80	111	1	1	1	70.10	10	4
15	Power	C	22JUN80	111	1	1	3	81.20	30	12
16	Power	A	12OCT80	111	1	1	1	93.50	10	4
17	Energy		03MAY80	101	1	1	2	90.85	.	8
18	Energy		22JUN80	101	1	1	1	69.90	.	4
19	Energy		12OCT80	101	1	1	3	93.00	.	12
20	Mgt S		07APR80	101	1	1	1	99.40	.	4
21	Mgt S		03MAY80	101	1	1	1	85.40	.	4
22	Mgt S		22JUN80	101	1	1	4	87.65	.	16
23	Mgt S		12OCT80	101	1	1	4	76.65	.	16
24	Power		07APR80	101	1	1	3	90.00	.	12
25	Power		03MAY80	101	1	1	1	93.50	.	4
26	Power		22JUN80	101	1	1	4	75.65	.	16
27	Power		12OCT80	101	1	1	1	93.50	.	4
28		A		010	1	1	10	90.95	100	40
29		B		010	1	1	5	90.00	100	20
30		C		010	1	1	10	84.00	100	40
31				000	1	1	25	88.90	.	100

Multiple CLASS statements give you the ability to specify different options to different variables.

```
PROC TABULATE  DATA=CLASS  FORMAT=7.1 NOSEPS;
    CLASS  LOC  / DESCENDING ;
    CLASS  DATE / MISSING ;
    CLASS  ORG  / MISSING;
    VAR   SCORE;
    TABLE  ORG*DATE ALL,
         (LOC ALL)*(N*F=3.0 COLPCTN);
    RUN;
```

	Location							
	C		B		A		All	
	N	ColPctN	N	ColPctN	N	ColPctN	N	ColPctN
Org Class Date								
.	.	.	.	.	1	9.1	1	3.8
Energy 03MAY	.	.	1	20.0	1	9.1	2	7.7
22JUN	.	.	.	.	1	9.1	1	3.8
12OCT	.	.	1	20.0	2	18.2	3	11.5
Mgt S 07APR	.	.	.	.	1	9.1	1	3.8
03MAY	.	.	1	20.0	.	.	1	3.8
22JUN	4	40.0	.	.	.	.	4	15.4
12OCT	3	30.0	.	.	1	9.1	4	15.4
Power 07APR	.	.	2	40.0	1	9.1	3	11.5
03MAY	.	.	.	.	1	9.1	1	3.8
22JUN	3	30.0	.	.	1	9.1	4	15.4
12OCT	.	.	.	.	1	9.1	1	3.8
All	10	100.0	5	100.0	11	100.0	26	100.0

```
PROC FORMAT ;
    VALUE $LOCFMT
        'A'='Knoxville'
        'B'='Chattanooga'
        'C'='Nashville'
        'D'='Memphis';
        RUN;
```

```
PROC TABULATE  DATA=CLASS  FORMAT=7.1 NOSEPS;
    CLASS  LOC  / DESCENDING PRELOADFMT;
    CLASS  DATE / MISSING ;
    CLASS  ORG  / MISSING;
    Format loc  $locfmt. ;
    VAR   SCORE;
    TABLE  ORG*DATE ALL,
         (LOC)*(N*F=3.0 COLPCTN)
    / PRINTMISS ;
    RUN;
```

		Location							
		Memphis		Nashville		Chattanooga		Knoxville	
		N	ColPctN	N	ColPctN	N	ColPctN	N	ColPctN
Org	Class Date								
	.	.	.	.	0.0	.	0.0	1	9.1
	07APR	.	.	.	0.0	.	0.0	.	0.0
	03MAY	.	.	.	0.0	.	0.0	.	0.0
	22JUN	.	.	.	0.0	.	0.0	.	0.0
	12OCT	.	.	.	0.0	.	0.0	.	0.0
Energy	.	.	.	.	0.0	.	0.0	.	0.0
	07APR	.	.	.	0.0	.	0.0	.	0.0
	03MAY	.	.	.	0.0	1	20.0	1	9.1
	22JUN	.	.	.	0.0	.	0.0	1	9.1
	12OCT	.	.	.	0.0	1	20.0	2	18.2
Mgt S	.	.	.	.	0.0	.	0.0	.	0.0
	07APR	.	.	.	0.0	.	0.0	1	9.1
	03MAY	.	.	.	0.0	1	20.0	.	0.0
	22JUN	.	.	4	40.0	.	0.0	.	0.0
	12OCT	.	.	3	30.0	.	0.0	1	9.1
Power	.	.	.	.	0.0	.	0.0	.	0.0
	07APR	.	.	.	0.0	2	40.0	1	9.1
	03MAY	.	.	.	0.0	.	0.0	1	9.1
	22JUN	.	.	3	30.0	.	0.0	1	9.1
	12OCT	.	.	.	0.0	.	0.0	1	9.1
All		.	.	10	100.0	5	100.0	11	100.0

But what I think is one of the most useful new features of version 8 is ODS. If you get a chance, attend one of Ray Pass's workshops or papers on ODS; they are very well done and very informative. Actually any class or paper on ODS would be useful. Virtually any output SAS can generate can use ODS.

```
ODS  html  body="e:\sugi\advtab1.html";
PROC TABULATE  DATA=CLASS  FORMAT=7.1 NOSEPS;
    CLASS  LOC  / DESCENDING ;
    CLASS  DATE / MISSING ;
    CLASS  ORG  / MISSING;
    VAR   SCORE;
    TABLE  ORG*DATE ALL,
         (LOC ALL)*(N*F=3.0 COLPCTN);
RUN;
ODS  html close;
```

		Location						All	
		C		B		A			
		N	ColPct N	N	ColPct N	N	ColPct N	N	ColPct N
Org	Class Date								
	.	.	.	.	.	1	9.1	1	3.8
Energy	03MAY	.	.	1	20.0	1	9.1	2	7.7
	22JUN	.	.	.	.	1	9.1	1	3.8
	12OCT	.	.	1	20.0	2	18.2	3	11.5
Mgt S	07APR	.	.	.	.	1	9.1	1	3.8
	03MAY	.	.	1	20.0	.	.	1	3.8
	22JUN	4	40.0	.	.	.	.	4	15.4
	12OCT	3	30.0	.	.	1	9.1	4	15.4
Power	07APR	.	.	2	40.0	1	9.1	3	11.5
	03MAY	.	.	.	.	1	9.1	1	3.8
	22JUN	3	30.0	.	.	1	9.1	4	15.4
	12OCT	.	.	.	.	1	9.1	1	3.8
All		10	100.0	5	100.0	11	100.0	26	100.0

IN SUMMARY

I never thought when I first wrote the first version of this in 1992, that it would survive until the next version. Well, it has and versions 7 and 8 have introduced some new features…***So is this really the FINAL CHAPTER?!?!*** Only time will tell!

The one thing SAS has done over the years is virtually guarantee code written in prior versions will continue to work in later ones … upward compatibility. TABULATE is no different; even though some of the new features make old techniques obsolete; it is still good to know them.

This paper is not intended to be a cure for all your TABULATE problems. Every use of TABULATE is unique in some ways. All I have attempted to do is give you a good starting point or foundation to better understand how to get TABULATE to give you what you want. The more complicated your "crossings", as the SAS manuals refer to them, the tougher it is going to be to determine the denominator specification. Most everything else about TABULATE is very straight forward.

So good luck and happy tabulating!!!

ACKNOWLEDGEMENTS

For a complete discussion of TABULATE and its uses see the "SAS Guide to TABULATE Processing, Second Edition" and the "SAS Language and Procedures, Usage, Version 6, First Edition", chapter 25, "Creating Summary Tables". There is a very good article in the first issue of "Observations, The Technical Journal for SAS Software Users", vol. 1, no. 1, entitled "Computing Percentages with PROC TABULATE" by Tina Keene.

Lauren Haworth has written a book entitled "PROC TABULATE By Example" that is full of all kinds of examples for all kinds of applications and well worth the money.

AUTHOR

If you have any questions or comments, please write or call:

Dan Bruns
Tennessee Valley Authority
1101 Market Street (MP 2B)
Chattanooga, TN 37402
423/751-6430 Fax 423/751-3163
Email: debruns@tva.gov

ODS for PRINT, REPORT and TABULATE

Lauren Haworth, Genentech, Inc., San Francisco

➢ ABSTRACT

For most procedures in the SAS system, the only way to change the appearance of the output is to change or modify the ODS style definition. There are three exceptions: REPORT, TABULATE, and PRINT. These procedures allow you to change the output style attributes on the fly when the output is generated.

With these three procedures, you can create almost any type of tabular report. Add in the extra control over style attributes, and you have a reporting powerhouse.

This paper will show how to change the fonts, colors, and alignment of your output. You will also learn how to use formats to highlight key results in special colors and use images in table headings.

For convenience, all of the examples are shown as HTML output. Except where noted, the examples all work for RTF or printer output as well.

The examples in this paper are based on SAS version 8.2. The PRINT examples will only work with version 8.2 or later. Most of the REPORT and TABULATE examples will also work with versions 8.0 and 8.1.

➢ INTRODUCTION

This first series of examples will show how you can use style attributes to modify the appearance of the result table headings. Later examples will show techniques for rows and table values.

