Abstract

The SAS® System allows users to write programs in third generation languages (PL/I, C). These programs are interfaced to the SAS supervisor and operate using the same familiar SAS syntax.

The SAS System to SYSTEM 2000® DBMS interface procedures are general purpose tools that serve a wide variety of uses. However, they may not be suitable for some situations. Recovery and error checking are important in many update situations.

We consider a program that must update the employee database from information supplied as a SAS data set. The program itself is quite simple: it updates the employee record based on the employee number. However, it does illustrate all the points that need addressing when writing a user-written procedure.

All you need to know about writing user written procedures is documented in SAS Programmers Guide for PL/I, Version 5 Edition.

Outline

The program used as an example in this paper was written for the SUGI demonstration system. Complete listings are available from the author, c/o SAS Institute Inc., Box 8000, Cary, NC 27511.

1. Getting started.
2. The benefits of a custom SAS procedure.
3. The SAS environment.
4. The design of the program.
5. The GRAMMAR processor.
6. Data definitions.
7. Establishing connection with the SAS System.
9. Accessing data from SAS data sets.
10. PLEX calls.
11. Processing the data set.
12. Termination.
13. Testing the procedure.

Getting started

In order to tackle a project of this nature, you should have access to:

- The PL/I compiler, and know how to invoke it at your site.
- The linkage editor, and know how to invoke it at your site.
- The PLEX precompiler, and know how to invoke it at your site.
- The user-written procedure modules, and source samples. These are available on the SAS installation tape, but are often not installed "on-line."
- Some free time!

Benefits

Why go to the trouble of writing a program as a SAS procedure?

- The syntax will be familiar to existing SAS users, and the PROC that you write will be capable of exchanging information with all other SAS procedures.
- The SAS Display Manager environment is a comfortable environment for end-users - a reporting procedure for a SYSTEM 2000 data base could make use of the full-screen procedure output window, allowing the end user to scroll the results without any extra work required from the procedure implementor.
- Writing a custom SAS procedure will give you an understanding of the SAS system that is hard to acquire elsewhere.

The SAS environment

All SAS procedures are dynamically loaded for execution. When the SAS System encounters the PROC statement (PROC FOOBAR for example), it will load the module FOOBAR, and invoke it. FOOBAR in turn calls an entry point within the supervisor to establish the addresses of service routines that the PROC will call to accomplish various tasks. The SAS supervisor provides services for statement parsing; access to SAS and external data sets; writing message to the SAS log and print file; memory; and various utility functions. The collection of supervisor interface routines are known as the X-routines, and they share the common prefix "X."

The user-written library has another collection of "useful" functions, referred to as sublib functions. These do not always access the supervisor, but do perform useful tasks. Both the X-routines and the sublib functions are documented in the SAS Programmers Guide for PL/I, Version 5 Edition.
The SAS supervisor does not provide access to SYSTEM 2000 PLEX functions, so these are coded in the same fashion as for stand-alone programs.

The design of the program

When designing any program you need to consider the inputs, outputs, and how to specify the options that the program should understand. It is a SAS convention that program options are specified in the PROC statement as well as in the statements that the procedure will recognize. We will need to understand how to interface with the SAS supervisor so as to retrieve values specified by the user of our program.

The input to our program is a SAS data set, with a variable identifying the employee number, and others that contain the values with which we wish to modify the data base. Consider the input data set a "transaction data set" (built using SAS full-screen procedures) that we must validate before applying transactions.

The output of our program is to a SYSTEM 2000 data base. Using values from the transaction data set, we will update the employee data base that is used in the SYSTEM 2000 installation verification process. The program will also write informatory and error messages to the SAS log.

The steps our program must accomplish are: retrieve specified options; open input data set and set up variable definitions; open data base; process each record; close data base; return to the SAS supervisor.

The GRAMMAR processor

A grammar is the set of language constructs that define the features of the procedure. The SAS supervisor uses the grammar tables to parse the user's statements and detect syntax errors. Version 6 of the SAS System also uses the grammar for spelling error recovery - it can examine the list of potential tokens looking for an "almost match."

Grammars are specified to the SAS System using a language, which is processed by the grammar preprocessor, to produce a module that can be link-edited to the procedure. The grammar specifies options that the PROC will recognize, as well as which data set to process.

Grammars consist of four elements:

1. Rules - are used for modularizing the grammar. Rules expand into their definition when processed by the grammar processor.
2. Terminals - character strings in quotes. These are tokens that are required by the grammar.
3. Actions - specify what to do with the information. Typically the action tells where to place a value in the grammar data structures so that the PROC can process it later.
4. Lexicals - are placeholders for "expected data-types" for example a password specified as PASS = "quoted string" would be coded in a grammar as:

   "PASS" = OPARM(4,3,8) QS

and would place the value that the user specified into the fourth position of the statement structure.

In the example below, our PROC is called PERSUP, and will accept three options.

