New Features in SAS/TOOLKIT™ Software

Richard D. Langston, SAS Institute Inc.

ABSTRACT

Release 6.07 of the SAS® System under MVS, CMS, and VMSTM operating systems is the initial production release of SAS/TOOLKIT™ software for those same operating systems. This paper discusses some of the features that have been introduced with this production release since the experimental release in July 1990. The major emphasis of the paper is on the new engine writing feature.

INTRODUCTION

The five main areas of new capability are: the Graphics Kernel System (GKS) interface, the SAS/IML® software function interface, the special DATA step variable interface, the TLKTDBG facility, and the engine writing capability. This paper discusses each of these new features.

Before discussing any of these new features, it is prudent to briefly introduce SAS/TOOLKIT software to readers who may not be familiar with this new product but may find that it will meet their needs. The purpose of the product is to allow the user to develop procedures, functions, formats, or engines, and make them available to the SAS systems as extensions to the existing SAS System. Possible applications include database interfaces not provided by SAS/ACCESS® Software, statistical analysis procedures tailored specifically to the user’s needs, and customized table lookup that goes beyond the capability of the FORMAT procedure. The user writes an application in any of several available languages: C, PL/I, FORTRAN, or IBM® assembly language. SAS/TOOLKIT Software provides the necessary tools to successfully compile and link applications so that the SAS System will recognize the resulting modules as appropriate for use by the SAS supervisor software.

GKS INTERFACE

The GKS interface consists of a large number of new routines that can be called by the procedure writer. These routines correspond to the specifications of the GKS interface. The internal design of the graphics capability of the SAS System is very different from the GKS specifications, thus the GKS interface is a layer of software that allows the programmer familiar with GKS to write applications that can take full advantage of the graphics capability of the SAS System.

With the GKS interface, you can write your own graphics procedures, or DATA step graphics functions. SAS/TOOLKIT Software includes several graphics procedure examples that you can use as models to build your own application.

SAS/IML SOFTWARE INTERFACE

The capability of writing standard SAS Software functions has been a part of SAS/TOOLKIT software since its first experimental release. Such functions can be invoked from within a DATA step, an SCL step, or a SAS/IML Software step. However, these functions are limited to scalar arguments, which the user may find too restrictive. If the user wishes to take advantage of vector-processing routines or perform analysis on arbitrarily large matrices, he or she may wish to consider the SAS/IML function interface. A SAS/IML function consists of additional interface routine calls to communicate with the SAS/IML supervisor software. For a detailed discussion on an application that interfaces with IBM’s ESSL software, reference the paper entitled “Writing SAS/IML Functions to Call Subroutine Library Functions,” in the Proceedings of the Sixteenth Annual SAS Users Group International Conference, 1991, pp. 94-99.

Note that SAS/IML functions can be invoked only from within a DATA step or by a SCL step. These functions cannot be invoked by the DATA step or by a SCL step.

SPECIAL DATA STEP VARIABLE INTERFACE

As already mentioned, the user can write SAS functions that can be invoked in a variety of locations within the SAS System. However, these functions operate specifically on the arguments passed. Some ap-
Applications need to deal with a specific set of variables by name or with the entire set of variables. Examples of such applications are those that extract data from a database and insert the data values directly in place for use by the DATA step. The special DATA step variable interface allows the user to write such applications.

The user gains access to the DATA step symbol table elements for a certain group of variables. Symbol table elements can be extracted by variable name or number. The symbol table element contains all the information necessary for the application to process data associated with the variable. For example, the address of the data value in the program data vector is available in the element.

The application can assemble all information about data location and processing with a single call to a DATA step function responsible for initializing the information. Then, the DATA step need only call an "execute" routine for every observation to fill in the program data vector locations. Here is an example:

```
DATA _NULL_; 
IF _N_=1 THEN CALL FUNCX1; 
CALL FUNCXN; 
ABC=X; DEF=Y; 
```

The FUNCX1 function will obtain information on the location of the X and Y variables. When FUNCXN is called, those locations will be filled in with the desired values. From that point on, the variables X and Y will have the desired values and can be used in subsequent DATA step statements, such as in the assignments to ABC and DEF, respectively.

**THE TLKTDBG FACILITY**

When developing SAS/TOOLKIT applications, the user may find results that are not as expected. Abends can occur, or the output may not seem right. The user can consider using an interactive debugger, such as the SAS/C® debugger on IBM systems, but this requires additional knowledge of debuggers, and access to an interactive environment.