The STYLE= option is a new option available for just the PRINT, REPORT, and TABULATE procedures. What it allows you to do is to specify a number of different style attributes for specific parts of your output. These style attributes control things like typefaces, foreground and background colors, text alignment, and table borders.

➢ EXAMPLE #1: CHANGING TABLE HEADING STYLES FOR PROC PRINT

Applying style attributes to PROC PRINT output is very simple. All you have to do is add STYLE= options to whichever part of the output you wish to change.

There are a number of ways you can use STYLE= in the PRINT procedure. To keep things simple, we'll just look at one technique: adding a STYLE= option to the PROC PRINT statement. We'll start with the following code. It produces a listing of three variables.

```
ODS HTML FILE='tables.htm';
proc print data=tables noobs label;
   var Type Material Price;
run;
ODS HTML CLOSE;
```

This code produces the output shown below. It's a basic table in the Default style definition (only the first few rows of the table are shown).

Table Type	Construction	Average Price
Coffee	Wood	$62
Coffee	Glass/Metal	$112
Dining	Glass/Metal	$559
Dining	Wood	$274
Dining	Wood	$220

The typeface used in the headings and table body is Arial. As an example, we will change this typeface to Arial Narrow, which will make the headings narrower. This is useful if you have a table that is too wide.

We do this by adding a STYLE= option to the PROC PRINT statement. The STYLE keyword is followed by the name of the style element we wish to modify. In this case, the element is called Header. A list of the elements you can modify is in the Online Documentation for the PRINT procedure.

The style element is listed in parentheses. Following the name of the style element, we use an equal sign and then list the attribute we wish to change. Detailed documentation on these attributes and their settings can be found in the "Guide to the Output Delivery System" in the Online Documentation.

The attribute must be contained between square brackets "[ ]" or curly brackets "{ }". The code below changes the font typeface to Arial Narrow. No-

tice the use of quotes around the typeface name. Because 'Arial Narrow' contains spaces, these quote marks are necessary.

```
ODS HTML BODY='tables.htm';
proc print data=tables noobs label
   STYLE(Header)=[FONT_FACE='Arial Narrow'];
   var Type Material Price;
run;
ODS HTML CLOSE;
```

The results are shown below. Now each of the table headings takes up less width. This makes the overall table narrower, allowing you to fit more information on the page (or screen). This technique can be used to change any of the column heading attributes.

Table Type	Construction	Average Price
Coffee	Wood	$62
Coffee	Glass/Metal	$112
Dining	Glass/Metal	$559
Dining	Wood	$274
Dining	Wood	$220

➢ Example #2: Changing Table Heading Styles for PROC REPORT

Next, we'll run through the same column heading modification for PROC REPORT output. We'll start with a table similar to the previous example.

```
ODS HTML BODY='tables.htm';
proc report data=tables nowd;
   column Type Material Price;
   define Type / group;
   define Material / group;
   define Price / analysis mean;
run;
ODS HTML CLOSE;
```

This code produces the output shown in below. It's a basic table using the Default style definition, and is very similar to the PROC PRINT table we modified earlier.

Table Type	Construction	Average Price
Coffee	Glass/Metal	$103
	Wood	$59
Dining	Glass/Metal	$404
	Wood	$196
End	Glass/Metal	$107
	Wood	$61

Once again, we're going to change the heading font. We do this with the exact same technique as we used for PROC PRINT. We add a STYLE(Header)= option to the PROC REPORT statement.

```
ODS HTML BODY='tables.htm';
proc report data=tables nowd
   STYLE(Header)=[FONT_FACE='Arial Narrow'];
   column Type Material Price;
   define Type / group;
   define Material / group;
   define Price / analysis mean;
run;
ODS HTML CLOSE;
```

The new results are shown below. Again, we get a table with narrower headings. This technique can be used to change any of the column heading attributes.

Table Type	Construction	Average Price
Coffee	Glass/Metal	$103
	Wood	$59
Dining	Glass/Metal	$404
	Wood	$196
End	Glass/Metal	$107
	Wood	$61

➢ Example #3: Changing Table Heading Styles for PROC TABULATE

To round out these examples, we'll run through making the same change for a table produced by PROC TABULATE.

Once again, we will add a STYLE= option to the part of the output we wish to change. We'll start with a table similar to the previous examples.

```
ODS HTML BODY='tables.htm';
proc tabulate data=tables f=dollar8.;
   class Type Material;
   var Price;
   table Type*Material,
         Price*Mean=" ";
run;
ODS HTML CLOSE;
```

This code produces the output shown below. It's a basic table using the Default style definition, and is very similar to the previous tables created with PRINT and REPORT.

		Average Price
Table Type	Construction	
Dining	Wood	$196
	Glass/Metal	$404
Coffee	Wood	$59
	Glass/Metal	$103
End	Wood	$61
	Glass/Metal	$107

This time, instead of adding a STYLE(Header)= option to the main procedure call, we need to do things variable by variable. With TABULATE, you specify styles for the row and column headings in the VAR and CLASS statements that identify the variables.

```
ODS HTML BODY='tables.htm';
proc tabulate data=tables f=dollar8.;
   class Type Material /
      STYLE=[FONT_FACE="Arial Narrow"];
   var Price /
      STYLE=[FONT_FACE="Arial Narrow"];
   table Type*Material,
         Price*Mean=" ";
run;
ODS HTML CLOSE;
```

The results are shown below. Notice how the three top headings now show the new narrower font. However, the row headings do not show the same change. This is because these are the class level *value* headings, not the class *variable* headings.

		Average Price
Table Type	Construction	
Dining	Wood	$196
	Glass/Metal	$404
Coffee	Wood	$59
	Glass/Metal	$103
End	Wood	$61
	Glass/Metal	$107

To make all of the headings match, we need to add one more statement to our code. The CLASSLEV statement is used to apply options to class level values. You can use the STYLE= option on the CLASSLEV statement to change the fonts to match the top headings.

```
ODS HTML BODY='tables.htm';
proc tabulate data=tables f=dollar8.;
   class Type Material /
      STYLE=[FONT_FACE="Arial Narrow"];
   classlev Type Material /
      STYLE=[FONT_FACE="Arial Narrow"];
   var Price /
      STYLE=[FONT_FACE="Arial Narrow"];
   table Type*Material,
         Price*Mean=" ";
run;
ODS HTML CLOSE;
```

The new results are shown below.

		Average Price
Table Type	Construction	
Dining	Wood	$196
	Glass/Metal	$404
Coffee	Wood	$59
	Glass/Metal	$103
End	Wood	$61
	Glass/Metal	$107

This technique can be used to change any of the row or column headings. You can also use different style attributes for each classification variable, by creating multiple CLASS and CLASSLEV statements, one set for "Type" and one set for "Material". Then you could use a different STYLE= option for each variable. The same technique also works when you have two analysis variables, you can use two VAR statements with two different STYLE= options.

➢ Example #4: Applying Traffic Lighting to PROC PRINT Results

What do traffic lights have to do with SAS Output? The answer is that the familiar red, yellow, and green lights can be used to highlight results in your output. You use red for bad results, yellow for neutral results, and green for good results. If you're creating a large table, this technique is great for focusing the reader's attention on the key results.

In this example, we will take our familiar table and use traffic lighting to mark low prices in green, high prices in red, and prices in between in yellow.

The first step is to set up a format with the colors we'd like to use. The three color settings are RGB values for red, yellow, and green.

```
proc format;
   value traffic low-100='cx006600'
                 100<-300='cxFF9900'
                 300<-high='cxCC0000';
run;
```

The next step is to set up the code for the table, and then apply the format. The code is shown below.

```
ODS HTML FILE='tables.htm';
proc print data=tables noobs label;
   var Type Material;
   var Price /
      STYLE=[BACKGROUND=traffic.];
run;
ODS HTML CLOSE;
```

It's the same code from our previous example. The only difference is that the VAR statement has been split into two statements. That's so the STYLE= option can be applied to just the variable Price. The style attribute we are modifying is BACKGROUND, which controls the background color.

You could just specify a single background color here, such as "Red" or "cxCC0000", which would make the entire column of results red. Instead, we will use our format. This allows us to have a conditional background color. Depending on the value of each table cell, the background will be set as green, yellow or red. The results are shown below.

Table Type	Construction	Average Price
Coffee	Wood	62.361
Coffee	Glass/Metal	112.461
Dining	Glass/Metal	559.462
Dining	Wood	273.571
Dining	Wood	220.275

Now the low prices show up in green, the high prices in red, and the ones in between are in yellow. However, with these dark backgrounds, the price values get a bit hard to read. The default font weight is a bit light for these intense backgrounds. To fix that, we'll make the fonts bold, as shown in the code below.

```
ODS HTML FILE='tables.htm';
proc print data=tables noobs label;
   var Type Material /
      STYLE=[FONT_WEIGHT=BOLD];
   var Price /
      STYLE=[BACKGROUND=traffic.
             FONT_WEIGHT=BOLD];
run;
ODS HTML CLOSE;
```

The font is changed to bold in both VAR statements so that the table will look consistent. The new output is shown below.

Table Type	Construction	Average Price
Coffee	Wood	62.361
Coffee	Glass/Metal	112.461
Dining	Glass/Metal	559.462
Dining	Wood	273.571
Dining	Wood	220.275

By the way, you can also do traffic lighting by changing the foreground values. Instead of making the cell background red, yellow, or green, you can make the cell text red, yellow, or green. The code is the same, except instead of using BACKGROUND= in the STYLE= option, you use FOREGROUND=.