1. The data set to be processed. There is a SAS convention that if the user does not specify the data set to process, then the system will process the last one created in this session. This information is reflected in the grammar with the @DSDFLT action.
2. A Multi-user indicator. If the user specifies "MULTUSER" or its abbreviation, as defined here ("MU") then the PROC will open a multi-user data base. The $ suffix tells the supervisor not to print MU as one of the tokens it was expecting, and is a useful scheme for making secret options!
3. A debug indicator. While we are debugging the PROC we can set the debug indicator and dump information to the SAS log as appropriate.

```
%include stubsrc.
A---------STATEMENTS TO PARS---------------#
PROGRAM = PERSUPSTMT ENDB,
#-------- PERSUP SPECIFIC STMTS ---------#
PERSUPSTMT = @PROCINIT @SYMINIT(8)
   "PERSUP" @DSDFLT (1,1,1) @STMTEND ,
PERSUPOPT = ("DATA" "=" DSFIELD) @STMTINIT(8) 1 "DEBUG" @OPT(1) 1 ("MULTUSER" "=" MU$) @OPT(2).
```

Grammars, and the parameters to the semantic actions are discussed in chapters 2, 3 and 4 of the "SAS Programmer's Guide for PL/I, Version 3 Edition."

Data definitions

We need to define data structures to access the information collected by the grammar processor, as well as other structures used for information about SAS variables, SAS data sets and the like. In addition, since we are using more than one independently compiled module, I have chosen to define a structure for passing information backwards and forwards between modules. One could use function/procedure arguments, but this approach seemed easier.

The abbreviations FXL, FTL are defined in the MACPORT include file and are used for definitions of double precision variables.

First, we declare the entry points for the SAS interface routines. You are quite at liberty to type them all out, but it is easier to include them from files supplied on the SAS installation tape. You will need to override the //SYSLIB DD ... specification in your PL/I compile job stream.
In the interest of modular programming, the PERSUP procedure is broken up into three modules: the grammar tables; the interface to the SAS system; and the interface to the SYSTEM 2000 DBMS. Rather than passing many arguments back and forth between these modules, I decided to define a global data structure and pass its address.

*--------------------------------------------------
* THE INFOSTR IS IN EFFECT A GLOBAL STRUCTURE, USED
* TO PASS INFO BETWEEN MODULES. OF COURSE, ONE COULD
* CODE EACH INDIVIDUAL ARGUMENT.
*--------------------------------------------------*
DCL INFOPTR POINTER;
DCL INFOSTR, OPTS, DEBUG BIT(1), RSVD(7) BIT(1), OPCODE CHAR(6), ACCESS FIXED BIN(31), RC FIXED BIN(31), FEEDBACK CHAR(20), EMPNO FIXED DEC(4), EMPSTATUS CHAR(9), EMPOFFEXT CHAR(9), EMPADDRESS CHAR(20), EMPCITYSTATE CHAR(15), EMPZIP CHAR(5);
INFOPTR=ADDR(INFOSTR);

/* LOCAL VARIABLES */
DCL XEMPNO, XEMPSTATUS, XEMPOFFEXT, XEMPADDRESS, XEMPCITYSTATE, XEMPZIP CHAR(16);

Connecting with the SAS System

When the SAS supervisor loads our PROC, one-way communication is established — it called us. We have to call predefined entry points to establish connection with the supervisor so that we may call it.

CALL XINIT();
CALL XCOMGET();
CALL XEXIT();

At this point, the statements for our PROC need to be parsed. We call the supervisor to do this task for us, but it needs to know the address of the grammar tables; the address of which is returned by the module generated by the grammar processor.

CALL XPARSE(PERSUPG());
IF (PROCERROR > 0) THEN DO;
  CALL XEXIT(PROCERROR); END;
 CALL XTOKE();
STMTPTR=PROCPTR;
The SAS supervisor will fill in the fields of the STMTSTR, given values specified by the user in his source statements. The supervisor takes care of converting numeric values and other such chores. The STMTSTR is a general purpose data structure that contains overlays of the different data types supported by the grammar.

Accessing specified options

It is up to the PROC writer to examine the statement structure for all fields that he would expect to be there. In our PROC, this means looking at OPT(1) — the debug indicator, OPT(2) — the multi-user indicator, and the address of the input data set in field 1.

/ *-- DEBUG OPTION SPECIFIED ? ----------------*/
DEBUG=STHTOPTS(1);
IF DEBUG
  THEN CALL XLOGA('PERSUP: DEBUG SPECIFIED.');

/ *-- SINGLE OR MULTI USER ? ---------------*/
IF (STHTOPTS(2))
  THEN ACCESS=1; /* MULTI USER */
  ELSE ACCESS=0; /* SINGLE USER */

/ *-- INPUT SAS DATA SET TO UPDATE WITH -------*/
FIELDPTR=ADDR(STMTFLD(1));
IF (STMTFLDTYPE(1) = 0)
  THEN DSPTR=FIELDPTR;
  ELSE DO;
    CALL XLOG2('ERROR: YOU MUST SPECIFY THE DATA SET',
' TO BE PROCESSED.');
    CALL XEXIT(XEXITDATA); END;

Accessing data from SAS data sets

Our PROC is designed to process variables with a given (hard-coded) name. This is a foolish practice in most applications, but we are trying to keep it simple. Is is appropriate for a custom PROC that is built for a single purpose such as ours — to update the employee data base. One of the “rules” in our design was that the SAS variable EMPNO must be present in the data set, and it must be a numeric variable.

