User-Written Procedures and Extensions in Version 6 of the \textsc{sas} System
Richard D. Langston, \textsc{sas} Institute Inc., Cary, NC

\textbf{ABSTRACT}

One of the more attractive aspects of the \textsc{sas} System is its extensibility. If no available item can perform a desired function, then a new item can be written to fulfill this function. The capability of system extension has been available almost since the \textsc{sas} System's inception. This paper describes the features of the product in Version 6 of the \textsc{sas} System for system extensions, and offers an introduction to writing \textsc{sas} procedures for the Version 6 environment. It contrasts the architecture of such procedures with those from the 8.2.4 and Version 5 \textsc{sas} environments. Also, it describes system extensions, such as user-written formats, informats, functions, and call routines. A final topic of discussion is a compatibility feature of base \textsc{sas} software that allows for certain user-written entities from previous versions to function without change under Version 6.

\textbf{INTRODUCTION}

The product is a toolkit that allows programmers to develop extensions to the \textsc{sas} System. These extensions include \textsc{sas} procedures, informats, formats, functions, and call routines (these three items are referred to as \textsc{iffs}). These extensions, if written in the \textsc{c} language according to the documented specifications, can run on any host on which the \textsc{sas} System runs.

\textbf{TERM DEFINITIONS}

It is best to properly define exactly what is meant by a \textsc{sas} procedure, \textsc{iff}, and so on, even though most readers will have experience with these concepts.

A \textsc{sas} procedure is an executable module that consists of one or more object modules linked together in a special process. The object modules are the result of compilation of source code that is written based on the documented specifications. The purposes of \textsc{sas} procedures are to (1) access \textsc{sas} data sets, (2) write \textsc{sas} data sets, (3) send printed output to the \textsc{sas} log and/or output files, and (4) perform other functions within the \textsc{sas} environment. All \textsc{sas} procedures use a standard syntax as in

\begin{verbatim}
PROC ABC X Y; 

\end{verbatim}

where ABC is the procedure name, X is an option with no value, and Y is an option that expects a value. There are facilities linked into the procedure that describe the syntax of a given procedure (what its options are called, what values are expected, and so forth) and process all the syntax. These facilities are described later in this paper.

An informat is an entity that is invoked by components of the \textsc{sas} System (usually the DATA step's \texttt{INPUT} function or statement, or via \texttt{FSEDIT}) that convert strings into numeric values or other character values. Informats that are table-driven can be defined by \textsc{proc format}. For example:

\begin{verbatim}
PROC FORMAT; INFORMAT XXX 'A=1 B=2 C=3';
DATA TEMP; INPUT X: XXX. 20; CARDS;
A B C
run;

\end{verbatim}

The data set TEMP will contain three observations with the values 1, 2, and 3, respectively, for X.

If your application calls for an informat that is algorithmic in nature, and not table-driven as the example above, then a user-written informat is needed. Most of the Institute-supplied informats are algorithmic. For example, the \texttt{DATE} informat that converts \texttt{dmmmyy} (for example, 01JAN89) into a \textsc{sas} date value is algorithmic, since there is no table that matches an individual string with the proper value. Algorithmic informats are written in a lower-level language such as \textsc{c}.

User-written formats are analogous to informats. \textsc{proc format} can create table-driven formats, but your application may call for an algorithmic format. Again, such formats are written in a language such as \textsc{c}.

User-written functions and call routines are those adjuncts to components of the \textsc{sas} System (primarily the DATA step) that are passed arguments and optionally return values. Consider an example of the Institute-supplied function of \texttt{mdy} that provides the \textsc{sas} date value given the month, day, and year:

\begin{verbatim}
DATA _NULL_; INPUT M D Y 20; DATE-MDY(M,D,Y); CARDS;
1 1 89
run;

\end{verbatim}

The numeric values of 1, 1, and 89 are passed as the three arguments to the function. If you have the need for a function that is not provided for by an Institute-supplied function and it is too complex or not possible for the DATA step to handle in its own native language, you can write a function in a language such as \textsc{c}. Call routines are essentially the same as functions, except that they do not return a value and can update the value of any argument passed.

\textbf{USER-WRITTEN PROCEDURES IN VERSION 6}

Of all the extendable features, \textsc{sas} procedures are the most complicated. This is because they need many services. After all, \textsc{iffs} are given everything they need to compute a group of values that are returned in a very controlled environment. \textsc{sas} procedures are given control and are expected to parse statements, read data sets, print information, write data sets, and perform any other necessary function.