One additional note regarding this example: you may have noticed that the price amounts were not formatted as dollar amounts. That is because there is a bug in version 8.2 that prevents the use of a format in this way on a variable that is already formatted. The problem affects only the PRINT procedure, and not the REPORT or TABULATE procedures.

➢ Example #5: Applying Traffic Lighting to PROC REPORT Results

For PROC REPORT, the technique for doing traffic lighting is almost the same.

You add the STYLE= options to the DEFINE statements that set up each table column. For the Price column, the background color is set to the format we defined previously. For all of the columns, the font weights are set to bold to increase readability. The code is shown below.

```
ODS HTML BODY='tables.htm';
proc report data=tables nowd;
   column Type Material Price;
   define Type / group
      STYLE=[FONT_WEIGHT=BOLD];
   define Material / group
      STYLE=[FONT_WEIGHT=BOLD];
   define Price / analysis mean
      STYLE=[BACKGROUND=traffic.
             FONT_WEIGHT=BOLD];
run;
ODS HTML CLOSE;
```

The resulting table is shown below. It looks much like the PROC PRINT results.

Table Type	Construction	Average Price
Coffee	Glass/Metal	$103
	Wood	$59
Dining	Glass/Metal	$404
	Wood	$196
End	Glass/Metal	$107
	Wood	$61

➢ Example #6: Applying Traffic Lighting to PROC TABULATE Results

For PROC TABULATE, the technique for doing traffic lighting is a bit different. The STYLE= option settings are the same as the previous two examples, but the way to apply the STYLE= option is different.

You might thing that to apply a style to the values of Price, you would add the STYLE= option to the VAR statement for Price. However, that would only affect the heading for Price.

To change the appearance of values within a table, you need to apply the STYLE= option in the TABLE statement. The following code shows how this is done.

```
ODS HTML BODY='tables.htm';
proc tabulate data=tables f=dollar8.;
   class Type Material;
   var Price;
   table Type*Material,
         Price*Mean=" "*
         [STYLE=[BACKGROUND=traffic.
                 FONT_WEIGHT=BOLD]];
run;
ODS HTML CLOSE;
```

The STYLE= option is added to the TABLE statement by using an asterisk operator. Notice that it is added to specification for the variable Price in the column dimension. This can be a little confusing to see, because the statistic MEAN is also applied to the variable Price. Another thing to notice is that this time the STYLE= option uses two sets of brackets. The additional brackets surround the entire STYLE= option. These are required when adding STYLE= options within a TABLE statement.

The resulting table is shown below.

		Average Price
Table Type	Construction	
Dining	Wood	$196
	Glass/Metal	$404
Coffee	Wood	$59
	Glass/Metal	$103
End	Wood	$61
	Glass/Metal	$107

➢ Example #7: Creating Alternate Row Shading for PROC REPORT Results

While most everything you can do in PRINT and TABULATE can be done in REPORT, a few tricks work best in PROC REPORT. This is because you can use a COMPUTE block to calculate the output formatting.

A good example of this is creating a table with alternate row shading. This means that the rows of data alternate light and dark colors, making them easier to read. The code for doing this is shown below.

```
ODS HTML BODY='tables.htm';
proc report data=tables nowd;
   column Type Material Price;
   define Type / group;
   define Material / group;
   define Price / analysis mean;
   compute Material;
      count+1;
      if (mod(count,2)) then do;
         CALL DEFINE(_ROW_, "STYLE",
         "STYLE=[BACKGROUND=cxFFFFFF]");
      end;
   endcomp;
run;
ODS HTML CLOSE;
```

What this code does is figure out whether each row is an even or odd number. For odd numbered rows, the background color is assigned to white (the default color is gray).

Instead of using STYLE=, we will use a CALL DEFINE statement. This allows us to call up the row style element, and modify its attributes. The CALL DEFINE has three parameters: the name of the element being modified (_ROW_), the description of what is being modified (STYLE), and the STYLE= code to make the change.

The resulting table is shown below.

Table Type	Construction	Average Price
Coffee	Glass/Metal	$103
	Wood	$59
Dining	Glass/Metal	$404
	Wood	$196
End	Glass/Metal	$107
	Wood	$61

➢ Example #8: Adding a Logo to a PROC REPORT Table

Changing fonts and colors goes a long way to livening up your output, but adding graphics takes your results to a new level. This example will show how to add a heading with a logo to your PROC REPORT results.

The first thing you need to do is set up the heading. This is done with a COMPUTE BEFORE _PAGE_ statement and a LINE statement.

```
ODS HTML BODY='tables.htm';
proc report data=tables nowd;
   column Type Material Price;
   define Type / group;
   define Material / group;
   define Price / analysis mean;
   compute before _page_ / LEFT;
      LINE "Totally Tables, Inc.";
   endcomp;
run;
ODS HTML CLOSE;
```

The result is that a new row is added to the top of the table, and it contains the text given in the LINE statement, as shown in the output below. Unfortunately, the row is created using the default table cell style attributes, which makes it rather faint in appearance.

Totally Tables, Inc.		
Table Type	**Construction**	**Average Price**
Coffee	Glass/Metal	$103
	Wood	$59
Dining	Glass/Metal	$404
	Wood	$196
End	Glass/Metal	$107
	Wood	$61

To fix the appearance of the title and add a graphic, we can use the STYLE= option on the COMPUTE statement. First, we'll add the graphic using the PREIMAGE attribute. This example assumes that the graphic is stored in the same file location as the HTML page we are creating. The second fix we need to make is to make the font bolder, bigger, and in a brown color that coordinates with the graphic.

```
compute before _page_ / LEFT
   STYLE=[PREIMAGE='table.gif'
          FONT_WEIGHT=BOLD
          FONT_SIZE=5
          FOREGROUND=cx993300];
   LINE "Totally Tables, Inc.";
endcomp;
```

The new results are shown below.

Totally Tables, Inc.		
Table Type	**Construction**	**Average Price**
Coffee	Glass/Metal	$103
	Wood	$59
Dining	Glass/Metal	$404
	Wood	$196
End	Glass/Metal	$107
	Wood	$61

Note: This example is set up for HTML output, and gives the font size using a number which represents an HTML font size. If you are creating RTF or printer output, list the font size in points (for example, "14pt").

➢ Example #8: Adding a Logo to a PROC TABULATE Table

Adding a logo is even easier using PROC TABULATE. You can easily place a graphic image in the top left corner of the table using the BOX= option.

The style attribute information is added with a STYLE= suboption on the BOX= option. The PREIMAGE style attribute allows you to name an image that precedes the text in the cell.

In this example, the title is added using a LABEL= option, and then is followed by the STYLE= option. Note the unusual syntax. In order to have both a LABEL= and a STYLE= suboption, both need to be enclosed in brackets following the BOX= option.

```
ODS HTML BODY='tables.htm';
proc tabulate data=tables f=dollar8.;
   class Type Material;
   var Price;
   table Type*Material,
      Price*Mean=" "
      / BOX=[LABEL='Totally Tables, Inc.'
        STYLE=[PREIMAGE='dining.gif']];
run;
ODS HTML CLOSE;
```

This code produces the following table.

Totally Tables, Inc.		Average Price
Table Type	**Construction**	
Dining	**Wood**	$196
	Glass/Metal	$404
Coffee	**Wood**	$59
	Glass/Metal	$103
End	**Wood**	$61
	Glass/Metal	$107

Now our table contains the logo and title in the top left corner. However, the headings look funny because the title is sitting at the bottom of the table cell, but the heading "Average Price" is in the middle of the cell. We can use another STYLE= option to fix this.

```
ODS HTML BODY='tables.htm';
proc tabulate data=tables f=dollar8.;
   class Type Material;
   var Price / STYLE=[VJUST=B];
   table Type*Material,
      Price*Mean=" "
      / BOX=[LABEL='Totally Tables, Inc.'
        STYLE=[PREIMAGE='dining.gif']];
run;
ODS HTML CLOSE;
```

The VJUST=B style attribute added to the VAR statement for Price causes the heading for this variable to be vertically justified to the bottom of the table cell. The new results are shown below.

Totally Tables, Inc.		Average Price
Table Type	Construction	
Dining	Wood	$196
	Glass/Metal	$404
Coffee	Wood	$59
	Glass/Metal	$103
End	Wood	$61
	Glass/Metal	$107

➢ Other Style Attributes You Can Modify

These few examples have only scratched the surface as far as what you can do with ODS and the PRINT, REPORT, and TABULATE procedures. There are dozens of style attributes you can modify, and many places to use them within each procedure's output.

In general, the same style attributes are available for use with the STYLE= options for these three procedures. The table in Appendix A lists the attributes and what they do.

The examples in this chapter have shown how to modify individual cells, rows, columns, or headings. More examples of this type of modification are included in the "Guide to the SAS Output Delivery System" in the Online Documentation.

This paper has not covered how to apply a style attribute to the entire table or report. The syntax is actually quite simple. For PROC REPORT, use the following code:

```
PROC REPORT DATA=dataset
   STYLE=[style-attribute(s)];
```

For PROC PRINT, the syntax is very similar:

```
PROC PRINT DATA=dataset
   STYLE=[style-attribute(s)];
```

For PROC TABULATE, the syntax is different. You use a STYLE= option at the end of the TABLE statement:

```
TABLE <<page-definition,>
   row-definition,>
   column-definition
   / STYLE=[style-attribute(s)];
```

➢ Conclusion

I hope this paper has encouraged you to start experimenting with the new STYLE= options available for the PRINT, REPORT, and TABULATE procedures. The possibilities for output enhancement are endless. Have fun with these techniques.

