# SAS® Technical Report P-245 SAS/TOOLKIT® Software: Changes and Enhancements

Releases 6.08 and 6.09

SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513

The correct bibliographic citation for this manual is as follows: SAS Institute Inc., SAS® Technical Report P-245, *SAS/TOOLKIT® Software: Changes and Enhancements, Releases 6.08 and 6.09*, Cary, NC: SAS Institute Inc., 1992. 149 pp.

**SAS® Technical Report P-245, SAS/TOOLKIT® Software: Changes and Enhancements, Releases 6.08 and 6.09**



ISBN 1-55544-523-3



1st printing, October 1992



Doc P19, 092892

# Contents

# Reference Aids

## Tables



## Examples



## Tables



## Figures

# Using This Book

## Purpose

SAS Technical Report P-245, *SAS/TOOLKIT Software:Changes and Enhancements for Releases 6.08 and 6.09* documents the capabilities for writing engines, SAS/IML functions, and special DATA step functions that can update variables not listed as parameters to the function. The report also describes the new debugger facility of SAS/TOOLKIT software and describes how to work with C language debuggers under MVS, CMS, and VMS. Also in this report are appendices for running SAS/TOOLKIT software under OS/2 and UNIX.

This technical report is a Changes and Enhancements document to *SAS/TOOLKIT Software: Usage and Reference, Version 6, First Edition*. This book does not attempt to explain the SAS System or how standard SAS procedures work. For usage information on the SAS System, refer to *SAS Language and Procedures: Usage, Version 6, First Edition* . For reference information, refer to *SAS Language: Reference, Version 6, First Edition* and the *SAS Procedures Guide, Version 6, Third Edition* .

"Using This Book" describes how you can best use this book. It describes the book's intended audience, the audience's prerequisite knowledge, the book's organization and its conventions, and the additional SAS System documentation that is available to you.

## Audience and Prerequisites

Users must be familiar with SAS/TOOLKIT software and the SAS System. This book is intended for programmers who are experienced in programming in the C, PL/I, FORTRAN, or IBM 370 assembler languages.

Using SAS/TOOLKIT software also involves having

- SAS/TOOLKIT software, Release 6.08 or later.
- base SAS software, Release 6.08 or later.
- SAS/IML software, Release 6.08 or later if you are creating and running any SAS/IML functions.
- one of these operating systems: AIX, CMS, HP-UX, MVS, OS/2, SunOS, or VMS.
- the appropriate language compiler for the language you are using. Supported compilers are
  - the SAS/C Compiler under MVS and CMS
  - the IBM Set/2 Compiler under OS/2 2.0
  - the native C compiler under AIX, HP-UX, SunOS, or VMS
  - the VS FORTRAN Version 3 compiler under MVS and CMS
  - the native FORTRAN compiler under AIX, HP-UX, SunOS, or VMS
  - the PL/I Optimizing Compiler under MVS and CMS
  - the native PL/I compiler under VMS
  - the IBM 370 Version 2 H-level Assembler.

# How to Use This Book

This section gives an overview of the book's organization and content.

## Organization

- Chapter 1: "Using the New TLKTDBG Facility"
- Chapter 2: "Using C Language Debuggers with SAS/TOOLKIT Programs"
- Chapter 3: "Debugging Grammars"
- Chapter 4: "Writing Engines"
- Chapter 5: "Writing Special DATA Step Functions"
- Chapter 6: "Writing SAS/IML Functions"
- Chapter 7: "Other New Routines"
- Chapter 8: "PARMCARDS Processing"
- Appendix 1: "Using SAS/TOOLKIT Software under OS/2"
- Appendix 2: "Using SAS/TOOLKIT Software under UNIX"

## What You Should Read

This book describes writing several different types of SAS modules in several different languages. The following table describes users and their needs, and indicates what parts of the book to read.

| If you are . . . | You should read . . . |
|---|---|
| working with the OS/2 operating system | Appendix 1 to learn how to compile and link modules using the `make` facility under OS/2 2.0. |
| working with the UNIX operating system | Appendix 2 to learn how to compile and link modules using the `make` facility under UNIX. |
| writing a procedure | Chapter 1, Chapter 3, and any part of Chapter 2 that is relevant to your compiler. You should also check Chapters 7 and 8 to see if these new features are relevant to your procedure. |
| writing an engine | Chapter 4 to learn how to write engines and whether it is more appropriate to write a procedure or an engine to meet your needs |
| writing a special DATA step function | Chapter 5 for instructions on doing this. |
| writing a SAS/IML function | Chapter 6 for instructions on doing this. |

## Conventions

This section covers the typographical conventions this book uses.

| | |
|---|---|
| roman | is the basic type style used for most text. |
| UPPERCASE ROMAN | is used for references in the text to keywords of the SAS language, filenames, variable names, MVS JCL, CMS EXEC language, PL/I, FORTRAN, and IBM 370 assembler. Variable names from C language examples appear in uppercase in text only when they appear that way in the examples. |
| *italic* | is used to emphasize important information in text. Italic is also used to indicate variable values in examples and syntax. |
| `monospace` | is used to show examples of C or SAS programming code. In most cases, this book uses lowercase type for C programming statements and SAS code. Structure references and any variable names defined with the `#define` command are usually in uppercase monospace.<br>Monospace is also used for C variable names that appear in text. |

## Using the SAS System

This book does not attempt to describe how to use the SAS System in detail. Note that once you have created a procedure or other SAS module, you can run your SAS module using any method of running the SAS System, including the SAS Display Manager System. For more information on running the SAS System, refer to the SAS companion for your operating system.

## Additional Documentation

You may find the following documentation helpful when you are using SAS/TOOLKIT software and the SAS System.

### SAS Documentation

There are many SAS System publications available. To receive a free *Publications Catalog*, write to the following address or call the following telephone number:

SAS Institute Inc.
Book Sales Department
SAS Campus Drive
Cary, NC 27513
919-677-8000

The books listed here should help you find answers to questions you may have about the SAS System in general or specific aspects of the SAS System.

- *SAS/TOOLKIT Software: Usage and Reference, Version 6, First Edition* (order #A56049) provides primary usage and reference information on using SAS/TOOLKIT software. This book includes appendices for the CMS, MVS, and VMS operating systems. It documents the basic steps and required routines for writing a SAS/TOOLKIT procedure.
- *SAS Language and Procedures: Usage, Version 6, First Edition* (order #A56075) is a user's guide to the SAS System. It shows you how to use base SAS software for data analysis, report writing, and data manipulation. It also includes information on methods of running the SAS System and accessing SAS files.
- *SAS Language: Reference, Version 6, First Edition* (order #A56076) provides detailed information on base SAS software, the SAS programming language, and the types of applications the SAS System can perform. Chapter 6, "SAS Files," explains SAS engines and how the implementation of SAS data sets has changed in Version 6 of the SAS System.
- *SAS/IML Software: Usage and Reference, Version 6, First Edition* (order #A56040) provides detailed information about SAS/IML software.
- *SAS Procedures Guide, Version 6, Third Edition* (order #A56080) provides detailed information about the procedures available within base SAS software.
- *SAS Companion for the OS/2 Environment, Version 6, Second Edition* (order #A56111) provides detailed information on running the SAS System under OS/2 2.0.
- *SAS Companion for the UNIX Environment and Derivatives, Version 6, First Edition* (order #A56107) provides detailed information on running the SAS System under UNIX operating systems.
- *SAS Companion for the CMS Environment, Version 6, First Edition* (order #A56103) provides detailed information on running the SAS System under CMS.
- *SAS Companion for the MVS Environment, Version 6, First Edition* (order #A56101) provides detailed information on running the SAS System under MVS.
- *SAS Companion for the VMS Environment, Version 6, First Edition* (order #A56102) provides detailed information on running the SAS System under VMS.

# Summary of Changes and Enhancements

## Introduction

Release 6.08 of SAS/TOOLKIT software provides a number of new features. These changes are summarized here and described in detail in the remainder of this technical report. New graphics capabilities available with Release 6.08 are described in *SAS Technical Report P-246, SAS/TOOLKIT Software: Graphics Capabilities, Release 6.08.*

## Debugging Capabilities

This technical report includes descriptions of using the new SAS/TOOLKIT debugger, TLKTDBG, as well as notes on using the C language debugger under MVS, CMS, and VMS. The report also provides instructions on debugging grammars.

## Data Base Engines

With Release 6.08, you can now write your own data base engines. This report contains complete information on how to write data base engines with SAS/TOOLKIT software.

## Special DATA Step Functions

SAS/TOOLKIT software now enables you to write DATA step functions that do not require you to pass all variables to the function to have the variables updated. Instead, these special DATA step functions update variables by operating on the symbol tables that describe the variables used in a DATA step.

## IML Functions

SAS/TOOLKIT software now permits you to write special IML functions that can be called from within the IML environment to operate on vectors and matrices.

## Additional SAS_X and SAS_Z Routines

New SAS_X and SAS_Z routines provide additional capabilities for processing the members in a data library and catalog entry names, and for initializing a floating point missing value to be used with graphics procedures.

## PARMCARDS Processing

This technical report describes how to use the PARMCARDS statement with a user-written procedure that performs its own parsing of input statements (without using a standard SAS/TOOLKIT grammar module).

# Chapter 1 Using the new TLKTDBG facility



## Introduction

The TLKTDBG facility is a new feature in Release 6.08. This facility allows you to obtain debug information concerning SAS_ routines called by your code. You can turn this facility on and off without having to recompile or relink your code. This facility works with any of the supported programming languages.