The first step in writing a \textsc{sas} procedure is to decide what you want it to do. How many data sets will it need to read in? Do you expect certain variable names and types in this data set? Do you want to create output data sets? What should the printed output look like? What kind of log messages will be involved? Are you wanting to access some other vendor's database? Once you've answered questions like this you can move on to the next step of defining the syntax.

Since all \textsc{sas} procedures use some common syntax rules, you must do the same with your procedure. The rules are defined using a grammar written in a predefined language. Here is a portion of the grammar for the demonstration procedure \textsc{proc count}:

\begin{verbatim}
PROGRAM  = ANONYMOUS EXTERNAL,
AUTHORS  = CURRICULUM
| WRITING
| SYSTEM
COUNT:

\end{verbatim}
This grammar indicates that there are three possible statements for PROC COUNT: the PROC COUNT statement, a VAR statement, and a BY statement. The VAR statement is followed by a variable list, as is normally the case for procedures. The BY statement is handled in a special way to ensure its compatibility among all procedures using it. The PROC COUNT statement can have several options: FORMATTED (abbreviated as F), UNFORMATTED (or U), REWIND (or R), PRINT (or P), NOLOG (or N), or DATA—. The last two options specify SAS data sets, based on the use of the DSFIELD production.

The grammar is compiled by a special program called the grammar processor that is part of the product. The grammar processor generates a function in the language of choice that can then be compiled. This function is linked into the SAS procedure where it can be called to obtain the correct syntax.

After the grammar has been successfully created, you are ready to write the SAS procedure. The language of choice depends on your familiarity and its functionality with the product. The language availability planned are: IBM® MVS and CMS: SAS/C; IBM FORTRAN VS, IBM PL/I Optimizing Compiler, IBM COBOL, and 370 Assembly Language. VAX/VMS: VAX C, VAX FORTRAN, VAX PL/I, VAX MACRO-12. AOS/VS: C, FORTRAN, PL/I, PRIMOS: C, FORTRAN, PL/I, UNIX® systems: C, OS/2®: Latex C. Some of these languages may not be supported with initial releases of the product on certain systems, and other languages may be added as indicated by demand and viability.

You may find that you need to use more than one language. For example, you may have a FORTRAN subroutine that you wish to call, but you want to write most of the procedure in C. That is acceptable, as long as the compiler properly supports interlanguage communication. However, it is advised that all interfaces to the SAS supervisor be made in one language.

After choosing your language, you should begin writing a framework procedure that performs the most basic tasks to ensure that you have the proper design in mind. Get the procedure to read in a SAS data set and perform some printing before you add all the gory details. Add output data set processing after the input data set processing and printing are close to being complete.

The calling sequence for Version 6 SAS procedures involves setting values in a structure and calling the appropriate routine with the address of the structure. This mechanism is preferred over the standard process of passing arguments to a routine. The reasons for this include 1) simplicity of managing the interface and documentation for different languages, 2) allowing for argument initialization before repetitive calls in loops, and 3) reduced execution time because expensive parameter building code is limited. Note that macros can be written in C for those programmers who prefer an argument calling mechanism. For example:

```c
#define CALL_SPC(a,b,c)          
spc_struct.arg1 = a;           
spc_struct.arg2 = b;           
spc_struct.arg3 = c;           
DSC(spc);                      

... CALL_SPC("observation", -1, 0); 
```

Each procedure will have its own associated structure. All structure definitions are contained in the main #include files. Here are some examples of the structure declarations for the routines used in the sample procedure:

```c
struct WRITE_STRUCTURE write_structure; 
struct WRITE_STRUCTURE svwrite_structure, *svwp; 
struct WRITE_STRUCTURE svreach_structure, *svre; 
struct WRITE_STRUCTURE ssparse_structure, *ssp; 
```

Some of the structure declarations include pointers, in case the pointers are to be set to the addresses of the local copy of the structure. The interface routines expect the address of the structure, so it is more efficient to set a pointer variable once and pass its value than to use the & operator repetitively.

Parsing is accomplished in much the same way it is done in Version 5 procedures. The grammar function is called to obtain a pointer to the parsing structure; this pointer is in turn passed to the parsing routines.

```c
struct WRITE_STRUCTURE write_structure; 
struct WRITE_STRUCTURE svwrite_structure, *svwp; 
struct WRITE_STRUCTURE svreach_structure, *svre; 
struct WRITE_STRUCTURE ssparse_structure, *ssp; 
```

Parsing is accomplished in much the same way it is done in Version 5 procedures. The grammar function is called to obtain a pointer to the parsing structure; this pointer is in turn passed to the parsing routines.

```c
struct WRITE_STRUCTURE write_structure; 
struct WRITE_STRUCTURE svwrite_structure, *svwp; 
struct WRITE_STRUCTURE svreach_structure, *svre; 
struct WRITE_STRUCTURE ssparse_structure, *ssp; 
```

Above: The call to the parsing routine.

```c
struct WRITE_STRUCTURE write_structure; 
struct WRITE_STRUCTURE svwrite_structure, *svwp; 
struct WRITE_STRUCTURE svreach_structure, *svre; 
struct WRITE_STRUCTURE ssparse_structure, *ssp; 
```

The above example shows how the interface routines are called. The items in the local structure are filled in, and the address of the structure is passed. The routine names are called with uppercase names; the uppercase name may refer to macros or actual routines, depending on the particular name.