➢ Acknowledgements

➢ Contacting the Author

Please direct any questions or feedback to the author at: info@laurenhaworth.com

APPENDIX: ODS STYLE ATTRIBUTES

Attribute	What it affects	Can be used for individual cells, columns, rows, headings?	Can be used for entire table?
ASIS	leading/trailing spaces and line breaks	✓	✓
BACKGROUND=	background color	✓	✓
BACKGROUNDIMAGE=	background image	✓	✓
BORDERCOLOR=	border color	✓	✓
BORDERCOLORDARK=	dark border color for 3-D borders	✓	✓
BORDERCOLORLIGHT=	light border color for 3-D borders	✓	✓
BORDERWIDTH=	width of border	✓	✓
CELLHEIGHT=	height of table cell	✓	
CELLWIDTH=	width of table cell	✓	
CELLPADDING=	space between cell text and borders	✓	
CELLSPACING=	space between cells	✓	
FLYOVER	text to display for mouse over (HTML)	✓	
FONT=	font definition (face, size, weight/style)	✓	✓*
FONT_FACE=	font typeface	✓	✓*
FONT_SIZE=	font size	✓	✓*
FONT_STYLE=	font style	✓	✓*
FONT_WEIGHT=	font weight	✓	✓*
FONT_WIDTH=	font width	✓	✓*
FOREGROUND=	text color	✓	✓*
FRAME=	table frame type	✓	
HREFTARGET=	window or frame to open for link	✓	
HTMLCLASS=	name of stylesheet to use	✓	✓
JUST=	horizontal justification	✓	✓
NOBREAKSPACE=	handling of spaces at line breaks	✓	
OUTPUTWIDTH=	width of table	✓	
POSTHTML=	HTML code to add at end of item	✓	✓
POSTIMAGE=	image to display at end of item	✓	✓
POSTTEXT=	text to display at end of item	✓	✓
PREHTML=	HTML code to add at beginning of item	✓	✓
PREIMAGE=	image to display at beginning of item	✓	✓
PRETEXT=	text to display at beginning of item	✓	✓
PROTECTSPECIALCHARS=	handling of <, >, and & characters	✓	
RULES=	lines between table cells	✓	
TAGATTR=	string to insert in HTML tag for the item	✓	
URL=	URL to link to when item is clicked	✓	
VJUST=	vertical justification	✓	

Paper P829

PROC SQL - Is it a Required Tool for Good SAS® Programming?

Ian Whitlock, Westat

Abstract

No one SAS tool can be the answer to all problems. However, it should be hard to consider a SAS programmer well versed in SAS, who does not use DATA steps. SQL should be classified with the DATA step rather than procedures because it is really a programming language in itself.

In the past many good SAS programmers have resisted learning PROC SQL on the basis that it is a database tool and that they can get along without it. It is time (or past time) to reconsider the question and change that belief.

SQL has dramatically changed the nature of what good SAS macro code looks like. It can simplify and standardize a number of common SAS programming patterns involving combinations of the DATA step, PROC SUMMARY, PROC SORT, and PROC PRINT.

This tutorial will focus on problem examples with code where PROC SQL has a distinct advantage in terms of code simplicity over use of the more traditional SAS tools mentioned above.

Introduction

The typical SAS programmer needs PROC SQL because:

1. It is superb at accessing data stored in multiple data sets at different levels.
2. It can easily produce a Cartesian product.
3. It can perform matching where the condition of a match is not equality.
4. It is good at summarization.
5. With the introduction of 6.11, it can make arrays of macro variables or do away with the need for these arrays by assigning a whole column of values to one macro variable.
6. Macro - SQL interaction enhances both macro and SQL.

In addition to the direct values listed above, one should not underestimate the value of SQL training in teaching one data organization. An example of how SQL can teach data organization is given at the end of the summarizing section.

Matching multiple data sets at different levels

PROC SQL provides a powerful tool when extracting data from different data sets at several different levels. It not only provides simpler code, it provides a new way of looking at these problems.

Suppose we have data at the state, county and city level stored in three data sets.

```
State (state, region,...)
county (cntyid, state, cnty, area,
...)
city (city, cntyid, area, pop,...)
```

Prepare a report of all cities in the midwest with populations over 100,000 with the ratio of the city area to the enclosing county area.

A SAS procedural solution demands that we decide whether to start with states or cities, specify all sorts needed for the various DATA step merges, and specify those merges in detail ending up with a PROC PRINT.

In contrast, SQL asks the fundamental questions:

1. What are the data sets?
2. What are the subsetting conditions?
3. What are the linking conditions?
4. What columns should appear?

```
proc sql ;
   select st.state ,
          cn.cnty ,
          ct.city ,
          ct.area  /  cn.area  as
arearato
      from state as st ,
           county as cn ,
           city as ct
     where ct.pop > 100000 and
           st.region = 'MW' and
           st.state = cn.state and
           cn.cntyid = ct.cntyid
   ;
```

```
quit ;
```

In all the remaining example code the PROC statement and the QUIT will be omitted.

Cartesian Product

Cartesian product matches are far more common than one-to-one matches, but the MERGE statement assumes one-to-one within BY-groups. To find a Cartesian product match, let's look at a codebook example. I have three data sets:

```
specs ( variable , format )
freq ( variable, format, value,
       count )
fmts ( format, value, label )
```

The report might look something like this.

```
first   variable   using   fmt1name
format

   value label count
     1   first  500
     2   sec      0
     3   rem    300

second   variable   using   fmt2name
format

   etc.
```

Before tackling the problem let's look at the code for joining SPECS and FMTS. The problem involves a Cartesian product because a format may be associated with more than one variable in SPECS, and formats typically have more than one value. Thus FORMAT does not determine a single record in either data set; hence a merge by FORMAT will not work. Note that accomplishing this sort of combining records in a DATA step involves using sophisticated SAS techniques when SQL is not used.

```
create view sfm as
select s.variable ,
       s.format ,
       fm.value ,
       fm.label
   from specs as s , fmts as fm
   where s.format = fm.format
;
```

It is easy to see how to produce the report with a DATA _NULL_ step when the right information is in a data set (VARIABLE, FORMAT, VALUE, LABEL, and COUNT). Here is the SQL code to produce the file.

```
create table report as
select
   coalesce (sfm.variable,
             fq.variable)
       as variable ,
   coalesce (sfm.format,
             fq.format)
       as format ,
   coalesce (sfm.value, fq.value)
       as value ,
   sfm.label ,
   coalesce ( fq.count , 0 ) as
count

   from sfm full join freq as fq
   on sfm.variable = fq.variable
      and sfm.format = fq.format
      and sfm.value = fq.value
   order by variable, format,
            value
;
```

Note that in two SQL statements we have done a lot of the work toward creating a codebook. If one could produce SPECS, FREQ, and FMTS easily, then one could produce a codebook for any properly formatted SAS data set. The FMTS file is trivially produced with the FMTLIB option of PROC FORMAT. The FREQ file requires some macro code. We will postpone discussion of the SPECS file to a later section.

Fuzzy Matching

Fuzzy matching comes in two varieties. In date (or time) line matches one file holds a specific date (or time) and one wants the corresponding record which holds a range of dates (or time). For SAS dates DATE, BEGDATE, and ENDDATE the WHERE clause might be

```
where date is
   between begdate and enddate
```

For efficiency reasons it is important to add an equi-condition whenever possible. In date (or time) matches one often has an ID that must also match, hence the equi-join condition becomes

```
where a.id = b.id and
   date is between begdate and
enddate
```

In the other kind of fuzzy matching one cannot trust the identifying variables. Suppose we want to match on social security numbers, SSN, but expect transposition errors

and single digit mutations. Now the WHERE clause might be

```
where sum ( substr(a.SSN,1,1) =
              substr(b.SSN,1,1) ,
            substr(a.SSN,2,1) =
              substr(b.SSN,2,1) ,
            ....
            substr(a.SSN,9,1) =
              substr(b.SSN,9,1)
          ) >= 7
```

To make this an equi-join we might add

```
      and substr(a.zip,1,3) =
            substr(b.zip,1,3)
```

or some other relatively safe blocking variable.

Summarizing

One PROC SQL step can do the job of a PROC SUMMARY followed by merging of the results with the original data. For example, suppose we have a weighted student sample including many different schools. We want the percentage weight of each student in a school. Then we might have:

```
select school , student , wght ,
         100*wght/sum(wght) as
pctwt
    from studsamp
    group by school
;
```

In this case one gets a message that summary data was remerged with the original data, but that is precisely what we wanted.

Now suppose we want to look at all the students from any school which has some student contributing more than 20% of the weight. The code might be

```
select stu.*
   from studsamp as stu ,
     ( select distinct school
         from studsamp
         group by school
        having wght/sum(wght) > .2
     ) as want
   where stu.school = want.school
order by stu.school, stu.wght desc
;
```

The technique is important because there are many times one wants to view every one in a group if anyone in the group has some property. SQL provides a natural idiom for producing the report.