The TLKTDBG facility allows the user to dynamically increase the amount of information concerning the invocation of interface routines. This facility can be invoked without having to recompile or relink the application and can be used regardless of implementation language.

To use the TLKTDBG facility, the SAS user enters the following OPTIONS statement before invoking the application:

```
OPTIONS DEBUG='TLKTDBG=n';
```

where n is the value 1, 2, or 3, whose meanings are explained below. Then, the application is invoked. Once the application calls any interface routine starting with a SAS_. name, one or more lines will be printed to the log providing information about the call.

The various values of TLKTDBG= indicate how much information will be printed. For TLKTDBG=1, only the announcement of the routine name is printed when the routine is entered and exited. For TLKTDBG=2, the routine announcements are made, but the values of the arguments are also printed. For TLKTDBG=3, further information about the arguments is printed.

The user can determine if incorrect values have been inadvertently passed to the SAS_. routines. Since the SAS_ routines are optimized for best performance, extensive error checking often is omitted. The TLKTDBG facility allows the user to monitor the arguments to ensure that there are no logic errors in the application that would pass incorrect arguments.

Also, if the application is abending in an unknown location, the user can get an idea of that location by determining the last SAS_. routine invoked. The abend may be occurring during execution of that routine, or it may occur after its return but before the next SAS_. routine. However, this information is more readily available than that provided by the insertion of additional log messages in many positions in the code. The TLKTDBG facility can help the user narrow down the problem areas so that there need not be an overabundance of debug messages added.

The TLKTDBG facility is indispensable for the SAS Institute Technical Support Division. Many problems can be answered without having to supply source code or dumps once the TLKTDBG output is discussed. The user should always run the TLKTDBG=3 option before calling the Technical Support staff with SAS/TOOLKIT questions regarding aberrant behavior.

**ENGINE WRITING**

A long-awaited feature of the SAS System has been introduced with the engine-writing capability of SAS/TOOLKIT Software.
A user-written engine is a module that provides descriptors and data to the SAS Supervisor so that the supervisor treats the information as a SAS data set regardless of the manner in which the data are actually stored. With this design, the engine writer can provide data to the SAS System from a database management system, from a real-time sampler, from an in-house flat file layout, or from any other mechanism in which rectangular data can be provided.

Already within the SAS Supervisor is software known as the "engine supervisor." This supervisor is responsible for managing the LIBNAME statement and for calling engine routines when certain I/O processes are requested. Because writing a full-blown engine is an arduous task, the design of the SAS/TOOLKIT Software engine is based on the fact that a new level of software is introduced. This software, dubbed the "engine middle-manager," acts as a go-between to allow simplified engine writing. This way, the SAS/TOOLKIT software engine writer has to develop only the most basic routines in order to get an engine working.

Your engine can be written in the C language on any host that supports SAS/TOOLKIT Software. It can be written using the PL/I language on the MVS, CMS, and VMS operating systems. It can be written in IBM assembly language on the MVS and CMS systems.

An engine can be an input or output engine or both. The engine writer will provide specified routines based on the functionality of the engine. For any functionality, the writer will provide the ENGOPN routine, which is called by the middle manager when a request is made to open a data set. What the ENGOPN routine actually does is up to the developer. There may be an actual I/O open (such as an fopen call in C or a PL/I OPEN statement). There may be a call to a database open routine. Whatever is appropriate to initiate the I/O processing against the file is performed within the ENGOPN routine. There is also a corresponding ENGCLS routine to perform a matching close action.

Your engine will first be invoked via the LIBNAME statement. When the user enters

LIBNAME xxx youreng 'data-file-here';

your engine (whose executable module is named youreng) is invoked. Your initial code will indicate the features of the engine you are providing, such as input or output capability.

If you are writing an input engine, you will need to provide the ENGNAM and ENGRED routines. The ENGNAM routine is called by the middle manager to provide the variable descriptors to the middle manager. These descriptors contain the pertinent information concerning the variables that will be surfaced to the SAS System. The descriptors contain the variable name, label, length, and type; format name, format length, and number of decimals; and buffer position. The buffer position indicates the relative offset in a buffer that will be maintained by your engine and used by the engine middle manager to provide the data to the engine supervisor.

The ENGRED routine is called by the engine middle manager to read an observation of data from your engine. ENGRED is required to fill in the aforementioned buffer with the appropriate data. Note that since the SAS System deals with floating-point numerics and standard character data, it will be the responsibility of your engine to properly fill in the buffer fields with the data converted to floating-point representation or to character representation, whichever is appropriate.

