PROJECT DESCRIPTION

In late November 1985, the author was asked to implement a programming system that would analyze U.S. Census Bureau data. The purpose of the project was to ascertain the economic activity of smaller governmental units (towns and counties). For competitive reasons, the gross sales of industries in smaller governmental units is often suppressed. Members of the Sociology Department working with a Chicago area market research team, developed algorithms to estimate the suppressed gross sales figures. The raw data comes from three sources: the 1980 population census, the 1982 retail census and the 1982 service industry census.

Project specifications consisted of 30 typewritten pages. The overall development timetable was approximately six months.

LANGUAGE SELECTION PROCESS

Before approval to begin this project was obtained, we were required to submit a general project timetable along with associated costs. To develop a reasonable timetable we would need to select a programming tool. The project required a language that would:

(a) Allow rapid code development.
(b) Assist in data management and statistical functions.
(c) Be relatively easy to debug.

I became an early advocate of the use of a declarative or 4-GL language for this project. SAS appeared to be a natural choice, with its strong data management capabilities combined with its statistical orientation.

There was initial resistance to building a project of this size around what had been considered a report-generation package. We had little experience in large SAS projects in the way of programming standards, debugging tools and design methodology. In fact we finally choose SAS as our programming vehicle for these very reasons. It was felt that 4-GL languages would be the predominate programming tools in a few years, and that it would be strategically important to develop skills using these languages in large scale projects.

The overall project was divided into four phases: Requirements Definition, Design, Program Construction and Software Design.

REQUIREMENTS DEFINITION

The overall aim of this project was to estimate suppressed retail and service sales figures from smaller governmental units. Input consisted of U.S. Census Bureau data. Reports summarized input data and highlighted missing values. The product of this system is a diskette for each state, with all suppressed values estimated. No design changes were generated by the selection of SAS as the programming language vehicle.

DESIGN PHASE

Top down or bottom up? Low level languages (COBOL, BAL, FORTRAN) often profit from a bottom-up design approach as designed routines increase the richness of the language set. SAS procedures are effectively one-line implementations of pseudo-code statements, thus it would appear a top-down design approach would be reasonable. As is usually the case, a compromise worked the best.

Initial large scale design indicated the need for a group of small reoccurring functions. Often large projects profit from the construction of a "tool-kit", a group of small, basic subroutines that effectively expand the functions of the language. The "tool-kit" for this project consisted of I/O routines, variable management procedures, debugging aids and dataset management aids. All tool kit routines were implemented as SAS external macros.

Obviously, design was an iterative process, alternating between functional design and tool kit construction.

PROGRAM CONSTRUCTION

It was clear that this project was too large for a single SAS job. Once a project is divided into multiple jobs, another level of complexity is introduced into the process. How large should each job be? How do jobs communicate? How are errors processed? What information should be retained across job invocations? Clearly job segmentation was not going to be a trivial task.
For ease of maintenance it would be advantageous to segment SAS code into separate processes by high level function. For the most part this was followed if the estimated size of the function would be under 500 lines of code. 500 lines was an arbitrary number, but near the point at which we felt that the module was becoming unwieldy.

Inter-job information communication was accomplished by retaining SAS datasets in "pseudo-permanent" datasets. These were OS files created at the invocation of the job-stream and deleted at the completion of the job stream. In effect, three classes of datasets were created: work, for truly temporary information, passed, for information required by another module in a later job, and permanent for the project output.

Since the jobs were chained, or self-scheduling, a mechanism was designed to cause severe errors to terminate the job stream.

STANDARDS

Everyone agrees that standards are necessary. Unfortunately, human nature is to either resent standards or to ignore them. Programming is often too creative of an activity. It is rare though for a programmer to consistently have the same creative train of thoughts. Standards apply a logical structure to this creative activity allowing easier design and maintenance.

Specified standards fell into four areas: data set names, variable and array names, macro's and include construction and program style.

Eventually over 400 individual SAS datasets were created by the data analysis programs. These datasets were either temporary (work), informational (communication) or permanent (output). The same naming conventions were followed for all three classes of datasets:

C J P T xxxx
| | | | |...... Descriptive
| | | | |...... Task Number (from design)
| | | | |...... Process ID (From design)
| | | | |...... Creation Job ID
| | | | |...... Data Set Class

Thus C2CRMMEAN would be a inter-job communication dataset created by the second job of the stream. The overall design for the system assigned process and task identifiers to each system activity. These appear as the third and fourth characters of the dataset name (Process C, Task R), the remaining four characters are available as descriptions.

A naming convention for variable names was also developed. The actual convention was very closely related to the nature of the processed data with variable names composed:

AA B xxxx
| | | | |...... Descriptive
| | | | |...... Data Form (Raw, Summary)
| | | | |...... Data Source

Thus the variable RCRTOTR would represent the retail county raw total retail value. Array names are differentiated by the appending of an underscore to the end of the description. RCSEST would represent an array of retail, county level summary information.

Source include members follow a naming convention based on the dataset standards. The first two characters describe the process and task preformed by the include, the remaining characters are available as descriptors. Source includes were used primarily to call in "tool-set" I/O routines.

Macros were used extensively in this project, primarily as an extension to the SAS language, providing a meta-language designed for this project. Macros created for internal usage in one job were prefixed with an I, general tool-kit macros for usage in multiple jobs were prefixed by an X.

All external macros and source includes were grouped into a single common library.

All program source basically adhered to general SAS programming construction guidelines. DATA and PROC statements began in column 1 or 2, all statements within the range of a DATA or PROC statement were indented. Each process or task began with a block comment identifying the process and task numbers and containing a short description of the activity. Individual lines of SAS code were documented at the programmer's
discretion, in cases were the intent of the program logic was not entirely obvious.

SOFTWARE DESIGN

The only successful method to debug/test a program of this size is to incorporate a debugging superstructure into the system design. Two macros were developed to aid in this process. The first macro, XDPRINT would execute a PROC PRINT only if a global macro variable was 1. The second macro XDXEQ would invoke a debugging routine if a second macro variable was "turned-on". This allowed debugging output generators to be designed into the program. Extraneous output could be switched off as routines were tested, or turned back on to receive additional information.

/* Turn on all debug options*/
%DEBUG(ON,ON)
.
SAS Program Code
.
%XDPRINT('FA: Output from Second Merge')
.
More SAS statements
.
%XDXEQ(XDRANGE)

The large volume of printed output generated when debugging output was enabled encouraged the development of output labeling standards. All output, whether produced by a procedure or through PUT statements had an associated title. Output produced by debugging statements was directed to a special output group (SYSOUT). Exceptions to normal processing conditions appeared both on the LOG and on a special ERROR sysout group. The use of multiple output groups minimized the amount of time necessary to determine the validity of a job run.