XEMPNO=XVFIND(DSPTR,'EMPNO',O);
IF XEMPNO=0 THEN DO;
  CALL XLOG2('ERROR: THE INPUT DATA SET MUST CONTAIN A ',
' VARIABLE NAMED EMPNO.');
  CALL XEXIT(XEXITDATA); END;
Processing the data set

Now that we have the PERSUP2 module defined to do all the SYSTEM 2000 work, we need to process the input data set and apply the updates. Our first task is to establish connection with SYSTEM 2000 and open the employee data base.

```bash
/*-- WE ARE NOW SET UP --
  GO OPEN THE S2K DATA BASE
*/
OPCODE='START' ;
CALL PERSUP2(INFOPTR);
IF DEBUG THEN
  CALL XLOG2('PERSUP; SYSTEM 2000 START, RC=#', XPI12(RC));
IF (RC>O) THEN
  CALL XEXIT(XEXITERROR,D);
OPCODE='OPEN' ;
CALL PERSUP2(INFOPTR);
IF DEBUG THEN
  CALL XLOG3('PERSUP; EMPLOYEE DATA BASE OPEN, RC=#, FEEDBACK=#', XPI12(RC), FEEDBACK);
IF (RC>O) THEN
  CALL XEXIT(XEXITERROR,D);
```

The next observation is acquired from the input data set with a call to XOGET. The SAS supervisor reads the data set and moves the values of the variables we had asked for to the specified address - remember, XVGETD defined an address for the supervisor to return data values.

```bash
/*-- PROCESS EACH OBSERVATION IN THE DATA SET --*/
NOES=D;
OPCODE='UPDATE' ;
DO WHILE (XOGET(DSPTR)=D)
  NOBS=NOBS+1 ;
  CALL RESETEMP;
  CALL XVGET(XVPTR);
  IF $MISS(EMPNOFTL) THEN
    CALL XLOG3('NOTE: OBSERVATION # HAS A MISSING VALUE FOR #', XPI12(NOBS), 'THE EMPNO VARIABLE.');
  ELSE DO;
    EMPNO=EMPNOFTL;
    CALL PERSUP2(INFOPTR);
    IF DEBUG THEN
      CALL XLOG3('PERSUP; # UPDATE, RC=#', XPI12(NOBS), XPI12(RC));
    IF RC=-2 THEN CALL XLOG3('NOTE: EMPLOYEE IS NOT IN DATA BASE.');
    END;
  END;
END;
```

At this point we are ready to communicate with the SYSTEM 2000 data base subroutines (PLEX).

PLEX calls

In the interests of modular programming, I have placed all the SYSTEM 2000 calls into another module, and call this module whenever I require some SYSTEM 2000 function. The module is listed in appendix 1. The arguments and returned values are passed back and forth using a structure, who address is the only argument to PERSUP2.
Termination

When we are finished, we need to close the data base, and shut down communication with SYSTEM 2000. The SAS supervisor automatically closes all open SAS data sets at the end of each step.

OCODE='CLOSE';
CALL PERSUP2(INFOPTR);
IF DEBUG THEN
CALL HLOG2
'PERSUP: DATA BASE CLOSE, RC=',XPI12(RC));

OCODE='STOP';
CALL PERSUP2(INFOPTR);
IF DEBUG THEN
CALL XLOG2('PERSUP: SYSTEM 2000 STOPPED. RC=',XPI12(RC));
XEX:
CALL XEXIST(O,0);

Testing our procedure

The DEBUG flag in our procedure options allows us to see the values of certain variables as the procedure executes. Ordinarily, the end-user would not specify this option—indeed, they would probably not even document it! Here is an example of our procedure.

First, we set up some trial data. The SAS log and procedure output are listed in Appendix 2.

Enhancements

Our procedure performs a useful task of updating a data base and would be a useful model for building more complex ones. It is probably useful to output an SAS data set with the "error" observations, so that these can be checked and re-processed. Real applications would probably involve more than one data base and/or data set, and so our code would need generalizing.

I hope this example has pointed out the benefits/pitfalls of writing a custom SAS procedure, and will inspire many "PROC MYPROC" that does things exactly as you would expect!

Trademark Information

SAS and SYSTEM 2000 are registered trademarks of SAS Institute Inc., Cary, NC 27511, USA.
/*--- SIX UPDATE PROCESSING HERE -------------------------------*/
/*--- SIX TERMINATION HERE -------------------------------*/
/*--- SIX TERMINATION HERE -------------------------------*/
/*--- SIX TERMINATION HERE -------------------------------*/
/*--- SIX TERMINATION HERE -------------------------------*/
/*--- SIX TERMINATION HERE -------------------------------*/
/*--- SIX TERMINATION HERE -------------------------------*/