Knowing SQL should make one more sensitive to bad patterns of storing data. For example, a common question on SAS-L is how to array data. Given the data

ID	DATE	COUNT
1	5jun1993	50
1	16oct1993	25
1	21dec1993	8
2	14may1990	16
2	27jan1991	3

how do you produce one record per ID with as many date and count fields as needed, say ID, DATE1 - DATE32 and COUNT1 - COUNT32? Another common question is how to work with the arrayed data. For example, how can you compute the rate of decrease in count per month and per year for each ID. The answer is a trivial SQL problem, when the data are stored as they were originally given.

```
select id ,
     (max(count)-min(count))/
intck('month',min(date),max(date))
        as decpmon,
     calculated decpmon * 12
        as decpyr
   from origdata
   group by id
;
```

After arraying it becomes a harder problem. Perhaps if SAS programmers learned SQL, and how to solve problems without arrays, then they would also learn the advantages of storing data in a non-arrayed form. With SQL training, one comes to realize the importance of putting the information into the data instead of the variable names. Of course, this also means that the usefulness of SQL is highly dependent on how well the data are stored, but it would be wrong to conclude that one might as well avoid learning SQL because of bad data management practices.

Macro Lists Via PROC SQL

PROC SQL's ability to assign a whole column of values to a macro variable has drastically changed how one writes macro code. Consider the splitting problem. Given a data set ALL with a variable SPLIT naming a member, split ALL by the variable SPLIT. Before version 6.11 one had to use CALL SYMPUT to create an array of data set names and values and then write a monster SELECT statement. The whole thing had to be in a macro in order

to repetitively process the array. Now one might view it as a problem to produce two lists

1. The names of data sets
2. WHEN / OUTPUT statements for a SELECT block

The first case is easily handled by

```
select distinct 'lib.'||split
       into :datalist separated
by ' '
  from all ;
```

The second is more of the same, only harder.

```
select distinct
        'when ('
     || split
     || ') output lib.'
     || split
   into :whenlist
             separated by ';'
   from all
;
```

Now the code to produce the split is trivial and need not even be housed in a macro.

```
data &datalist ;
   set all ;
   select ( split ) ;
      &whenlist ;
      otherwise ;
   end ;
run ;
```

In the section on the Cartesian product, we postponed discussion of the data set SPECS. It could be generated from one of the "dictionary" files documented in the Technical Report P-222. Suppose we are interested in making a codebook for the data set LIB.MYDATA, then the following code could generate SPECS.

```
create specs as
select name as variable
     , case
         when format=''
              and type='char'
            then $char
         when format=''
              and type= 'num'
            then best
         else  format
       end as format
   from  dictionary.columns
   where libname = 'LIB' and
         memname = 'MYDATA'
;
```

To prepare for doing the frequencies needed to make the data set FREQ we could use the array form of generating variables from a column.

```
select variable ,
       format ,
       into :var1 - var9999 ,
            :fmt1 - fmt9999
   from specs
;
%let nvar = &sqlobs ;
```

The frequency data sets can then be generated in a PROC FORMAT with the macro code

```
proc freq data = lib.mydata ;
   %do i = 1 %to &nvar ;
      table &&var&i /out=&&var&i ;
      format &&var&i &&fmt&i... ;
   %end ;
run ;
```

We still have not combined the frequency data sets into one data set, but that task can be left to a competent macro programmer, even one who doesn't know SQL (assuming that that is not a contradiction in terms).

Macro - SQL Interaction

The previous section showed how the making of lists has had a dramatic effect on the way one codes macro problems involving lists. Now we consider a more complex interaction between PROC SQL and macro, where macro code is used to write the SQL code in a loop and the whole problem is much easier, precisely because it is SQL code.

Suppose we have a data set, W, containing the variables NAME and GROUP.

NAME	GROUP
A	1
B	1
B	2
C	2
D	2
D	3
E	3
F	4
G	4
G	5
H	5

We want to collapse groups to the lowest level. For example, since A and B belong to group 1, and B and C belong to group 2, then all members of group 2 are part of group 1 because the groups have the common member B. Once this is seen one can add group 3 to the new group 1 because of the common member D. Thus group 1 covers A, B, C, D and E. Similarly F, G, and H ultimately belong to group 4. More formally, two groups are in the same chain if there is a sequence of groups containing the given groups such that each consecutive pair of groups contains a common name. Using this definition the data set consists of disjoint chains. The problem is to write a program identifying each chain by the minimum group number in the chain.

The intuitive argument given in the previous paragraph uses two kinds of minimization.

1. Find the minimum group (call it MINGROUP) for all names having the same value (e.g. NAME = 'B' has MINGROUP = 1).

2. Find the minimum of all MINGROUP values for all names in a common group (e.g. GROUP = 2 has MINGROUP = 1).

PROC SQL is very suitable to both types of minimization. In the first case we might have

```
create table t as
select name, group,
       min (group) as mingroup
   from dataset
   group by name ;
```

In the second case we might have

```
create table t as
select name, group,
       min (mingroup) as mingroup
   from t
   group by group ;
```

These two operations must be repeated over and over until no new minimums are found, since each new extension of a group may mean further collapsing. To express the iteration of this code to an arbitrary level, we need a macro %DO-loop. This time we will present the complete macro, %GROUPIT.

For generality, we make parameters to name the input and output data sets, and the variables represented by NAME, GROUP, and MINGROUP. The parameter MAX is added to insure that the macro does not execute for an excessively long time. (Since the algorithm does converge one could do away with this parameter or set it to the number of observations.)

```
%macro groupit
   ( data=&syslast,/*input data */
     out=_DATA_,   /*output data*/
     name=name,    /* name var  */
     group=group,  /* group var */
     mingroup=mingroup,
                   /* minimum var*/
     max=20  /*limit #iterations*/
   ) ;

/* ------------------------------
  minimize group on name and
  then group repeat until max
  iterations or done
------------------------------ */
%local i done ;

proc sql ;
   /* -------------------------
      initial set up - get first
      minimums, start numbered
      sequence of data sets
      ------------------------- */

      create table __t0 as
        select &name
             , &group
             , min (&group)
                  as &mingroup
          from &data
          group by &name
      ;

      create table __t0 as
        select &name
             , &group
             , min (&mingroup)
               as &mingroup
          from __t0
          group by &group
      ;

   /* ---------------------------
      iterate until done or too
      many iterations
      ------------------------- */

      %do %until (&done
                  or &i > &max) ;
         %let i = %eval (&i + 1) ;

         create table __t&i as
         select &name
              , &group
```

```
              , min (&mingroup)
                  as &mingroup
           from __t%eval(&i-1)
           group by &name
        ;
         create table __t&i as
           select &name
                , &group
                , min (&mingroup)
                    as &mingroup
             from __t&i
             group by &group
         ;

         /* are we finished? */
         reset noprint ;
         select w1.&name
            from __t%eval(&i-1)
                    as w1
                , __t&i as w2
           where w1.&name=w2.&name
             and &group=w2.&group
             and w1.&mingroup
                   ^= w2.&mingroup
         ;

         %let done =
           %eval ( not &sqlobs ) ;
         reset print ;
        drop table __t%eval(&i-1);
       %end ;/*end iterative loop*/

       %if not &done %then
%put WARNING(GROUPIT):Process
    stopped by condition MAX=&max;
       %else
       %do ;
           create table &out as
           select &name
                , &group
                , &mingroup
              from __t&i
              order by &name
                     , &group
           ;
           drop table __t&i ;
       %end ;

   quit ;
%mend  groupit ;

%groupit ( data = w
         , name = name1
         , group = group1
         , out = w2 )

proc print data = w2 ;
     run ;
```

Conclusion

I have pointed out six areas where SQL code excels. My conclusion is that a good SAS programmer can no longer ignore PROC SQL and remain good.

The author can be contacted by mail at:

Westat Inc.
1650 Research Boulevard
Rockville, MD 20850-3129

or by e-mail at:

whitloi1@westat.com

To Annotate or Not to Annotate, There Should Be No Question!

Keith Cranford, Marquee Associates, LLC

Abstract

The Annotate Facility within SAS/GRAPH is a powerful tool for enhancing your graphs. This is one of the tools that distinquishes SAS/GRAPH from other graphing software. This tutorial outlines the basic concepts of the Annotate Facility. Annotate macros, which are provided with SAS/GRAPH software and facilitate the use of annotation, are also discussed.

Introduction

The Annotate Facility within SAS/Graph provides a tool for enhancing graphics in data-driven form. Some of these enhancements include adding values to points on a line graph or histogram, providing custom labelling, or producing custom graphics from scratch. The Annotate Facility may be used in conjunction with other SAS/Graph procedures, or it can be used with PROC GANNO or PROC GSLIDE to control all the graphics.

Annotate Data Set

The Annotate Facility is comprised of a data set which contains graphic instructions. This data set consists of specially named variables, which define these instructions. By assigning values to these variables for each observation, you control

- Coordinate system that is used
- Task that is to be performed
- Position of the graphic objects
- Definition and position of graphic text

The coordinate system is controlled by three variables, XSYS, YSYS and HSYS. Each of these variables is a 1-byte character field. The XSYS variable indicates the horizontal axis coordinate system, the YSYS variable the vertical axis, and the HSYS variable the coordinate system used by the SIZE variable. There are three different coordinate systems:

- **Data**: uses the same coordinate system as the graphics
- **Procedure**: includes all the graphics output area between the titles and footnotes
- **Display**: includes the entire graphics output area

Table 1 gives some of the possible values of the coordinate system variables in terms of the different coordinate systems. Generally, it is easiest to work with the percentage values, since cell values can vary based on device.