The facility is activated by the SAS System option DEBUG= when its value is of the form TLKTDBG=x. For example,

```
OPTIONS DEBUG='TLKTDBG=1';
```

The different values of TLKTDBG are:

```
0 no debugging information printed (default)
1 report entry/exit for SAS_ routines
2 report argument values passed to SAS_ routines
3 report dereferenced pointer information
```

For larger values of TLKTDBG, all effects of lower values remain in effect. So, for example, if TLKTDBG=2, the debug report will show entry/exit information and argument values passed.

When a nonzero TLKTDBG value is specified, the SASPROCD module (shipped with the base product in 6.08 and later releases) is loaded, and the appropriate level of debug information is printed. When TLKTDBG is zero, SASPROCI (also shipped with the base product in 6.08 and later releases) is loaded, and no debug information will appear. Note that changing the TLKTDBG value does not require any recompilation or relinking of your code.

The TLKTDBG facility is very useful in helping you find bugs in your code. After all, if you call the SAS_ routines in the incorrect order, or if you pass these routines incorrect arguments, your code may generate incorrect results or cause abends. The TLKTDBG facility allows you to generate quick debugging output without having to write your own debugging statements. In addition, the facility will allow SAS Institute Technical Support to better assist you in locating problems.

Note that TLKTDBG only operates on SAS_ routines. It has no effect on FMTxxx and FNCxxx routines, nor will it be invoked for any of the UWPRCx routines. It will also not be invoked for any engine routines.

The SAS_XPSLOG, SAS_XPSPRN, SAS_XPS, and SAS_XVDFLST calls are not referenced in TLKTDBG output.

# Example

The following example illustrates the output produced from running a user-written procedure, PROC SIMPLE, without using the TLKTDBG facility.

```
1          DATA TEMP; X=1; RUN;

NOTE: The data set WORK.TEMP has 1 observations and 1 variables.
NOTE: The DATA statement used 0.07 CPU seconds and 2703K.

2          * WITH DEFAULT TLKTDBG (0) ;
3          PROC SIMPLE DATA=TEMP OUT=NEW;RUN;
NOTE: This is PROC SIMPLEC for the C Compiler.

NOTE: The data set WORK.NEW has 1 observations and 1 variables.
NOTE: The PROCEDURE SIMPLE used 0.05 CPU seconds and 2877K.
```

The next example show the results of running the same procedure with DEBUG='TLKTDBG=1'.

```
4          * WITH TLKTDBG 1 (ANNOUNCE ENTRY/EXIT ONLY);
5          OPTIONS DEBUG='TLKTDBG=1';
6          PROC SIMPLE DATA=TEMP OUT=NEW;RUN;
NOTE: This is PROC SIMPLEC for the C Compiler.
TLKTDBG1: Calling SAS_XMEMEX  (SAS027)...
TLKTDBG1: --Returning from SAS_XMEMEX  (SAS027)...
TLKTDBG2:    return value is: 05363CA8
TLKTDBG1: Calling SAS_ZMOVEI  (SAS122)...
TLKTDBG1: --Returning from SAS_ZMOVEI  (SAS122)...
TLKTDBG1: Calling SAS_XSPARSE (SAS001)...
TLKTDBG1: --Returning from SAS_XSPARSE (SAS001)...
TLKTDBG2:    return value is:0 (00000000)
TLKTDBG1: Calling SAS_XVGETI  (SAS083)...
TLKTDBG1: --Returning from SAS_XVGETI  (SAS083)...
TLKTDBG2:    return value is:0 (00000000)
TLKTDBG1: Calling SAS_XVPUTI2 (SAS095)...
TLKTDBG1: --Returning from SAS_XVPUTI2 (SAS095)...
TLKTDBG2:    return value is:0 (00000000)
TLKTDBG1: Calling SAS_XVPUTT  (SAS096)...
TLKTDBG1: --Returning from SAS_XVPUTT  (SAS096)...
TLKTDBG1: Calling SAS_XVPUTT  (SAS096)...
TLKTDBG1: --Returning from SAS_XVPUTT  (SAS096)...
NOTE: The data set WORK.NEW has 1 observations and 1 variables.
NOTE: The PROCEDURE SIMPLE used 0.06 CPU seconds and 3054K.
```

The next example show the results of running the same procedure with DEBUG='TLKTDBG=2'.

```
7          * WITH TLKTDBG 2 (ALSO PROVIDE ARGUMENT VALUES);
8          OPTIONS DEBUG='TLKTDBG=2';
9          PROC SIMPLE DATA=TEMP OUT=NEW;RUN;
NOTE: This is PROC SIMPLEC for the C Compiler.
```

```
TLKTDBG1: Calling SAS_XMEMEX  (SAS027)...
TLKTDBG2:    arg1= 176 (000000B0)
TLKTDBG1: --Returning from SAS_XMEMEX  (SAS027)...
TLKTDBG2:    return value is: 05363CA8
TLKTDBG1: Calling SAS_ZMOVEI  (SAS122)...
TLKTDBG2:    arg1= 05438FD0,arg2= 05363CA8,arg3= 176 (000000B0)
TLKTDBG1: --Returning from SAS_ZMOVEI  (SAS122)...
TLKTDBG2:    arg1= 05438FD0,arg2= 05363CA8,arg3= 176 (000000B0)
TLKTDBG1: Calling SAS_XSPARSE (SAS001)...
TLKTDBG2:    arg1= 05363CA8,arg2= 00000000,arg3= 054384D8
TLKTDBG1: --Returning from SAS_XSPARSE (SAS001)...
TLKTDBG2:    return value is:0 (00000000)
TLKTDBG1: Calling SAS_XVGETI  (SAS083)...
TLKTDBG2:    arg1= 053C4800,arg2= 1 (00000001),arg3= 000D50B4
TLKTDBG1: --Returning from SAS_XVGETI  (SAS083)...
TLKTDBG2:    return value is:0 (00000000)
TLKTDBG1: Calling SAS_XVPUTI2 (SAS095)...
TLKTDBG2:    arg1= 05383800,arg2= 1 (00000001),arg3= 000D50B8
TLKTDBG1: --Returning from SAS_XVPUTI2 (SAS095)...
TLKTDBG2:    return value is:0 (00000000)
TLKTDBG1: Calling SAS_XVPUTT  (SAS096)...
TLKTDBG2:    arg1= 05364E68
TLKTDBG1: --Returning from SAS_XVPUTT  (SAS096)...
TLKTDBG2:    arg1= 05364E68
TLKTDBG1: Calling SAS_XVPUTT  (SAS096)...
TLKTDBG2:    arg1= 00000000
TLKTDBG1: --Returning from SAS_XVPUTT  (SAS096)...
TLKTDBG2:    arg1= 00000000
NOTE: The data set WORK.NEW has 1 observations and 1 variables.
NOTE: The PROCEDURE SIMPLE used 0.07 CPU seconds and 3054K.
```

The next example show the results of running the same procedure with DEBUG='TLKTDBG=3'.

```
10         * WITH TLKTDBG 3 (ALSO PROVIDE DEREFERENCING);
11         OPTIONS DEBUG='TLKTDBG=3';
12         PROC SIMPLE DATA=TEMP OUT=NEW;RUN;
NOTE: This is PROC SIMPLEC for the C Compiler.
TLKTDBG1: Calling SAS_XMEMEX  (SAS027)...
TLKTDBG2:    arg1= 176 (000000B0)
TLKTDBG1: --Returning from SAS_XMEMEX  (SAS027)...
TLKTDBG2:    return value is: 05363CA8
TLKTDBG3:    deref'd return ptr->00000000
TLKTDBG1: Calling SAS_ZMOVEI  (SAS122)...
TLKTDBG2:    arg1= 05438FD0,arg2= 05363CA8,arg3= 176 (000000B0)
TLKTDBG1: --Returning from SAS_ZMOVEI  (SAS122)...
TLKTDBG2:    arg1= 05438FD0,arg2= 05363CA8,arg3= 176 (000000B0)
TLKTDBG3:    deref'd outgoing values... arg1->00010029,arg2->00010029
TLKTDBG1: Calling SAS_XSPARSE (SAS001)...
TLKTDBG2:    arg1= 05363CA8,arg2= 00000000,arg3= 054384D8
TLKTDBG1: --Returning from SAS_XSPARSE (SAS001)...
TLKTDBG2:    return value is:0 (00000000)
TLKTDBG3:    deref'd outgoing values... arg1->16161616,arg3->05362FC0
```

```
TLKTDBG1: Calling SAS_XVGETI  (SAS083)...
TLKTDBG2:    arg1= 053C4800,arg2= 1 (00000001),arg3= 000D50B4
TLKTDBG1: --Returning from SAS_XVGETI  (SAS083)...
TLKTDBG2:    return value is:0 (00000000)
TLKTDBG3:    deref'd outgoing values... arg1->05383800,arg3->0542E200
TLKTDBG1: Calling SAS_XVPUTI2 (SAS095)...
TLKTDBG2:    arg1= 05383800,arg2= 1 (00000001),arg3= 000D50B8
TLKTDBG1: --Returning from SAS_XVPUTI2 (SAS095)...
TLKTDBG2:    return value is:0 (00000000)
TLKTDBG3:    deref'd outgoing values... arg1->05270284,arg3->05364E68
TLKTDBG1: Calling SAS_XVPUTT  (SAS096)...
TLKTDBG2:    arg1= 05364E68
TLKTDBG1: --Returning from SAS_XVPUTT  (SAS096)...
TLKTDBG2:    arg1= 05364E68
TLKTDBG3:    deref'd outgoing values... arg1->05364FA8
TLKTDBG1: Calling SAS_XVPUTT  (SAS096)...
TLKTDBG2:    arg1= 00000000
TLKTDBG1: --Returning from SAS_XVPUTT  (SAS096)...
TLKTDBG2:    arg1= 00000000
NOTE: The data set WORK.NEW has 1 observations and 1 variables.
NOTE: The PROCEDURE SIMPLE used 0.07 CPU seconds and 3054K.
```

## Explanation of TLKTDBG Output

Here is a sample of output from the TLKTDBG facility.

```
TLKTDBG1: Calling SAS_XVNAME  (SAS089)...
TLKTDBG2:    arg1= 053C4800,arg2= 1 (00000001),arg3= 000D50D0
TLKTDBG1: --Returning from SAS_XVNAME  (SAS089)...
TLKTDBG2:    return value is:0 (00000000)
TLKTDBG3:    deref'd outgoing values... arg1->05383800,arg3->0542E280
```

The following paragraphs explain this output line by line.

```
TLKTDBG1: Calling SAS_XVNAME  (SAS089)...
```

The TLKTDBGn prefix indicates what TLKTDBG level produced the output. In this case, TLKTDBG level 1 produced the message. Level 1 output consists of routine invocation announcements.