If your engine supports writing, you will need to provide the ENGDFV and ENGWRT routines. The ENGDFV corresponds to the ENGNAM routine, because variable descriptors are provided. The difference is that the user provides descriptors to ENGDFV, and this routine is responsible for conveying that information to your file system as appropriate. The descriptor structures provided use the same format as those for ENGNAM.

The ENGWRT routine is responsible for writing the observations out to your file system. This is done by providing ENGWRT with a buffer that is filled in with the proper values. The offset information conveyed in the descriptor structures when ENGDFV is called indicates where the data are located in the buffer. It is the responsibility of ENGWRT to convert those fields into the proper representation for your file system and to write out the records.

Completing the basic set of routines for all input engines is the ENGNOT and ENGPNT routines. ENGNOT is provided to allow the engine to reposition within a data set. This repositioning is essential to the proper operation of the SAS System, because BY-group processing takes advantage of the feature. The ENGNOT routine is given the address of a "record ID," or "rid," that is filled in with positioning information. This information is engine-specific and will only be used by the engine. The application will store the rid for a given observation and pass this rid to the ENGPNT routine at a later point. The ENGPNT routine is then responsible for interpreting the rid and repositioning to the observation referenced by that rid.

Your engine may need some special initialization when
it is first referenced in the LIBNAME statement. If
t hat is the case, you can provide the ENGASG routine,
which will be called after the LIBNAME statement is
parsed. Likewise, there is an ENGDAS routine that is
called when the LIBNAME CLEAR statement is seen
(or at the end of the SAS job if no CLEAR statement
is provided).

Another feature that extends the capability of your en­
gine is the n-to-rid and rid-to-n function. Recall that a
"rid" is a record ID that identifies an observation. The
rid-to-n function is capable of determining the observa­
tion number by examining the rid. This is usually
quite simple because the engine can store the observa­
tion number in the rid and extract it as requested.
The ENGR2N routine can be provided by the engine
writer to convert rids to observation numbers. This
feature is used by applications that display the obser­
vation number to users. An example of its use is in the
observation number printing performed by the PRINT
procedure.

The n-to-rid function allows the engine to convert an
observation number into a rid so that the ENGPNT
routine can be called with the rid. This capability is
used by applications, such as the FSEDIT procedure,
in which the user provides the observation number.
The application, in turn, will call the ENGN2R rou­
tine, providing the observation number. ENGN2R will
convert the observation number into a rid, which will
then be passed to ENGPNT. This mechanism can be
simple if your engine has the capability of easily con­
structing a rid from the observation number. However,
if the observation number is not a direct indicator of
the location of the observation number (for example,
compressed data files where observations are not of
uniform length), the rid may be unobtainable without
building an entire array of rids when the ENGPNT
routine is called. It is up to you as the engine writer
to decide if you support n-to-rid or rid-to-n. You do
not have to provide the ENGN2R or the ENGR2N
routines if you do not support the behavior.

A skeletal sample engine written in C is provided at the
end of this paper. The source code of each ENGxxx
routine has been omitted due to space considerations,
but the calling sequence of each routine is supplied to
give the reader an idea of the coding necessary.

ENGINE VERSUS PROCEDURE

The primary advantage of writing an engine versus a
procedure is that the user can access the data directly
without having to convert the entire data file to a SAS
data set. Also, the syntax for accessing data through
an engine is the same as that for accessing data from
SAS data sets, allowing the user to operate with syn­
tax with which they are familiar. If the data files to
be accessed are large, the user does not need the ad­
tional disk space to convert the file before using it.

CONCLUSION

SAS/TOOLKIT Software can meet a variety of needs
for users who wish to access external systems. User­
written procedures and functions can access data and
subroutine libraries at your site. These extensions to
the SAS System use the same syntax and structure
as other components of the SAS System to allow for
the best user understanding. The new feature of user­
written engines allows the user to access external data
in a seamless fashion without having to convert to a
SAS data set.

SAS, SAS/ACCESS, SAS/C, SAS/IML, and
SAS/TOOLKIT are registered trademarks or trade­
marks of SAS Institute Inc. in the USA and other
countries. ® indicates USA registration.

IBM is a registered trademark of International Busi­
ness Machines Corporation.