If the parsing is not successful, this code is used to exit the procedure:

```c
if (proc_error) { 
  svexit_structure.arg1 = EXITERROR; 
  ssviil_structure.arg2 = 0; 
  SEXIT(ssviil_structure); 
}
```

Here is a simplified portion of the data-reading code:

```c
/* svwp points to the SVWRITE_STRUCTURE, which is filled in */ 
/* once with the "filid" of the input data set */ 
while (SVWRITE(filid) > 0) { 
  if (print) SVMALL(); 
  /*---print observations---*/ 
  for (obs = 0; SVWRITE(0);) { 
    *obs: 
    SVGET(log); /* read data */ 
    /*---print the total number of observations---*/ 
    splog_structure.arg1 = "real observations in %", 
    splog_structure.arg2 = nods; 
    splog.argument = deleteby; 
    STLOG(splog_structure); 
  }
```

```c
/*---print the total number of observations---*/ 
splog_structure.arg1 = "real observations in %", 
splog_structure.arg2 = nods; 
splog.argument = deleteby; 
STLOG(splog_structure); 
```
The SBYNEXT routine will cause repositioning to the beginning of a by-group. The SPAGE routine will skip to a new page. The SBYGET routine will read observations within a by-group. The SPSLOG routine will print data on the log.

The FORTRAN interface will include a preprocessor to allow a #include-type mechanism similar to C, to simplify the inclusion of COMMON block definitions.

It is beyond the scope of this paper to explain the linking of the procedure into an executable image. It is sufficient to state that the product provides all the necessary include files to compile the procedure, and all the necessary object libraries to successfully link the module. Linking is done using the standard system tools, such as IEWL on MVS or the LINK command on VAX/VMS.

USER-Written InFORMATS, FORMATs, AND FUNCTIONS

Now let us consider user-written formats. It is simplest to show a complete example and then discuss it. This example is a version of the $EBCDIC format that converts input ASCII data to EBCDIC.

```c
#include "as" 
#include "fmtsub.h"

int s370dlc intc l....PA;RK (i ptr, int, int, int, ph »);

Ft"R'UNES] [ "EBCDIC"
FMTDEC["EBCDIC",1.0,240,0,1,0,0]
FNTDEC[4]

FMTFUNS[1] 
FMTFUN("s370eobc)
FMTFUN(""

FMTFUN:

```c
int s370ebc(intcrom,luw,w,d,to)
ptr from; 
/* I - ptr to data to format */
int iw; 
/* I - width of input data */
int d; 
/* I - no. of decimals always 0*/
int to; 
/* I - ptr to output data */

/* RETURN: always 0 */
*/
*/

#define translate table for ASCII - EBCDIC-----
#include "as3eb.m"
int i,j;

/*translate each input byte, up to w bytes, rest EBCDIC blank-----
 j = ASCII(intcrom,luw); 
 for ([i;=j] i=j+1 [ ]
 to[i] = (int) 7 as3eb(intcrom[i]);.Dec0;
 ]