"Calling SAS_XVNAME" indicates that the SAS_XVNAME routine has been called from your code. "(SAS089)" indicates the internal name of the routine. All SAS_ routines are actually referred to internally by a generic, unmnemonic SASnnn name. The SASnnn name appears for reference, in case you need to set some form of breakpoint for the routine.

```
TLKTDBG2:    arg1= 053C4800,arg2= 1 (00000001),arg3= 000D50D0
```

This message is generated by TLKTDBG level 2. There are three arguments expected by SAS_XVNAME, and the three values are listed here. The first argument is 053C4800. This argument happens to be a fileid. In debugging your code, you'd want to check to see that other SAS_ routines using fileids had values that matched 053C4800. If not, this would indicate bugs in your code. The second argument is 1. This argument is the variable number, and is passed as an integer. TLKTDBG level 2 messages include the integer formatted

normally, and also in hex representation. This is helpful if your code has overwritten the number and is passing a value that appears as some unrecognizable integer whose hex representation may be recognizable. The third argument is 000D50D0. This is a pointer to the NAMESTR pointer that will be filled in by SAS_XVNAME. This should be a valid pointer.

```
TLKTDBG1: —-Returning from SAS_XVNAME  (SAS089)...
```

TLKTDBG level 1 produces this message. This message appears when the SAS_XVNAME routine returns. If your output does not include this message, it means that the arguments passed to the routine (or possibly some prior routine) have caused a failure in the SAS_ routine.

```
TLKTDBG2:    return value is:0 (00000000)
```

TLKTDBG level 2 produces this message. This provides the return value from the SAS_XVNAME routine. Integer values are printed in integer representation and in hex format.

```
TLKTDBG3:    deref'd outgoing values... arg1->05383800,arg3->0542E280
```

TLKTDBG level 3 produces this message. All pointer arguments are dereferenced here, with the first four dereferenced bytes printed in hex. This feature is especially useful to ensure that you are passing and obtaining correct pointers. The TLKTDBG level 3 detects if a NULL pointer is provided, and will not dereference such values. In our example, arg1 (which is the fileid) points to data whose first four bytes are 05383800. Since the fileid is an internal structure not known to you, its contents should be irrelevant. arg3 (the NAMESTR pointer's address) is set by SAS_XVNAME, and its contents are of importance to you. In our example, SAS_XVNAME has set the pointer to 0542E280.

Note that TLKTDBG level 3 does not dereference pointers upon entry to the SAS_ routines. This is because, quite often, the SAS_ routines update data pointed to by pointer arguments, so the data upon entry is irrelevant, and could be confusing if printed in the message. Any pointers pointing to data that is not changed will still be pointing to the correct data at return, when TLKTDBG level 3 will dereference.

Note also that if your code calls SAS_ routines with an incorrect number of arguments, this will not be detected by TLKTDBG, except that you may notice invalid arguments if you pass too few to the routine.

# Chapter 2 Using C Language Debuggers with SAS/TOOLKIT Programs



## Introduction

This chapter discusses how to use the C language debugger on your MVS, CMS, or VMS operating system.

## Using the SAS/C Debugger Under MVS and CMS

If you are writing your SAS/TOOLKIT application in SAS/C on MVS or CMS, you can use the SAS/C Debugger and bring up your application under the debugger.

First, be sure that you are using the -d option when compiling your code. If you forget to do this, the debugger will not be able to step through your code. Also, be sure to associate the DDname SYSDBLIB with a member of your debug library when compiling.

After you have compiled and linked your code, you can bring up the SAS System in the normal way. Before invoking the appropriate SAS command, be sure to associate the DDname DBGLIB with your debug library.

Once the SAS System has prompted you for statements, enter the following statement:

```
options parm='=d';
```

The PARM= string is always passed to an IBM SAS/TOOLKIT application, regardless of implementation language. The =d value activates the SAS/C Debugger from the first routine of the SAS/C framework. For procedures, that will be your U_MAIN routine. For IFFCs, that will be your IFFMAI routine. For engines, that will be your U_ENG routine. You will be prompted to enter SAS/C Debugger commands. For more information on using the SAS/C Debugger, consult the SAS/C Source Level Debugger User's Guide.

Here is a section of a TSO session using the SAS/C debugger:

```
alloc f(ctrans) da(local.SASC.library) shr reu
READY
alloc f(saslib) da(your.sas.proc.library) shr reu
READY
alloc f(dbglib) da(your.debug.library) shr reu
```

```
READY
sas608



1?
data temp;x=1;run;
NOTE: The data set WORK.TEMP has 1 observations and 1 variables.
NOTE: The DATA statement used 0.07 CPU seconds and 3017K.

2?
options parm='=d';
3?
proc multiply data=temp out=new;run;
Set system breakpoint at 0FABAC to activate ESCAPE command
SAS/C Source Debugger Release 4.50H
TRACE: (L$CMAIN) -> MAIN(MAIN)
Listing source from dsn: your.sas.proc.source.library(CMULTIPL)
MAIN(MAIN) 32: void    U_MAIN(MULTIPLY)()
break on entry - context of MAIN
CDEBUG:
b prnt_obs e
CDEBUG:
b prnt_sum e
CDEBUG:
g
SAS/TOOLKIT SOFTWARE ** MULTIPLY PROCEDURE **.
This procedure is written in C

TRACE: MAIN(MAIN), line   204 -> PRNT_OBS(MAIN)
PRNT_OBS(MAIN) 338: void    Prnt_obs(obser, old_obser, printopt,varinfo)
break on entry - context of PRNT_OBS
CDEBUG:
g
TRACE: MAIN(MAIN), line   206 -> PRNT_SUM(MAIN)
PRNT_SUM(MAIN) 353: void    Prnt_sum(sum, offset, varinfo)
break on entry - context of PRNT_SUM
CDEBUG:
g



                                The SAS System                      1
                                                          (current date)

OBS     VARIABLE    PREVIOUS VALUE         NEW VALUE             SUM

 1       X                      1                 1           $1.00


NOTE: The data set WORK.NEW has 1 observations and 2 variables.
NOTE: The PROCEDURE MULTIPLY used 0.28 CPU seconds and 3267K.
```

```
4?
endsas;
```

You can also use the SAS/C Debugger with a SAS/C SAS/TOOLKIT application in an MVS SAS batch job. When running in batch, be sure to include 'ON' commands that instruct the debugger on what to do when breakpoints are hit.

Here is the same example from above, but run in an MVS batch environment:

```
//DEBUGSSN  JOB
//         EXEC SAS608
//CTRANS     DD DSN=local.SASC.library,DISP=SHR
//SASLIB     DD DSN=your.sas.proc.library,DISP=SHR
//DBGLOG     DD SYSOUT=A
//DBGSRC     DD DSN=your.sas.proc.source.library,DISP=SHR
//DBGLIB     DD DSN=your.debug.library,DISP=SHR
//DBGIN      DD *
b     Prnt_obs e
on prnt_obs e {g}
b     Prnt_sum e
on prnt_sum e {g}
g
EXIT
//SYSIN      DD *
DATA TEMP; X=1; RUN;
OPTIONS PARM='=D';
PROC MULTIPLY DATA=TEMP OUT=NEW; RUN;
//
```

Here is the output from the DBGLOG file:

```
Set system breakpoint at 0BEBAC to activate ESCAPE command
SAS/C Source Debugger Release 4.50H
TRACE: (L$CMAIN) -> MAIN(MAIN)
Listing source from ddn:DBGSRC(CMULTIPL)
MAIN(MAIN) 32: void    U_MAIN(MULTIPLY)()
break on entry - context of MAIN
INPUT: b     Prnt_obs e
INPUT: on prnt_obs e {g}
INPUT: b     Prnt_sum e
INPUT: on prnt_sum e {g}
INPUT: g
TRACE: MAIN(MAIN), line   204 -> PRNT_OBS(MAIN)
PRNT_OBS(MAIN) 338: void    Prnt_obs(obser, old_obser, printopt,varinfo)
break on entry - context of PRNT_OBS
TRACE: MAIN(MAIN), line   206 -> PRNT_SUM(MAIN)
PRNT_SUM(MAIN) 353: void    Prnt_sum(sum, offset, varinfo)
break on entry - context of PRNT_SUM
```

Note that you cannot "step into" code which has not been compiled with the -d option of the SAS/C compiler. The debugger can alert you upon entering such routines, but will report that the module was not compiled with the DEBUG options. This restriction also applies to all Institute-supplied code, such as PRCINT and IFFINT, since these modules are

not shipped in debug format.