Table 1: Coordinate System Values

XSYS/YSYS/ HSYS value	Coordinate System	Range
'1'	Data %	0% to 100% of Axis
'2'	Data value	Axis Min to Axis Max
'3'	Display %	0% to 100% of Display
'4'	Display value (in cells)	0 to Edge of Display
'5'	Procedure %	0% to 100% of Procedure
'6'	Procedure value (in cells)	0 to Edge of Procedure

The position of the graphics or text within the specified coordinate system is controlled by the X and Y variables. These variables control the horizontal and vertical axis locations, respectively. They are numeric variables with the range of values indicated in the table above.

The FUNCTION variable indicates the action to be taken, i.e., what you want to do. This is an 8-byte character variable with a default value of LABEL. Some of the possible values are

- LABEL: draws text at the specified location
- MOVE: moves the positional pointer to a specified location
- DRAW: draws a line from the current location to a specified location
- BAR: draws and, optionally, fills a rectangle
- POINT: draws a point
- POLY: begins drawing a polygon
- POLYCONT: continues drawing polygon
- SYMBOL: draws a symbol

The values of additional variables control other attributes of the Annotate instruction. These variables have fixed names, types and lengths, control different aspects of the graphics element depending on the value of FUNCTION, and are assigned default values if not included in the data set. These variables, along with their characteristics and default values, are given in Table 2.

Table 2: Descriptions of Annotate Variables

Variable	Type (length)	Default
ANGLE	Num	0
COLOR	Char (8)	First in device list
LINE	Num	1
POSITION	Char (1)	'5'
ROTATE	Num	0
SIZE	Num	1
STYLE	Char (8)	Depends on FUNCTION
TEXT	Char (<= 200)	Blank
WHEN	Char (1)	'B' (before)

Use of these attribute variables and the other Annotate variables can best be illustrated with the LABEL function. To place text at a specific location, assign FUNCTION the value 'LABEL' and assign values to the attribute variables to control the look of the text. Table 3 gives the possible and default values for these variables.

Table 3: Possible Values of Annotate Variables

Variable	Possible Values	Default Value
X,Y	Depends on coordinate system	Last values of X,Y
XSYS,YSYS, ZSYS	'1' to 'C'	'4'
POSITION	'1' to 'F'	'5'
SIZE	> 0	1
STYLE	Valid font name	'NONE'
COLOR	Valid color name	First in device list
ANGLE	-90 to 90	0
ROTATE	0 to 360	0
TEXT	Any text	Blank
WHEN	'A' or 'B'	'B'

The POSITION variable is coded to align the text. Table 4 provides possible values and alignments for this variable.

Table 4: POSITION Values

Vertical Pos	Right Align	Center	Left Align
One cell above	'1'	'2'	'3'
Half cell above	'A'	'B'	'C'
Centered	'4'	'5'	'6'
Half cell below	'D'	'E'	'F'
One cell below	'7'	'8'	'9'

The ANGLE variable specifies an angle of rotation of the entire text from horizontal. For example, ANGLE=-90 would print the text vertically (reading the text by tilting your head to the right). The ROTATE variable specifies an angle of rotation for each character in the text starting from the orientation after the text has been angled. Thus, ROTATE=90 would rotate each letter in the ANGLE=-90 example back to horizontal, so the text can be read down the page.

The WHEN variable specified when the text is printed, either before ('B') or after ('A') the graph is displayed. This determines what is displayed when there is an overlap between the text and graphics. The STYLE variable for the LABEL function specifies the font to be used. These can be SAS-supplied fonts, such as Swiss, Duplex and Simplex, or True type fonts, which are specified in quotes.

Annotate Macros

Annotate variables can be populated either through assignment statements, or by using annotate macros that are available through SAS/GRAPH. Each macro is controlled by a set of parameters, which determine the values assigned to the Annotate variables. The macros are accessible upon issuing the macro %annomac, which activates these macros. Some of the common macros listed below are more fully documented in the SAS/GRAPH Software: Reference, Volume 1, or in the online documentation.

- **%LABEL**: draws text
- **%MOVE**: moves without drawing
- **%DRAW**: draws a line from the previous point
- **%LINE**: draws a line between to points
- **%BAR**: draws a bar
- **%CIRCLE**: draws a circle
- **%SYSTEM**: sets the coordinate system
- **%SEQUENCE**: specifies when to draw

Each of these macros are controlled by a set of parameters. These parameters determine the values of the Annotate variables needed to produce the desired effect. Since a single macro call can produce multiple observations in the Annotate data set, this also makes for succint code.

Use with SAS/Graph Procedures

Annotate data sets can be used with SAS/Graph procedures to enhance the standard output of these procedures. Some simple uses are to add data values to line plots or bar charts and to add custom text to any graph. A more complex example is to use Annotate to overlay a line plot on a bar chart, or add a data table to a graph. In these cases, the data set that is used as input to the SAS/GRAPH procedure is also used to create the Annotate data set.

For example, you may want to label the points on a line plot. To do this, you need to indicate what function you want to perform (FUNCTION = 'LABEL'), where you want to place the label (X = horizontal-axis variable value, Y = vertical-axis variable value, and POSITION = centered above the point), and what you want to label (TEXT = vertical-axis variable value). The following code would accomplish this.

```
data data ;
   input X Y @@ ;
   datalines ;
5 8  6 10  7 11  8 14
;
run ;

data anno ;
   set data ;
   length function $ 8 text $ 15 ;
   retain xsys ysys '2'  when 'a' ;
   function = 'LABEL' ;
   x = x ;
   y = y ;
   text = left(put(y, best.)) ;
   style = 'swissb' ;
   size = 2 ;
   position = '2' ;
run ;

goptions htext=2 ftext=swissb ;
axis1 order=6 to 15 by 3 minor=none ;
```

```
axis2 order=5 to 8 minor=none
      offset=(5 pct,) ;
proc gplot data=data anno=anno ;
   plot y*x / vaxis=axis1
              haxis=axis2 ;
   title 'Figure 1' ;
   symbol i=join value=star ;
run ;
quit ;
```

This code would produce the graph in Figure 1.

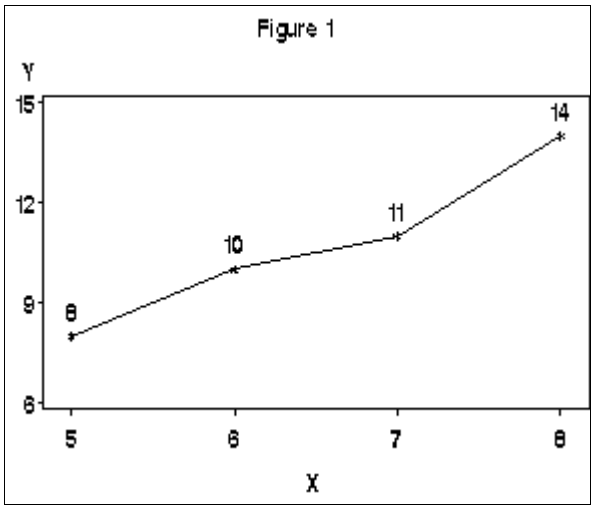


Figure 1: PROC GPLOT with Annotate

In the previous example, the assignment statements could be replaced by Annotate macros.

```
%annomac ;
data anno ;
   set data ;
   length function $ 8 text $ 15 ;
   retain xsys ysys '2'  when 'a' ;
   %label(x,y,left(put(y,best.)),black,
          0,0,2,swissb,2) ;
run ;
```

Also, the retain statement could be replaced by

```
%system(2,2,4) ;
%sequence(A) ;
```

Instead of labelling the data within the graph, you might include a table below the graph. In this case, you would use the display coordinate system and fix the position of the axes. This will allow you to know where to place the table in relation to the graph and the horizontal axis values. The following code would produce the graph in Figure 2.

```
%annomac ;
data anno ;
   set data(rename=(y=y1)) ;
   length function $ 8 text $ 15 ;
   retain xsys ysys '3'  when 'a' ;
   if _n_=1 then do ;
     %label(5,4,'X',black,0,0,2,swissb,B) ;
         %line(10,10,90,10,black,1,.5) ;
         %line(90,10,90,3,black,1,.5) ;
         %line(90,3,10,3,black,1,.5) ;
         %line(10,3,10,10,black,1,.5) ;
         do i=1 to 4 ;
           %line(10+i*20,3,10+i*20,10,
                 black,1,.5) ;
         end ;
   end ;
   j+1 ;
   %label(j*20,4,left(put(y1,best.)),
          black,0,0,2,swissb,B) ;
run ;

goptions htext=2 ftext=swissb;
axis1 order=6 to 15 by 3 minor=none ;
axis2 order=5 to 8 by 1 minor=none
      offset=(10pct,) origin=(10pct,20pct)
      length=80 label=none;
proc gplot data=data anno=anno ;
   plot y * x / vaxis=axis1 haxis=axis2 ;
   title 'Figure 2' ;
   symbol i=join value=star;
run ;
quit ;
```

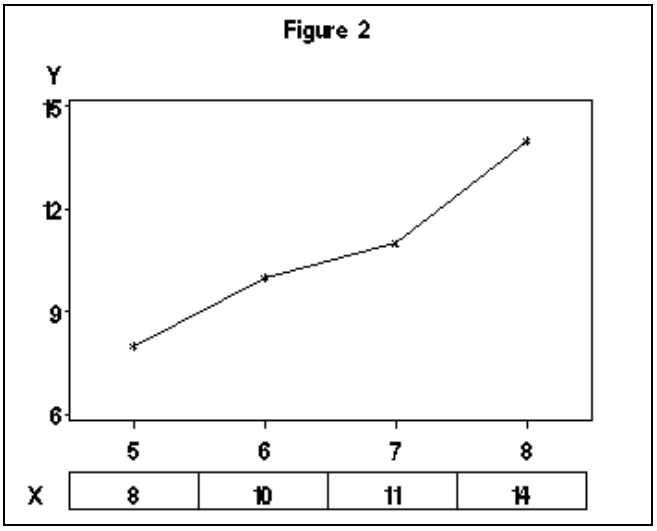


Figure 2: PROC GPLOT with Table

Use with GANNO or GSLIDE Prodecures

The Annotate Facility can also be used to produce custom graphics using the GANNO or GSLIDE procedures. In this case the Annotate data set will be used to issue all the instructions to produce a graphic. This may be in the form of a more traditional graphic output such as a line plot, bar chart or pie chart, or it may be a custom graphics table.