Other brand and product names are registered trade­
marks or trademarks of their respective companies.
/*-----include files needed-------------------------------------------*/
#include <stdio.h>
#include "uwproc.h"
#include "engdef.h"

/*-----appropriate RID (record id) structure for this engine----------*/
struct RID {
  char ridchar; /* standard rid identifier */
  long loc;   /* value returned from ftell */
};

/* Any engine can have an adjunctive structure to contain anything
   specific to the engine. In this example, we need to keep rids for
   the first record of the logical member (the 'TOF' record), the
   rid for the first observation for the logical member, and the rid
   for the most recently read observation. Also, we'll keep the file
   handle used by fopen et al. outbuf is the buffer to hold output
   values.
   The pointer to this structure is specfid, located in the fid. */

struct SPECFID {
  struct RID headrid; /* TOF rid */
  struct RID engrid1; /* firstobs rid */
  struct RID curr_rid; /* current obs rid */
  FILE *fp; /* file handle */
  char *outbuf; /* output buffer */
  int lastio; /* last I/O operation (read/write) */
  int newobs; /* observations added to end */
};

/*-----main declaration-------------------------------------------*/
int U_ENG( ENGXMPL ) ( argc , argv )
  int argc;
  char * argv[];
{

  struct ENGSTAT engstat;

  /*-----initialize SAS/Toolkit SAS_ routine environment----------------*/
  UWPROC(0);

  /*-----zero out the engentry structure-------------------------------*/
  SAS_ZZEROI((char *)&engstat,sizeof(struct ENGSTAT));

  /*-----indicate the status of different features of the engine-------*/
  engstat.support = 1; /* this engine is supported */
  engstat.read   = 1; /* this engine allows read access */
  engstat.write  = 1; /* this engine allows write access */
  engstat.update = 1; /* this engine allows update access */
  engstat.random = 1; /* this engine allows random access */
  engstat.assign = 0; /* no additional ASSIGN code */
  engstat.note   = 1; /* this engine supports NOTES */
  engstat.ridandn = 0; /* no n-to-rid or rid-to-n support */
  engstat.nopname = 1; /* LIBNAME without phasename allowed */
  engstat.ridlen = sizeof(struct RID); /* rid structure size */
  memcpy(engstat.engname,"ENGXMPL",8); /* engine name */
/*-----load and call the engine middle manager initialization code----*/
UWENGC(argv,engstat);

/*-----return SUCCESS to indicate successful initialization-----------*/
return(SUCCESS);
}

retype ENGPWn(fid, libmode)
fidptr fid; /* ptr to fileid ptr */
int libmode; /* open status */
{
    /* Open the member referred to by the libname.memname in the
    fileid, for the mode (input/output/update) given in the
    fileid. The libmode indicates CONTENTS mode vs. individual
    open. Set rc accordingly. */
    return(rc);
}

retype ENGCLS(fid, disp, libmode)
fidptr fid; /* fileid from engine middle mgr */
int disp; /* disposition (ignored) */
int libmode; /* libmode (explained above) */
{
    /* Close the member referred to by the fileid. Set rc accordingly. */
    return(rc);
}

retype ENGNAM(fid, pxonl)
fidptr fid; /* fileid from engine middle mgr */
xonlptr pxonl; /* ptr to xonlist */
{
    /* Fill in the namestr array referred to in the fileid, and fill
    in the xonlist array, which contains abbreviated variable
    information. */
    return(rc);
}

retype ENGDFV(fid, prec_len)
fidptr fid; /* fileid from engine middle mgr */
long *prec_len; /* output variable information along to the proper
    mechanism. */
{
    return(rc);
}

retype ENGRED(fid, rptr)
fidptr fid; /* fileid from engine middle mgr */
char **rptr; /* returned pointer to input buffer */
{
    /* Read the observation and fill in the values in the proper offsets
    into the buffer referred to by *rptr. */
return(rc);
}

rctype ENGWR(fid,rptr,status)

fidptr fid; /* fileid from engine middle mgr */
char *rptr; /* ptr to data to convert to output fmt */
int status; /* 0 = not update 1 = update */

/* Write the observation out given the values at the proper offsets in the buffer referred to by rptr. */
return(rc);
}

rctype ENGNOT(fid,ridp)

fidptr fid;
char *ridp;

/* Fill in the rid (pointed to by ridp) with the information on the current observation location. */
return(rc);
}

rctype ENGPNT(fid,ridp)

fidptr fid;
char *ridp;

/* Point to the observation referred to by the rid pointed to by ridp. */
return(rc);
}

void ENGTRM() {
/* Perform any necessary termination code. */
}