Also note that when you use the debugger in interactive line mode, you will see an interlacing of debugger output and SAS log and list output. When running in batch, these files are all separate. This may be an important factor in determining the location of a problem, since the interlacing will help pinpoint the time in which a problem occurred.

## Using the C Language Debugger under VMS

You should compile your code with the /DEBUG option, and most likely with the /NOOPTIMIZE option. Optimized code may cause certain variables to be "optimized away" and not be displayable under the debugger. You should also link your executable with the DEBUG option.

When you invoke the SAS command, you will need to add an extra option:

```
$ SAS/$INTERNAL="/$BR=module"
```

where *module* is the name of your module. It is important to ensure that the syntax matches that given above. When your module is loaded, the debugger will be invoked. You will need to SET IMAGE, SET MODULE, and SET LANG (if other than C) in order to set the proper breakpoints.

### Sample Session

```
$ sas/nodms/$internal="/$br=multiply"

NOTE: Copyright(c) 1989 by SAS Institute Inc., Cary, NC USA.
NOTE: SAS (r) Software Release 6.08

1? data temp; x=1;run;
NOTE: The data set WORK.TEMP has 1 observations and 1 variables.

2? proc multiply data=temp; run;

         VAX DEBUG Version V5.4-019
%DEBUG-I-INITIAL, language is C, module set to VILOADV

image name                      set    base address    end address

*MULTIPLY                        yes    0023D000        00242DFF

total images: 1                 bytes allocated: 355032
DBG> set image multiply
DBG> set module CMULTIPL
DBG> set break Prnt_obs
DBG> set break Prnt_sum
DBG> g

SAS/TOOLKIT SOFTWARE ** MULTIPLY PROCEDURE **.
This procedure is written in C

break at routine CMULTIPL\Prnt_obs
```

```
  1987: double  obser,

DBG> g

break at routine CMULTIPL\Prnt_sum
  2002: double  sum;

DBG> g

                          The SAS System                    1
                                                  (current date)

OBS     VARIABLE    PREVIOUS VALUE         NEW VALUE        SUM

 1       X                      1                 1      $1.00


NOTE: The data set WORK.DATA1 has 1 observations and 2 variables.

3? endsas;

NOTE: SAS Institute Inc., SAS Campus Drive, Cary, NC USA 27513-2414

%DEBUG-I-EXITSTATUS, is '%SAS-S-NORMAL, normal successful completion'
DBG> exit
```

Note that the debugger is case-sensitive about module and breakpoint names.

# Using the dbx Debugger under AIX

You can use the dbx debugger under AIX to debug your SAS/TOOLKIT applications.

To access symbolic information for SAS/TOOLKIT executables, you must compile your source with the -g option. Otherwise, you will not have access to the source when debugging.

The dbx debugger allows you to access symbols in executables that have already been loaded. Since the SAS System consists of many executables that are dynamically loaded as needed (via the `load` system call), you must break in `load` and ensure that it is loading the executable you're interested in.

This discussion does not attempt to explain the dbx commands. Consult your AIX and/or UNIX documentation for more information on dbx.

## Using dbx with the SAS System

To invoke the SAS System with dbx, first determine the actual command that is being invoked, not necessarily your script. Once you determine the command, enter

```
dbx -r /usr/lib/sas/sas -nodms
```

When dbx comes up, you receive messages that look like this:

```
Entering debugger ...
```

```
dbx version 3.1 for AIX.
Type 'help' for help.
reading symbolic information ...warning: no source compiled with -g

stopped due to load or unload in .load.load at 0xd000e638
0xd000e638 (load+0x40) 80410014          l   r2,0x14(r1)
```

The message about "no source compiled with -g" refers to the first SAS System host supervisory code which was indeed not compiled with the -g option. This is not a problem. The message about "stopped due to load . . . " is also normal and not a problem.

You do not need to enter any other dbx commands at this point, so enter the `cont` command to allow the SAS System to begin its initialization:

```
cont
```

You receive the 1? prompt. Enter whatever SAS statements are necessary, but do not begin the step that invokes your application. You must invoke a dbx breakpoint before that point.

When you reach the point where you want to invoke your application, enter CNTL-C or whatever your current interrupt signal is (use `stty -a` to query signals), and you'll again be prompted for dbx commands. Enter

```
stop in load
cont
```

This sets a breakpoint in the `load` routine, which is called to load executables. The `cont` command allows the SAS System to proceed. You then see a message from the SAS System:

```
Press Y to cancel submitted statements, N to continue.
```

This message is a response to the interrupt you issued previously. Respond with 'n'.

```
n
```

You continue to receive more ? prompts to enter SAS statements. Enter the statements that invoke your application. You encounter the load breakpoint from dbx, and the message looks something like this:

```
[1] stopped in .load.load at 0xd000e5f8
0xd000e5f8 (load)    7c0802a6       mflr   r0
(dbx)
```

To verify if load is about to load your executable, first enter the `x` command:

```
x
```

This command prints the value of all the registers. The output looks something like this:

```
$r0: 0xd000e5f8  $stkp: 0x2010b648   $toc: 0x2001bdf4    $r3: 0x2008f46e
$r4: 0x00000000    $r5: 0x00000000    $r6: 0x20052c20    $r7: 0x00b88830
$r8: 0x00e2ac00    $r9: 0x00000000   $r10: 0x00000000   $r11: 0x00000000
$r12: 0x2001ab18   $r13: 0xdeadbeef   $r14: 0xdeadbeef   $r15: 0xdeadbeef
```

```
 $r16: 0xdeadbeef   $r17: 0xdeadbeef   $r18: 0x3000f074   $r19: 0x200f96ac
 $r20: 0x2010bd5c   $r21: 0x200fafd8   $r22: 0x2010bbac   $r23: 0x20054f90
 $r24: 0x00000000   $r25: 0x2010b800   $r26: 0x300ce730   $r27: 0x2010b818
 $r28: 0x300f1000   $r29: 0x2008f31e   $r30: 0x2010b6a9   $r31: 0x20055280
 $iar: 0xd000e5f8   $msr: 0x0000d0b0    $cr: 0x28284844  $link: 0x100133b0
 $ctr: 0xd000e5f8   $xer: 0x00000009    $mq: 0x00b88830   $tid: 0x00000000
          Condition status = 0:e 1:l 2:e 3:l 4:g 5:l 6:g 7:g
        [unset $noflregs to view floating point registers]
in .load.load at 0xd000e5f8
0xd000e5f8 (load)    7c0802a6       mflr   r0
```

You are interested in the value of $r3. You want to display the data for approximately 50 bytes at the address referred to by $r3. Given the $r3 value above, the command to examine the 50 bytes after location 0x2008f46e would be

```
0x2008f46e/50c
```

The response from the command is something like this:

```
2008f46e:  '/' 's' 'a' 's' '1' 'h' 'o' 'm' 'e' '/' 's' 'a' 's' 'x' 'y' 'z'
2008f47e:  '/' 't' 'o' 'o' 'l' 'k' 'i' 't' '/' 'c' '/' 'l' 'o' 'a' 'd' '/'
2008f48e:  'o' 'p' 'e' 'n' 't' 's' 't' '\0' '\0' '\0' '\0' '\0' '\0' '\0' '\0' '\0'
2008f49e:  '\0' '\0'
```

The last byte of the module name is the byte before the first \0. In the above case, the module name is /sas1home/sasxyz/toolkit/c/load/opentst, which is the module you're interested in. You can now disable the breakpoint for load by learning its breakpoint number. Enter the command

```
status
```

to get a list of the breakpoints. The command output looks something like this:

```
[1] stop in load
```

The number in brackets is the breakpoint number. Remove this breakpoint with the delete command:

```
delete 1
```

Now run dbx until `load` finishes the load. Use the `return` command to run until it finishes:

```
return
```

In response, dbx produces messages like these:

```
stopped in . at 0x100133b0
0x100133b0 (???) 80410014          l   r2,0x14(r1)
```

Now the module is loaded. You can set whatever breakpoints you want. In this example, the first breakpoint is set for the first executable statement in PROC OPENTST:

```
stop in OPENTST
cont
```

dbx stops at the requested breakpoint and produces messages like these:

```
[2] stopped in OPENTST at line 24 in file "copentst.c"
   24   UWPRCC(&proc);
```

You can now set any other breakpoints and do whatever is appropriate. When you're done, terminate the SAS System normally, and use the dbx `quit` command to terminate dbx:

```
quit
```

## Breakpoints for SAS/TOOLKIT Applications

To stop at the first executable statement in a SAS/TOOLKIT procedure, use the name that appears in the U_MAIN macro in the procedure source. For SAS/TOOLKIT informats/functions/ formats/call routines (IFFCs), use IFFMAI. For SAS/TOOLKIT engines, use ENGMAI. Note that for IFFCs, `load` may be called several times to load different DATA step modules, so you must look at the module name via $r3. For engines, issue an interrupt before the LIBNAME statement that specifies the engine. Set the breakpoint for `load`, then examine $r3 accordingly.