For example, the Annotate Facility could be used to produce an invoice containing a customers address information as well as purchase detail. Suppose customer data was stored in a data set called CUSTOMER and the invoice detail in a data set called INVOICE. These two data sets are then linked by the customer_id variable. The following code would produce the invoice graphic output in Figure 2.

```
proc sql ;
   create table detail as
   select *
   from customer as a, invoice as b
   where a.customer_id=b.customer_id and
         invoice_no=1 ;
quit ;

%annomac ;
data anno ;
   set detail end=last ;
   length text $ 30 ;
   retain xsys ysys '2' when 'a' ;
   if _n_=1 then do ;
     %label(5,95,name,black,0,0,2,'Arial',C) ;
     %label(94,95,'Inv #'||put(invoice_no,3.),
            black,0,0,2,'Arial',A) ;

     %label(5,90,address,black,0,0,2,
            'Arial',C) ;
     %label(5,85,city_st_zip,black,
            0,0,2,'Arial',C) ;
     %line(5,80,95,80,black,1,.5) ;
     %line(5,72,95,72,black,1,.5) ;
     %line(5,28,95,28,black,1,.5) ;
     %line(95,80,95,20,black,1,.5) ;
     %line(95,20,5,20,black,1,.5) ;
     %line(5,20,5,80,black,1,.5) ;
     %label(6,74,'Item',black,0,0,2,
            'Arial',C) ;
     %label(30,74,'Quantity',black,0,0,2,
           'Arial',C) ;
     %label(70,74,'Unit $',black,0,0,2,
            'Arial',A) ;
     %label(94,74,'Total $',black,0,0,2,
            'Arial',A) ;
     %label(6,30,'Tax',black,0,0,2,'Arial',C) ;
     %label(6,21,'Total',black,0,0,2,
            'Arial',C) ;
     y1=70 ;
   end ;
   y1+(-5) ;

   %label(6,y1,item,black,0,0,2,'Arial',C) ;
   %label(40,y1,put(quantity,8.0),black,0,0,2,
          'Arial',A) ;
   %label(70,y1,put(unitcost,8.2),black,0,0,2,
          'Arial',A) ;
   %label(94,y1,put(quantity*unitcost,8.2),
          black,0,0,2,'Arial',A) ;
   totcost+(quantity*unitcost) ;
   if last then do ;
      tax=round(totcost*.08,.01) ;
      %label(94,30,put(tax,8.2),black,0,0,2,
             'Arial',A) ;
      totcost + tax ;
      %label(94,21,put(totcost,8.2),black,0,0,2,
             'Arial',A) ;
      %label(5,12,'Balance Due Upon Receipt',
             black,0,0,2,'Arial',C) ;
   end ;
run ;

proc ganno anno=anno ;
run ;
quit ;
```

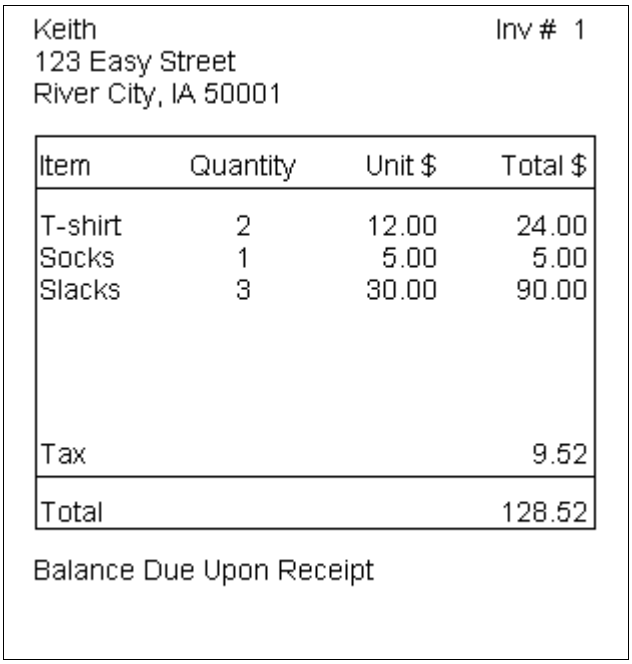


Figure 3: PROC GANNO Example

Some Helpful Suggestions

The following suggestions or guidelines can help make using the Annotate facility easier and more useful.

- Let SAS/GRAPH procedures do as much as possible
- Break the annotation into groups
- Use the ORIGIN= option to anchor the graph

The first suggestion is a time saver. The SAS/GRAPH procedures provide a lot of flexibility and features. Take advantage of this functionality. Before adding any annotation the basic graph should first be refined. In most cases, this is enough and no annotation is

needed; however, when enhancements are needed the amount of annotation is kept to a minimum.

This leads to the second suggestion. It is best to work in pieces by defining groups of annotation, such as a table, title or labeling, and then add one annotation at time. Doing this will allow you to concentrate development in one area and will also help in debugging.

A helpful hint is using the ORIGIN= option on the AXIS statement. This allows you to anchor your graph, which helps in placing annotation outside the graph, such as tables.

Conclusion

The Annotate Facility in SAS/Graph is a very powerful tool that can transform your ordinary SAS/Graph output into something extraordinary. Its use hinges on an understanding of the Annotate data set, which has a definite structure. Once this is accomplished, there really should be no question as to whether it should be used.

Keith Cranford
Marquee Associates, LLC
2101 Wychwood Dr.
Austin, TX 78746
kcranford@marquee-assoc.com
www.marquee-assoc.com

How Fast Can You Type
or
Go Ahead and Get Snippety

John Charles Gober
U.S. Department of Commerce, Bureau of the Census

Introduction

When this paper was first envisioned, a number of titles were considered: Creating a Better Toolbox, Its OK to Plagiarize, Programming for Power, etc. However after discussing the concept with my fellow technical experts they all said the same thing. It all comes down to one thing. Time is money, and how fast can your people come up with the necessary code and type in those needed statements is the critical factor. Can you do the application in three weeks, three days, or three hours?

Whether the office you work in employs a single programmer or hundreds of programmers containing the complete spectrum of experience levels it always helps to have pre written code you can draw upon. Code a junior programmer can learn from. Code the experience programmer can easily adapt to suit their needs. Little pieces of code or snippets that can be copied, modified, or used as a learning tool. Code already located in a common area so you don't have to reinvent the wheel.

Scenario

It is 9:00 Friday morning on a Holiday weekend. You have been tasked with the following assignment which needs to be completed by Monday morning: Create a series data steps or SAS® procedures that will read a control file containing Fips State and Fips County codes for the entire United States. Using this control file, do a check for each State to see if that State has a full complement of County files with a name pattern of ssu{ssccc} where ss represents the state and ccc represents the county. If yes, rollup the County files to a State file called ssu_{ss} and upload the original County files and the rolled up State level files to a remote platform. If not, e-mail your supervisor that there are some States with missing county files and that those State files containing the missing Counties could not be created. All files are SAS Version 8. The control file will have two variables call FIPST and FCNTY and the program will execute individually for each State.

None of this code is very difficult. Read a file, read a directory, a few if then else statements, and email, and an upload program. However, if you have never done an application similar to this it may take a few minutes to just come up with the ideas let alone the code. Fortunately examples of almost all of the code you need for this task has already been placed in a toolbox. You will be done by noon. Details to follow.