return(0); 
```

The fmtsub.h file provides the necessary macros to define the format. The format definition is provided by the FMTxxx functions at the beginning of the source. All character informats are passed five arguments, as specified earlier. It is the responsibility of the format to fill the target area appropriately and return the proper return code.

All formats, informats, and functions have a standard calling sequence. With numeric formats, the arguments passed are: the pointer to the floating point number (double), the desired output width, the number of decimal places, and the pointer to the output field. Character formats have an additional argument to indicate the input width. Numeric formats have a pointer and length for input data and a pointer to the output double. Character formats have a pointer and length for input data and a pointer and length for the output field.

Once you're past the calling sequence, you're on your own. You will provide the appropriate return codes to indicate whether your format or format was successful. There are no interface routines to call back to the supervisor, since all work is performed within the routine. You may want to call some other external routines (such as database access), but those accesses are still external to the SAS supervisor.

The user-written functions and call routines use a similar philosophy, in that all work is performed within the routine without reaccessing the supervisor. These routines must handle the input arguments and determine validity.

The IFFs can be packaged into a single module, using a special naming scheme. All user-written informats use the prefix of UWU for executable names. User-written informats use UWU and functions use UWU. The remaining five letters indicate the name(s) of the routines therein. For example, UWUABC indicates that this is the informat ABC. Multiple informats can be placed in an executable if each informat is given a name whose first five characters match all other informats in the module. For example, UWUABCDE would be the name of a single module that had the informats ABCDE, ABCDEY, and ABCDZ contained within it. This same scheme can be used for formats and functions, but there can be no mix between informats, formats, and functions in the same module.

VERSION 82.4 AND VERSION 5 AND COMPATIBILITY

The capability of writing SAS procedures was available to users of the 82.4 and Version 5 SAS Systems. Only PL/I and FORTRAN were supported in 82.4, and only PL/I for Version 5. 82.4 procedures run under Version 5, but only on IBM systems. The reason that 82.4 procedures function under Version 5 is that the Version 5 procedure interface was built atop the 82.4 interface; those procedures using the old-style interface simply fall through to the old-style processing that was still resident.

Architecturally, the 82.4 and Version 5 SAS procedures are completely different. 82.4 procedures use a parsing module to handle syntaxing. This module is written in an assembler macro language that makes extensive references to the SAS supervisor control blocks. The programmer can also add custom code in assembler language that does not use the macros. The supervisor interface is cumbersome and relies heavily on buffer management that must be carefully handled by the programmer. The concept of current data set is used to indicate which SAS data set is in force. Dynamic processing of SAS data sets and more complex parsing is very difficult. Plus, portability is impossible because of the heavy reliance on assembly language and tricks that are inherent to the IBM version of the PL/I compiler.

The Version 5 procedure is architecturally very similar to Version 6. It is grammar-driven and all buffer management is handled by the supervisor. All data set references are performed using the filed pointer. SAS data sets can be opened and closed at will, and the syntaxing process is simplified by the introduction of the grammar. Moreover, the grammar function generated is portable, so that the procedure can be compiled and used on any operating system on which Version 5 is available.
However, the SAS supervisors that are necessary for the 82.4 and Version 5 procedures to run are simply not present in any form under Version 6. Version 5 was not a complete rewrite of the 82.4 system; it was instead an additional implementation atop the existing system. Version 6 is a complete rewrite, almost entirely in C. To support 82.4 or Version 5 procedures, an emulation of those systems would have to be introduced. This implementation would be unreliable at best; it is for this reason that the Institute does not plan to support 82.4 or Version 5 procedures in the Version 6 SAS environment. Both types of procedures will continue to function in the Version 5 SAS environment, which will be supported as a parallel track to the Version SAS 6 system.

IBM user-written IFFs are another story. These modules are much less complicated and much less extensive in scope. Therefore, there is a special interface in base SAS software that allows these entities to continue to function in executable module form, without recompiling or rethinking. This interface is called the Host Format Engine. The engine will be called by the SAS supervisor to attempt to load an IFF that is believed to be an old-style IFF. If the engine determines that it is, a special stub routine is built to convert the standard calling sequence (given in the Version 6 IFF description above) to the old-style calling sequence.

Special consideration is also taken to deal with memory that is allocated above the line in XA systems. The old-style calling sequence for functions with character arguments required the first byte of a pointer to contain the length of the character value. This meant that 24-bit pointers were the limit. If you run the SAS System on MVS/XA, almost all dynamically allocated memory is obtained above the line, that is, requiring 31-bit addressing. The stub routines will be allocated below the line, and all incoming data will be copied below the line to ensure the compatibility with the old style 24-bit addressing. In fact, the stubs switch to AMODE 24 before calling the old-style routines and switch back again upon return.

This processing is totally transparent to the SAS user, since there is no message printed indicating old-style IFFs. You may find that you want to convert old-style informat and formats to Version 6 style. If these are modules that were built by the FORMAT procedure, then you can use the VSTOV6 procedure to convert them. If they are modules created as true user-written informats or formats (that is, written in assembly language), they will have to be converted using the facilities of the product. This is always the case with functions and call routines, since they are always programmer-generated.

**CONCLUSION**

SAS Institute is dedicated to providing users with the necessary tools to accomplish their data processing tasks. The Institute recognizes the need for extendability to the system to handle custom tasks, and feels that the user-written procedure and IFF product help to fill this need.

SAS and SAS/C are registered trademarks of SAS Institute Inc., Cary, NC, USA.

IBM is a registered trademark and OS/2 is a trademark of International Business Machines Corporation.

UNIX is a registered trademark of AT&T.

VAX and VMS are trademarks of Digital Equipment Corporation.