## Avoiding Load Breakpoints

If your application is not aborting, you can simplify the breakpoint approach and omit breaking in `load`. Invoke your application once, issue an interrupt signal, then specify where to break in your SAS/TOOLKIT code, since your executable is already loaded.

For example, consider this test of PROC OPENTST:

```
dbx -r SAS-command
cont

1? data temp; x=1; run;
2? proc opentst; run;

   PROC OPENTST runs to completion

   press CNTL-C at this point

stop in OPENTST
cont

Press Y to cancel submitted statements, N to continue.
n

3? proc opentst; run;

   dbx stops in OPENTST as requested
```

This method does not work if your application aborts, because dbx stops as soon as it

encounters the error, and recovery may not be possible.

## Using with SAS Display Manager

Use can also use dbx if you are running your program from the SAS Display Manager System. Just remember that you must return to the shell in which the dbx command was invoked in order to issue an interrupt such as CNTL-C. Also, all dbx commands must be entered in this shell.

## Using xde

The X-windows front-end to dbx, xde, can also be used to debug SAS/TOOLKIT applications. Use the `xde -r` command instead of the `dbx -r` command to invoke xde. All dbx commands discussed in this document apply to xde as well.

# Chapter 3 Debugging Grammars



## Introduction

This chapter describes how to debug grammars for user-written procedures. The DEBUG= option can be used to assist in debugging grammars. If you use the STMTDUMP and/or TOK keywords with the DEBUG= option, you will get a trace of the parsing process.

## Sample Session

Consider this SAS session. First, we create a data set, then run PROC EXAMPLE with the DEBUG= option of STMTDUMP.

```
1          DATA TEMP; X=1; Y=2;
2          OPTIONS DEBUG='STMTDUMP';

NOTE: The data set WORK.TEMP has 1 observations and 2 variables.
NOTE: The DATA statement used 0.06 CPU seconds and 2368K.

3          PROC EXAMPLE DATA=TEMP SKIP; VAR X; RUN;
SAS/TOOLKIT example procedure written in C.
<followed by other log messages from the procedure>
```

In the print file, we will see the following listing.

```
 .SEM. act 1 PROCINIT args=   tok->spell = EXAMPLE
 .SEM. act 3 STMTINIT args= 20   tok->spell = EXAMPLE
 .SEM. act 30 SPECIAL args=208   tok->spell = TEMP
 .SEM. act 30 SPECIAL args=108   tok->spell = TEMP
 .SEM. act 15 DSMINMAJ args=   tok->spell = TEMP
 .SEM. act 17 DS args=  7  1  1  2  2  3  0   tok->spell = SKIP
 XSEM: OPENING DS  LIB=         MEM=TEMP     mode=1
 XSEM: OPENING DS  LIB=         MEM=TEMP     mode=87323972
  .SEM. act 5 OPT args=  4   tok->spell = ;
 .SEM. act 18 DSDFLT args=  7  1  1  2  2  3  0   tok->spell = ;
 .SEM. act 4 STMTEND args=   tok->spell = ;
 .SEM. act 2 STMTPROC args=   tok->spell = VAR
 .SEM. act 8 STMTLIST args=  8  3   tok->spell = VAR
 .SEM. act 9 VAR args=   tok->spell = X
 .SEM. act 10 VAREND args=  1   tok->spell = ;
 XSEM: STATEMENT NAME=EXAMPLE
 OPTIONS=  4
 NFLD=   20
 FIELD=  7 TYPE=  4 MODE=  1
       VALUE=SAS DATA SET: WORK.TEMP (INPUT)
```

```
FIELD=  8 TYPE=  3 MODE= 13
      VALUE= VAR LIST (BOTH)
  1
```

The following lists explains sections of the output.

| | |
|---|---|
| .SEM. | identifies it as a sentinel for semantic action debugging. |
| act n | indicates the semantic action number, followed by the name of the semantic action. This name is the one you place in your grammar, preceded by @. |
| args=*values* | are the arguments passed to the semantic action. |
| tok → spell | is the current token when the semantic is invoked. |
| XSEM: | marks other debug messages such as OPENING DS and STATEMENT NAME, described below. |
| OPENING DS | reports what data set is to be opened by the @DS semantic. |
| STATEMENT NAME | begins a dump of the statement contents. |
| OPTIONS=4 | indicates that option 4 has been set (via @OPT(4), associated with the SKIP option of PROC EXAMPLE). |
| NFLD=20 | shows there are 20 fields allocated (via the @STMTINIT(20)). |
| FIELD=7 | reports that Field 7 has a type 4 (indicating a data set) with mode 1 (indicating input). The VALUE= line reports the name of the data set and shows its open mode in parentheses. |
| FIELD=8 | is a type 3 (variable list) with mode 13 (both numeric and character permitted). The VALUE= line reports that it's a variable list with both types permitted. The '1' on the next line is the list of variable numbers in the list. |

If we resume our SAS session, and use the DEBUG value of TOK, we'll see the following in the log:

```
4          OPTIONS DEBUG='TOK';
5          PROC EXAMPLE DATA=TEMP SKIP; VAR X; RUN;

 .tok. type=  1 subtype=    1 ident=0 ss=1 loc=5:0 xzploc=6:0 ts= ;
 .tok. type=  2 subtype=    2 ident=0 ss=0 loc=5:1 xzploc=6:1 ts= PROC
 .tok. type= 27 subtype=-1984 ident=1 ss=0 loc=5:6 xzploc=6:6ts= EXAMPLE
 .tok. type= 25 subtype=-1776 ident=1 ss=0 loc=5:14 xzploc=6:14 ts= DATA
 .tok. type= 20 subtype=-1776 ident=0 ss=0 loc=5:18 xzploc=6:18 ts= =
 .tok. type=  2 subtype=    0 ident=1 ss=0 loc=5:19 xzploc=6:19 ts= TEMP
 .tok. type=  2 subtype=    0 ident=1 ss=0 loc=0:1 xzploc=7:1ts=  8 TEMP
  .tok. type= 38 subtype=-1776 ident=1 ss=0 loc=5:24 xzploc=6:24 ts= SKIP
 .tok. type= 20 subtype=    0 ident=0 ss=1 loc=5:28 xzploc=6:28 ts= ;
 .tok. type= 39 subtype=-1776 ident=0 ss=0 loc=5:30 xzploc=6:30 ts= VAR
 .tok. type=  2 subtype=    0 ident=1 ss=0 loc=5:34 xzploc=6:34 ts= X
 .tok. type= 20 subtype=    0 ident=0 ss=1 loc=5:35 xzploc=6:35 ts= ;
 .tok. type=  2 subtype=-1111 ident=0 ss=0 loc=5:37 xzploc=6:37 ts= RUN
 .tok. type= 20 subtype=    0 ident=0 ss=1 loc=5:40 xzploc=6:40 ts= ;
```

This reports only the tokens read.

type= is the token number associated with the token. Note that tokens numbered 1-11 are lexicals, and are not specifically located among your grammar's terminals. For example, TEMP and X above are type 2 (valid SAS names). They do not appear in the grammar, since they are lexicals. All other terminals are indeed located somewhere in the grammar. For example, '=' is terminal 20, and 'SKIP' is terminal 38.

ident= indicates whether the token is flagged as an identifier. Anything that is a valid SAS name is considered an identifier, regardless of whether the token is a terminal or a lexical. An exception is when the @SPECIAL(1) semantic is used that ensures that certain valid SAS names are not tagged as identifiers.

The subtype, ss, loc, and xzploc values are for internal use.

# Chapter 4 Writing Engines

# Introduction

An engine is a set of internal instructions that the SAS System uses to read from and write to files. SAS engines allow data in an familiar format to be presented to the SAS System so that it appears to be a standard SAS data set. Engines supplied by SAS Institute (either as part of the base product or as SAS/ACCESS products) consist of a large number of subroutines, all of which are called by the portion of the SAS System known as the *engine supervisor.* However, for SAS/TOOLKIT software, an additional level of software, the *engine middle-manager* simplifies how you write your user-written engine. This chapter reviews how to use an engine in a SAS program, describes how to write user-written engines, and explains how your routines interface with the engine middle-manager.

## Languages Available

Currently, the languages supported for writing engines are C, PL/I, and IBM Assembly Language. The discussion in this chapter uses C for illustrations. To see how to implement engines in PL/I, examine the PENGXMPL sample program. To see how to implement engines in IBM Assembly Language, examine the AENGXMPL sample program. Ask your SAS Software Representative to help you locate where these sample programs were stored when Release 6.08 of SAS/TOOLKIT Software was installed at your site.

# Using Engines with SAS Programs

Version 6 of the SAS System allows you to specify an engine name on a LIBNAME statement. In most cases, you don't need to specify an engine name because by default, the system uses the basic engine for the current release or detects the appropriate engine for certain other releases. To explicitly use other engines, specify the engine name on the LIBNAME statement. Use the following form of the LIBNAME statement:

**LIBNAME** *libref* <*engine*> '*physical-name*' <*options*>;

where

*libref* is a name temporarily associated with the SAS data library

*engine* is the set of internal instructions that the SAS System uses to read from and write to the file named in *physical-file.*

*physical-file* names the physical file that is to be read by the engine and treated as a SAS data library. Using an engine allows you to read data files in non-SAS formats as SAS data sets.