Input Snippets

Here are examples of several input snippets that might be useful for our scenario. The first one reads the files in a directory using Unix commands invoked by shelling out to operating system by use of the SAS 'X' command. This is used more for reading the names of non-SAS files but I thought I would throw it in. The second snippet is very similar to the first but uses the SAS filename statement and the pipe engine. The third snippet uses PROC SQL to create a view table

containing the SAS dataset names. All SAS dataset are assumed to be in the pre assigned library mylib. All data is stored in the directory /ssu/.

```
*Getting file list using x command;
*In Unix, 7th word from ls command;
*contains filename.;
*Best for non-SAS files.;
   x 'ls -lR /ssu/*.sas7bdat >
flist';
   data read1;
      infile 'flist' lrecl=100 pad
         missover;
      input (v1 - v7) ($);
      fipst = substr(v7,1,2);
      fcnty = substr(v7,3,3);
   run;

*Getting file list using pipe;
*In Unix, 7th word from ls command;
*contains the filename.;
*However may be flavor dependent.;
*Best for non-SAS files.;
   filename flist pipe
       'ls -lR /mylib/*.sas7bdat';
   data read1;
      length v1 - v7 $20;
      infile flist lrecl=100 pad
         missover;
      input (v1 - v7) ($);
      fipst = substr(v7,1,2);
      fcnty = substr(v7,3,5);
   run;

*Getting filelist using the;
*SASHELP.VTABLE;
  data read1;
     set sashelp.vtable;
     if libname='mylib' and
     substr(memname,1,4) = 'SSU' and
     memname ^= 'SSU_04';*arizona;
     fipst = substr(v7,1,2);
     fcnty = substr(v7,3,3);
   run;
```

The last method is my preferred method since you are using one of SAS's many behind the scene administrative files. It also gives a chance for less experienced programmers to see what SAS has hidden away in other administrative that they might not have found out otherwise.

An Email Snippet

In a production environment, in my opinion, there is not a better way of notifying people of a job status. This is especially true when you have hundreds of jobs a day executing. Messages that range from '*MR. Smith, Your weekly status reports are ready.*' to '*Warning!!! Job xyx123 has possible data anomalies and has been halted* '. Below is an email snippet from our system, which is easily modifiable to suit our needs. Just break out the code you need and insert.

```
*Email to two recipients;
*with attachment;
filename pgmmail email
      "recipient.one@yourworkplace
      recipient.two@yourworkplace"
       subject="SAS email test"
       attach="/ssu/attach.txt";
data _null_;
   file pgmmail;
   put "this is a test.";
   put "more lines, etc.";
run;

*email info can also be placed on;
*the file statement.;
file pgmmail
to=("recipient.one@your.workplace"
   "recipient.two@your.workplace")
subject="SAS email test"
attach="/ssu/attach.txt";
```

An item to note that with mail, depending on your mail application, messages can also be sent to group addresses. This is a very nice feature for contacting support teams and help desks.

A Snippet to Create a String of Datasets

According to the SAS Map viewtable there are 3,143 Counties in the United States. Since these Counties at to be 'rolled up' to the State level this means the possibility that someone will have to manually type in those thousands of County datasets into their programs. Two ideas on how to accomplish this task come to

mind. The first idea uses a data step routine to read a view containing the State and County variables for all the Counties in a State. These two variables are concatenated together along with a prefix representing a library name. This new variable is then retained and added together with the same variable from the next dataset read. This creates the extremely long variable representing all of the state county combinations found in the input view. Note the length of the variable 'filestring' which means that this snippet can only be executed in SAS Version 8.

```
data temp;
   length filestring $32000;
   retain filestring ' ';
   set sashelp.vtable end=last;
   where lowcase(libname) = 'mylib'
     and substr(memname,1,4) = "ssu_"
     and memname ^= 'ssu_us';
   x='mylib.ssu' || put(memname,$8.);
   filestring = x||' ' ||
      left(filestring);
   if last then call
symput('filestring',filestring);
run;
```

The second snippet does the exact same thing as the snippet above except that it uses PROC SQL and the ability to automatically create a macro variable containing all of the state/county combinations in a view.

```
proc sql noprint;
   select distinct "mylib." ||
      trim(memname)
    into: filestring separated by ' '
    from sashelp.vtable
    where lowcase(libname) = 'mylib'
       substr(memname,1,4) = "ssu_"
          and memname ^= 'ssu_us';
quit;
```

Notice the use of the colon to create a macro string in the PROC SQL example. Both sets of codes above should also have their records subset by State to make the scenario run correctly.

An Upload Snippet

Using proc upload is relatively easy to use. You can upload text files, SAS datasets, or even entire SAS libraries. I am including it here just to show a relatively unknown feature of PROC UPLOAD and PROC DOWNLOAD. The ability to group and upload/download SAS datasets by prefix. The following example will upload all datasets with the prefixes of 'ssu_'. This is a great advantage for uploading multiple files however one must be very conscious of dataset naming patterns.

```
*libname for local;
 libname one '/mylib';
 signon;
 rsubmit;
   *libname for
   libname two '/mylib/';
   proc upload in=one out=two;
      select temp: txp: ;
   run;
 endrsubmit;
 signoff;
```

A Snippet to Subset a SAS Dataset

This snippet is not needed for our scenario but might be useful for other applications so I thought I would also include it in this paper. It is not very efficient and can definitely be improved but the general concept is one I thought might be useful. Actually it is more of an entire program than a snippet but still warrants some attention. However since it is also rather long I will include the code at the end of the paper after we piece together all the other snippets.

Conclusion

It does not matter if you are an experienced programmer or one who just dabbles. It also does not matter if you are the only SAS programmer in your office or one of a hundred. Having a centralized location where you can access ideas, techniques and code has

great benefits. Foremost it saves time and effort when coding. Secondly it is a good learning tool. A toolbox can contain small pieces of adaptable code, macro routines, example of functions, and redundant code. It is something every office needs to create and to serve its full potential it must be dynamic.

Credits and Acknowledgments

This paper contains programming code and techniques other than those developed by the author. Many people from the U.S. Bureau of the Census have contributed and acknowledgement is duly given to all.

Information has also been collected and modified from SAS OnlineDoc®, Version 8.

SAS Institute Inc., SAS OnlineDoc®, Version 8, Cary, NC: SAS Institute Inc., 1999.

***

```
*PROGRAM TO DEMONSTRATE SNIPPETS;
*THAT CAN BE USED TO ROLL UP    ;
*DATASETS.;
*ARIZONA IS THE TEST STATE. BE  ;
*CAREFULL ABOUT LEADING ZEROS   ;
*AND NAMING CONVENTIONS WHEN    ;
*WRITING YOUR OWN.;

*GET STATE NUMBER FROM SCRIPT FILE;
%let state = %sysget(STATE);

*OR HARDCODE THE VALUE;
*%let state = 4;

*ADD LEADING ZERO TO STATE;
*WOULD NOT NEED IF BETTER NAMEING;
*CONVENTION IS USED;
data null;
   call
symput('ST',put("&STATE",z2.);
   stop;
run;

*GET LIST OF ALL COUNTIES IN STATE;
data allctys(keep=state county);
   set
maps.cntyname(where=(state=&STATE));
run;

*GET LIST OF ALL FILE IN DIRECTORY;
data allfiles(keep=state county);
   set sashelp.vtable;
   if upcase(libname) = 'MYLIB' and
      upcase(substr(memname,1,4)) =
'SSU_' and
      upcase(memname) ^= "SSU_&ST";
   state =
input(substr(memname,5,2),2.);
   county =
input(substr(memname,7,3),3.);
run;

*DETERMINE IF ALL FILES ARE PRESENT
IN DIRECTORY;
data _null_;
   length found $2;
   retain found ' ';
   merge allctys(in=base)
         allfiles(in=trans)
            end=last;
   by state county;
   if base and not trans then found =
'no';
  *if last and found = 'no' then
abort abend 123;
run;

*CREATE STRING OF ALL FILE IN
DIRECTORY;
proc sql noprint;
   select distinct "mylib.ssu_&ST" ||
trim(put(county,z3.))
    into : filelist separated by ' '
    from maps.cntyname
    where state = &STATE;
quit;
*THIS SQL WILL CREATE THE MACRO
VARIABLE '&FILELIST';
*WHICH CONTAINTS THE VALUE
'mylib.ssu_04001 mylib.ssu_04003
  mylib.ssu_04005 mylib.ssu_04007
mylib.ssu_04009 mylib.ssu_04011
  mylib.ssu_04012 mylib.ssu_04013
mylib.ssu_04015 mylib.ssu_04017
  mylib.ssu_04019 mylib.ssu_04021
mylib.ssu_04023 mylib.ssu_04025
  mylib.ssu_04027';
```

```
*PERFORM ROLLUP IF ABEND DID NOT
OCCURR;
data mylib.ssu_&STATE;
   set &FILELIST;
run;
                ***

*PROGRAM TO SEPARATE DATASET INTO ;
*COMPONENTS;

*GET STRING OF ALL FILE IN DIRECTORY;
libname in1 '/acsw';
libname out1 '/hm/gober001';
*MASTER FILE IS ON THE US LEVEL;
*MUST SUBSET INTO SEPARATE STATE
FILES;

*MASTER FILE IS CALLED CO2000 AND ;
*HAS VARIABLE CALLED Fipst.        ;
*STATE LEVEL FILE WILL BE CALLED
ST_{STATE};

*create list of output datasets;
proc sql noprint;
   select distinct "out1.ssu_" ||
trim(fipst)
    into : filelist separated by ' '
    from in1.co2000;
quit;
%put &filelist;

*CREATE CODE TO OUTPUT RECORDS;
proc sql noprint;
   select distinct "if fipst = '" ||
trim(fipst) || "' then output
out1.ssu_" || trim(fipst) ||" ;else"
    into : pgmcode separated by ' '
    from in1.co2000;
quit;

data &filelist;
   set in1.co2000;
   &pgmcode;;
run;
```

Author Index

Author Index

AUTHOR INDEX

P

R

S

T

V

W

X

Y

Keyword Index

Keyword Index

KEYWORD INDEX

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z