*options* are host- or engine-specific options. This discussion focuses on the uses of engine-specific options. Options are in the format

```
option-name=option-value
```

The `option-value` can be numeric, a valid SAS name, or a quoted string, depending on how the engine defines the option. These are examples of valid engine options:

```
OPT1=5
CHAROPT=ABC
MYINFO='x y z'
```

In addition to the basic format of the LIBNAME statement shown earlier, users can specify this format to clear a libref previously established:

**LIBNAME** *libref*;

For more information on the LIBNAME statement, refer to page 431 of *SAS Language: Reference, Version 6, First Edition.*

## An Engine Versus a Procedure

To process data from an external file, you can write either an engine or a SAS procedure. In general, it is a good idea to implement data extraction mechanisms as procedures instead of engines. If your applications need to read most of a data file or if they need random access to the file, you probably need to create an engine. This section discusses these types of modules in more detail.

### Why Write an Engine

Engines enable the user to extract data from some external file and read it directly into a SAS application. The main advantage to an engine is that it reads observations one at a time from the external file, based on the application's request for the observations. There is no need to store the entire set of observations read by the engine, unless the application requires it for special processing. Therefore, if the amount of data read by the engine is quite large, much less media space is needed than for a procedure that would need to store a large portion of the file in the form of a SAS data set.

Another advantage of an engine is that it can permit random access to the data file. If your application calls for random reads or writes of the data, an engine will work better.

The primary advantage of writing an engine instead of a procedure is that the application can use the external data in a more transparent fashion. By simply specifying the engine name in a LIBNAME statement, your users can access the file with standard SAS statements. If you are converting a SAS application to run with an external file format, developing an engine allows the rest of the application to remain unchanged. This may be a major factor if your pre-existing application is large and difficult to maintain or modify.

### Why Write a Procedure

You can also write a procedure that reads external file formats and creates a SAS data set that can be used by any SAS application. The primary reason for writing a procedure instead of an engine is when you want to specify complex selection criteria for extracting data from the file. The syntax for a procedure is much more flexible than that of an engine. You can use options, parameters, and statements in a procedure to provide information. It is easier to rename data base fields, supply selection criteria, or provide passwords with a procedure. With an engine, all this information must be provided via options on a LIBNAME statement, whose syntax can become quite complicated compared to procedure syntax.

# Writing the Engine

This section provides an overview of how to write an engine. When you write an engine, you must include in your program a prescribed set of routines to perform the various tasks required to access the file and interact with the SAS System. These routines open and close the data set, obtain information about variables, and read and write observations. In addition, your program uses several structures defined by the SAS System for storing information needed by the engine and the SAS System. These structures, ENGSTAT, NAMESTR, XONLIST, POSLNG, and FILEID, are all described in "Data Structures Used with Engines" on page 47.

The SAS System interacts with your engine through the SAS engine middle-manager, the SASENGMM module. The following section describes how the engine middle-manager uses the routines you call to interact with your engine.

**Note:** Many of the routines in the user-written engine are optional. If a routine is not provided, the engine middle-manager performs a default operation.

## Accessing SAS/TOOLKIT Header Files

At the beginning of your program, include the SAS/TOOLKIT header files:

```
#include <uwproc>
#include <engdef>
```

## Defining the Record Id Structure

Your engine must describe the record id (rid) structure for the data file. The *record id* is a unique indentifier for each observation in a data set. The rid is used to locate the record in the data file. The format for the record id differs depending on the engine that accesses the file. For many engines, the record id is only a temporary identifier for the observation. For more information on record ids and how they are used by SAS/TOOLKIT software, refer to *SAS/TOOLKIT Software: Usage and Reference, Version 6, First Edition*. The sample program uses this simple RID structure:

```
struct RID {
char ridchar;                   /* standard rid identifier          */
long loc;                       /* value returned from ftell        */
};
```

The first byte of the structure must a reserved byte. Your engine can test this byte to see if it contains the value 0x00. If so, this is a rid to one of the records in the data file your engine is processing. There are other reserved values for this byte that your engine can test and handle. These reserved values are described in "ENGPNT Routine" on page 66.

The remainder of the rid is completely dependent on your application. In the example, a simple call to `ftell` returns the byte location of the record, so all that is needed is a `long` to store the location.

## Initialization

The first executable part of the engine consists of the initialization routine. This routine is called when the engine is first invoked by the LIBNAME statement. Begin the routine with the following:

```
int   U_ENG( engine-name ) ( argc , argv )
int    argc;
char * argv[];
{
struct ENGSTAT engstat;
```

where *engine-name* is the external module name for the engine.

For the first executable statement, you must call the UWPRCC routine to establish the environment to allow SAS_ routine calls. Then zero out the ENGSTAT structure so that you can set only the necessary fields and ensure all others are zero.

```
UWPRCC(0);
SAS_ZZEROI((char *)&engstat,sizeof(struct ENGSTAT));
```

Next, set the various ENGSTAT structure fields that describe the characteristics of your engine. Refer to "ENGSTAT Structure" on page 47 for detailed information on these fields. This example sets the capabilities as follows:

```
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 phsname allowed  */
engstat.ridlen   = sizeof(struct RID); /* rid structure size         */
memcpy(engstat.engname,"ENGXMPL ",8); /* engine name                */
```

**Note:** It is not necessary to specify the fields that are set to zero since the call to SAS_ZZEROI set all fields to zero. This example explicitly sets these fields so it is clear which capabilities are not supported.

After setting all the appropriate ENGSTAT fields, call the UWENGC routine. This establishes the C engine interface. In the call to UWENGC, you pass the `argv` argument that was passed to your main routine and the address of the ENGSTAT structure:

```
UWENGC(argv,&engstat);
```

If you have other initialization related to your application, include that at this point. Finally, complete initialization by returning SUCCESS.

```
return(SUCCESS);
}
```

Refer to Example 4.1 on page 29 for an example of the initializing portion of the program.

## Parts of a User-Written Engine

This section describes the complete set of routines that you can use to write a fully-functional engine. Keep in mind that your engine might not need to perform all the functions described here. Table 4.1 lists which steps are needed for various functions. Note that if you create any of the optional routines, you must set a flag in the ENGSTAT structure to indicate that the routine exists.

**Table 4.1** *Required Step for Writing Engines*

| If your engine performs this task . . . | include these steps in your program | Optional or Required |
|---|---|---|
| open the file | 2 | required |
| read a record | 3, 5, and 7 | optional |
| write a record | 4 and 10 | optional |
| close the file | 11 | required |
| end the program | 13 | required |
| return to previously read records | 6 and 7 | optional |
| provide observation numbers for records | 8 | optional |
| directly access a specific record | 9 | optional |
| permit engine options or allow special handling of LIBNAME statement | 1 | optional |
| perform special termination tasks | 12 | optional |

1. The first routine you might need to write is the ENGASG routine. This is an optional routine. If you write this routine, it should contain and statements needed to initialize the engine, check special options for the engine, or perform special handling if the user omits the *physical-name* from the LIBNAME statement. By default, if the user omits the *physical-name*, the libref is de-assigned. However, if you want your engine to handle this condition differently, you can define an alternate action in the ENGASG routine. The ENGASG routine uses the ENGSTAT structure to indicate to the engine supervisor what special features the engine supports.
2. The engine middle manager passes a valid libref and physical name to the ENGOPN routine, the first required routine in your engine. Refer to "Example of ENGOPN" on page 32 for a sample of this routine. The ENGOPN routine is invoked when the SAS System encounters a request to open a member of the library referred to by the libref. Here are three examples of SAS statements that cause the ENGOPN routine to be invoked:

```
   /* read data set ABC from the XXX library */
proc print data=xxx.abc;
run;

   /* read data set DEF from the XXX library */
data temp;
   set xxx.def;
```

```
run;

   /* write data set GHI to the XXX library */
proc means data=temp;
   var y;
   output out=xxx.ghi mean=m;
run;
```

The first two examples request to open the member in input mode; the third example needs to open the member GHI in output mode. In all cases, the engine middle manager calls the ENGOPN routine to open the member of the library. To open a file depends upon what type of file your engine processes. If your engine processes a flat file, opening the file means calling the C `fopen` routine. If your engine deals with a data base management system (DBMS), opening the file means making subroutine calls to the DBMS stating that you want to begin transactions against a given file.

Thus, the ENGOPN routine you write must appropriately open the file to be accessed and provide information to the SAS System so the system can understand the data passed to it. The ENGOPN routine uses certain structures to provide the information. These structures are described in more detail in "Data Structures Used with Engines" on page 47

If you want your engine to be able to provide information to PROC CONTENTS when it is invoked with this format:

```
proc contents data=libref._all_;
```

your engine must test to see if the LIBOPEN*x* mode is on when the ENGOPN routine is called by the engine middle manager. The engine might need to perform special processing for this mode. Refer to "ENGOPN Routine" on page 64 for more information. Note that this mode is also used by PROC COPY.

3. After the ENGOPN routine successfully opens the file to read, the engine middle manager calls the ENGNAM routine if the file is an input file. Refer to "Example of ENGNAM" on page 38 for a sample of this routine. If your engine can be used to read files (not just write them), ENGNAM is one of the routines you are required to write. The ENGNAM routine must fill in a set of NAMESTR structures, which describe all the variables in the data set. The ENGNAM routine might need to store position and length information in the POSLNG structures, if these data will be needed again later by the engine.

4. If your engine can write output to the file, you must also define an ENGDFV routine. Refer to "Example of ENGDFV" on page 40 for a sample of this routine. The ENGDFV routine describes how to translate the variable information stored in the NAMESTRs to the the correct format for the output file.

5. You must create the ENGRED routine to actually read records from the file and store them in the observation buffer. Refer to "Example of ENGRED" on page 42 for a sample of this routine. The ENGRED routine moves the data to the observation buffer using the positions and lengths stored in the POSLNG structures or by using positioning information provided by the data base system your engine interfaces with.

6. If your engine needs to be able to reposition to a record in the file after initially reading it, you must create the ENGNOT. Refer to "Example of ENGNOT" on page 45 for a sample of this routine. The ENGNOT routine notes the current position of a record (called the record id or rid) and stores this value. (Note that the ENGNOT routine is responsible for defining the rid. This is explained in more detail in "ENGNOT Routine"

on page 62.) To reposition in a file, you must also create the ENGPNT routine, described in the next step.

7. You create the ENGPNT routine to return to a record previously noted by ENGNOT. You must also create this routine to be able to position to the beginning of the file that you are reading. Thus, ENGPNT is required for all engines that allow read or update access. Refer to "Example of ENGPNT" on page 45 for a sample of this routine. Once you have positioned to a record by calling ENGPNT, you can read or write the record using ENGRED or ENGWRT.

8. Your engine might also need to be able to provide an observation number for the records in the file. To do this, you must create the ENGR2N routine to convert the record id (rid) to an observation number. Most engines should be able to do this simply by storing the observation number in the rid itself. Your engine should include this routine to support procedures, such as PROC PRINT, that print observation numbers for observations.

9. You might also need to provide the ability to convert observation numbers into rids, which must be defined in the ENGN2R routine. This capability is required if your users want to use the POINT= option on the SET statement of the DATA step or if they want to position to a specific observation using the FSEDIT or FSBROWSE procedures. Not all engines provide this capability. In some file formats, the record id information may provide enough information to create an observation number. If the position cannot be computed from the observation number, either the engine must build a table of all observation numbers with matching rids, or the engine must indicate in the ENGSTAT structure that n-to-rid capability is not supported.

10. You must create the ENGWRT routine to write data from the observation buffer to the output file. Refer to "Example of ENGWRT" on page 44 for a sample of this routine. The ENGWRT routine uses the positions and offset stored in the NAMESTR structures to move the data to the output file. The ENGWRT routine must interact with the output file as necessary to update the file.

11. You must also create the ENGCLS routine to close the data file and terminate any processing that was initiated since the ENGOPN routine was invoked. Refer to "Example of ENGCLS" on page 37 for a sample of this routine.

    If your ENGOPN routine can provide information to PROC COPY and to PROC CONTENTS when it is invoked with this format:

    ```
    proc contents data=libref._all_;
    ```

    your ENGCLS routine must also handle the LIBCLOS*x* mode. Refer to "ENGCLS Routine" on page 57 for more information.

12. You may need to create the ENGDAS routine to deassign the libref. This routine is invoked by the engine middle manager when a libref is cleared by the user, when a new LIBNAME statement is issued using the same libref, or when the SAS job terminates. This routine is required only if your engine needs to perform any termination that corresponds to initialization performed by ENGASG.

13. Finally, you must terminate the engine environment with the ENGTRM routine. Refer to "Example of ENGTRM" on page 47 for a sample of this routine.

## Special Error Conditions

Your engine may need to handle error conditions beyond the scope of the standard return codes. If you have special conditions, return the W_ESYSER return code from any of the appropriate ENGxxx routines. However, if you return this return code, you must also set the `errmsg` and `errmsgl` fields in the FILEID structure. `errmsg` must be a pointer that is set to the message that you want to print. `errmsg` is the length of the message. There is no limit to the length of the error message. If you return W_ESYSER and have left errmsg as NULL, or have set errmsgl to any value less than 1, this message appears:

```
Engine did not provide proper error message.
```

It is best to define errmsg and errmsgl properly before returning W_ESYSER, so your user will know the real reason for the failure.

# Example

The sample engine, CENGXMPL, is provided with Release 6.08 of SAS/TOOLKIT software. It is reprinted here for your reference.

Note that in this example, the SPECFID and RID structures are defined. The RID structure refers to the layout of a rid as used in this engine. For this engine, a rid consists of the identifying rid character, followed by a long integer that is used by the `ftell` and `fseek` I/O routines. The identifying rid character is 0x00 for the generated rids in this example. The SPECFID structure contains additional fields that are needed for this particular engine application.

**Example 4.1**
*CEXAMPL Example of Data Base Engine*

```
/*------------------------------------------------------------------*
/*------------------------------------------------------------------*/
/* NAME:      CENGXMPL */
/* PRODUCT:   SAS/TOOLKIT */
/* PURPOSE:   Sample engine */
/* TYPE:      Engine */
/* NOTES:     This sample engine demonstrates how to write simple     */
/*            engines that interface with the engine "middle manager" */
/*            (which in turn interfaces with the engine supervisor).  */
/*            Our sample engine reads and writes sequential files     */
/*            in a simple format. The sequential file can have many   */
/*            logical members. Each member begins with the record     */
/*                TOF-TOF-TOF-TOF                                     */
/*            and ends with the record                                */
/*                EOF-EOF-EOF-EOF                                     */
/*            The first "real" record of each member contains the     */
/*            name of the member, followed by the number of variables.*/
/*            The next n records (where n=number of variables) contain*/
/*            the variable name, followed by N or C (for numeric or   */
/*            character), followed by an optional variable length.    */
/*            The remaining records for the member are the observation*/
/*            records. There is one physical record for each obser-   */
/*            vation.                                                 */
/*            Suppose we have 2 members, named TEMP1 and TEMP2. TEMP1 */
/*            has 1 variable, X, which is numeric. TEMP2 has 2        */
```

```
/*           variables, A (character length 8) and B (numeric std    */
/*           length). TEMP1 has one observation with X being 1.      */
/*           TEMP2 has 2 observations, with A being ABC and B being  */
/*           1 for the first observation, and DEF and 2 for the      */
/*           second observation. The entire physical file would look */
/*           like this (all records starting in column 1):           */
/*                TOF-TOF-TOF-TOF                                    */
/*                TEMP1 1                                            */
/*                X N                                                */
/*                1                                                  */
/*                EOF-EOF-EOF-EOF                                    */
/*                TOF-TOF-TOF-TOF                                    */
/*                TEMP2 2                                            */
/*                A C 8                                              */
/*                B N                                                */
/*                ABC 1                                              */
/*                DEF 2                                              */
/*                EOF-EOF-EOF-EOF                                    */
/*           To allow for nested blanks in character data, all       */
/*           underscores are converted to blanks.                    */
/*                                                                   */
/*           For simplicity's sake, this engine only allows          */
/*           character variables to have lengths up to 12. Any data  */
/*           after 12 bytes will be ignored and not be written out.  */
/*                                                                   */
/*           This example assumes the input file will not be damaged */
/*           in any way, but does check for nonexistent member.      */
/*-------------------------------------------------------------------*
/*-------------------------------------------------------------------*/

/*-----include files needed-----------------------------------------*/
#include <stdio.h>
#include <uwproc.h>
#include "engdef.h"
#define BLANKS "                                          "
#define LAST_IO_READ 1
#define LAST_IO_WRITE 2

/*-----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 containanything    */
/* 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[];
{

/*--------------------------------------------------------------------*
/*--------------------------------------------------------------------*/
/* The main routine is responsible for establishing the SAS/Toolkit   */
/* SAS_   environment, and to indicate the proper status for this     */
/* engine. The environment is established by calling UWPRCC. This     */
/* allows any SAS_ routine to be called from a SAS/Toolkit engine.    */
/* The engine status is described by setting various fields in the    */
/* ENGSTAT structure.  After this structure is properly set, the      */
/* UWENGC routine is called to initialize the SAS/Toolkitengine       */
/* environment. This environment establishment will load the engine   */
/* "middle manager" and will control all subsequent callsfrom  the    */
/* engine supervisor to the user-written engine.                      */
/*--------------------------------------------------------------------*
/*--------------------------------------------------------------------*/

struct ENGSTAT engstat;

/*-----initialize SAS/Toolkit SAS_ routine environment----------------*/
UWPRCC(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 phsname 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);
}
```

## Example of ENGOPN

```
/*--------------------------------------------------------------------*
/*--------------------------------------------------------------------*/
/* The ENGOPN routine is responsible for opening the sequential       */
/* file.  For input/update, we will search through the file looking   */
/* for the requested member. For output, we will first try to find    */
/* the member (via an open for input). If we find the member, we      */
/* will overwrite starting at the logical member location.  If we     */
/* don't find the member, we append the new member onto the end of    */
/* the file.                                                          */
/*                                                                    */
/* Libmode explanation:                                               */
/*     NORMOPEN          normal open                                  */
/*     LIBOPEN1          first input open for what will be a PROC     */
/*                       CONTENTS series of opens                     */
/*     LIBOPENN          subsequent opens of a PROC CONTENTS series   */
/*     LIBOPENY          second open request for a member, which can  */
/*                       usually just be ignored                      */
/*                                                                    */
/* What happens with a PROC CONTENTS open is that a request will      */
/* first be made to obtain the member name, either with aLIBOPEN1     */
/* or LIBOPENN call. We will go ahead and open the memberwith  that   */
/* open request. Then, PROC CONTENTS will request that the member     */
/* with that name be opened. This is a LIBOPENY call, which can be    */
/* effectively ignored in our example, since the member has already   */
/* been opened.                                                       */
/*                                                                    */
/* Summary of required actions for ENGOPN:                            */
/* * open the member with the requested mode, returning the correct   */
/*   return code if unsuccessful. Return codes are XHENOLIB if the    */
/*   library can't be opened; XHENOMBR if the member can't be opened, */
/*   X_ENOMEM if there's insufficient memory. X_WEOF is acceptable    */
/*   for PROC CONTENTS-style opens (LIBOPENN mode). Any other         */
/*   condition warrants the W_ESYSER setting with a message and len   */
/*   in the fid.                                                      */
/* * set the infostr.memname in the fileid (input only)               */
/*   this is necessary in the case of a directory read (for PROC      */
/*   COPY or PROC CONTENTS) to determine the next member name.        */
/* * set the infostr.num_prec (number of observations) (input only)   */
/*   (this can be set to -1 if the number of observationsis  unknown) */
```

```
/* * set the infostr.num_vars in the fileid (input only)              */
/* * set the infostr.max_rc (usually to 0)                           */
/* * set the infostr.num_lrec (usually to MACLONG)                   */
/* * set the infostr.crdate and infostr.modate (creation and mod     */
/*   dates)                                                         */
/* * set the infostr.label (data set label)                         */
/*------------------------------------------------------------------*
/*------------------------------------------------------------------*/
rctype ENGOPN(fid,libmode)
fidptr fid;                  /* ptr to fileid ptr                   */
int libmode;                 /* open status                         */
{
int j,k,l;
char *p,*mode;
long rc;
FILE *fp;
char temp(|49|);

/*-----simply return for LIBOPENY status since we're already open-----*/
if (libmode == LIBOPENY)
   return(SUCCESS);

/*-----allocate our extended fileid for our engine's needs---------*/
if (fid->specfid == NULL)
   fid->specfid = (struct SPECFID *)
      SAS_XMALLOC(fid->poolid,sizeof(struct SPECFID),XM_ZERO);

/*------------------------------------------------------------------*
/*------------------------------------------------------------------*/
/* When we open for output, this engine behaves differently if the  */
/* member already exists. Therefore, we first attempt to open the   */
/* member for input. If it's found, we will open the physical file  */
/* with mode "r+", which indicates that we''l open in update mode,  */
/* at the beginning of the file. Later on, we'll reposition to the  */
/* place in the file where the member starts, so we can begin       */
/* overwriting there. If the member isn't found, we use mode "a"    */
/* which indicates append mode.                                     */
/*------------------------------------------------------------------*
/*------------------------------------------------------------------*/

j = k = fid->opnmode;
mode = " ";
if (fid->opnmode & XO_OUTPUT) {
   int opnmode;

   /*-----reset open mode to input after saving original mode---------*/
   opnmode = fid->opnmode;
   fid->opnmode &= ~XO_OUTPUT;
   fid->opnmode |= XO_INPUT;

   /*-----open for input; set mode to update if we found it-----------*/
   if (ENGOPN(fid,libmode) == SUCCESS) {
      mode = "r+";
```

```
      ENGCLS(fid,0,libmode);
      }
   else mode = "a";

   /*-----reset to original open mode---------------------------------*/
   fid->opnmode = opnmode;
   }

/*-----standard mode "r" for standard input--------------------------*/
else if (fid->opnmode & XO_INPUT)
   mode = "r";

/*-----standard mode "r+" for standard update------------------------*/
else if (fid->opnmode & XO_UPDATE)
   mode = "r+";

/*-------------------------------------------------------------------*
/*-------------------------------------------------------------------*/
/* Open the physical file with the appropriate mode. For input       */
/* mode, we do expect that the file will exist, so we will first     */
/* determine its existence with the access routine. This avoids      */
/* unnecessary messages about the file not existing if we're trying  */
/* to find a member to reset mode to update. If the file doesn't     */
/* exist, and our open mode is input, the XHENOLIB (library not found)*/
/* return code is returned. If the file is found (access is success- */
/* fule), we try to fopen the file. If that open isn't successful,   */
/* we will produce the message "File ... could not be opened         */
/* successfully." We allocate the memory for the message and build   */
/* it with SAS_XPSSTR. We set fid->errmsg and fid->errmsgl according- */
/* ly, and use the W_ESYSER return code to signal this condition.    */
/* Note that the XHENOMEM return code would not be sufficient here,  */
/* an unsuccessful open of the entire file is another matter.        */
/* If the original open was for output, but we've alreadydetermined  */
/* from a prior open that the logical member exists, we should fseek */
/* to the location of that member.                                   */
/*-------------------------------------------------------------------*
/*-------------------------------------------------------------------*/

if (libmode == NORMOPEN || libmode == LIBOPEN1) {

#if SASC
   /*-----use libname if physical name not specified in LIBNAME stmt--*/
   if (strlen(fid->physname) == 0) {
      strcpy(temp,"ddn:");
      strcat(temp,fid->libname);
      p = temp;
      }
#if MVS
   else if (memcmp("dsn:",fid->physname,4) != 0) {
      strcpy(temp,"dsn:");
      strcat(temp,fid->physname);
      p = temp;
      }
```

```
#endif
#if CMS
   else if (memcmp("cms:",fid->physname,4) != 0) {
      strcpy(temp,"cms:");
      strcat(temp,fid->physname);
      p = temp;
      }
#endif
   else
#endif
      p = fid->physname;

   /*-----attempt to open the file-------------------------------------*/
   fid->specfid->fp = fopen(p,mode);

   /*-----produce message if that open failed--------------------------*/
   if (fid->specfid->fp == NULL) {
      fid->errmsg = SAS_XMALLOC(fid->poolid,256,XM_EXIT);
      fid->errmsgl = SAS_XPSSTR(fid->errmsg,256,"File %s could not be \
opened successfully",fid->physname);
      fid->errmsg(|fid->errmsgl++|) = 0;
      return(W_ESYSER);
      }

   /*-----reposition if we're updating an existing member-------------*/
   if ((fid->opnmode & XO_OUTPUT) && !memcmp(mode,"r+",2)) {
      fseek(fid->specfid->fp,(fid->specfid)->headrid.loc,0);
      }
   }

/*--------------------------------------------------------------------*
/*--------------------------------------------------------------------*/
/* If we're opening for input, we search through the physical file    */
/* to search for our member. If libmode is NORMOPEN, we must find     */
/* an exact match on the member name. Otherwise, we are reading to    */
/* the next member, and we fill in the next member name we find. Note */
/* that we may not be positioned at the beginning of a member, and so */
/* we must search for a TOF tag. We'll either find that or hit EOF    */
/* trying.                                                            */
/*--------------------------------------------------------------------*
/*--------------------------------------------------------------------*/

fp = fid->specfid->fp;
if (fid->opnmode & (XO_INPUT | XO_UPDATE)) {
   char record[82];

   p = record;

   while(((fid->specfid->headrid.loc = ftell(fp)) || 1) &&
         (rc = (fgets(p,80,fp)) != NULL)) {

      l = strlen(p) - 1;
```

```
      /*-----look for the TOF record---------------------------------*/
      while(rc && !(l >= 15 && !memcmp("TOF-TOF-TOF-TOF",p,15))) {
         fid->specfid->headrid.loc = ftell(fp);
         rc = (fgets(p,80,fp) != NULL);
         l = strlen(p) - 1;
         }

      /*-----leave if there are difficulties (EOF or anything else)---*/
      if (!rc)
      break;

      /*-----now read the first record after the TOF record-----------*/
      fgets(p,80,fp);
      l = strlen(p) - 1;

      /*-----copy memname to a char8 and get the num_vars count-------*/
      j = SAS_ZSTRPOS(p,l,' ');
      SAS_ZSTRMOV(p,j,temp,8);
      p += j+1; l -= (j+1);
      SAS_ZSTOL(p,l,0,&j,&fid->num_vars);

      /*-----we've found the member if a match or not NORMOPEN--------*/
      if (libmode != NORMOPEN || !memcmp(fid->memname,temp,8))
         break;
      }

   /*-----if at EOF, we indicate EOF or member not found accordingly--*/
   if (!rc) {
      if (libmode != NORMOPEN)
         return(X_WEOF); /* indicates no more members */
      rc = XHENOMBR; /* not found */
      goto badopen;
      }

   /*-----set the memname, memtype, and num_prec as appropriate-------*/
   memcpy(fid->memname,temp,8);
   fid->num_prec = -1;
   }

/*-----if openmode is output-----------------------------------------*/
else {

   /*-----write a leading TOF record-------------------------------*/
   fputs("TOF-TOF-TOF-TOF\n",fp);
   fid->num_prec = 0;
   }

/*-----initialize the remaining fields as appropriate----------------*/
fid->max_rc = 0;
fid->num_lrec = MACLONG;
fid->crdate = fid->modate = SAS_ZDATTIM();
fid->lablen = 0;
fid->label = NULL;
```

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

/*-----for a problem after an open, ensure we've closed the file------*/
badopen:;
ENGCLS(fid,0,0);
return(rc);
}
```

## Example of ENGCLS

```
/*--------------------------------------------------------------------*
/*--------------------------------------------------------------------*/
/* The ENGCLS routine is responsible for closing the sequential file. */
/* for input or output.                                               */
/* Libmode explanation:                                               */
/*                                                                    */
/*     NORMCLOS          normal close                                 */
/*     LIBCLOSN          a close corresponding to an ENGOPN for       */
/*                       LIBOPEN1 or LIBOPENN                         */
/*     LIBCLOSL          the last requested close                     */
/*                                                                    */
/* What happens with a PROC CONTENTS open is that a LIBOPEN1 is       */
/* requested, followed by LIBOPENY, followed by LIBCLOSN,followed     */
/* by LIBOPENN, LIBCLOSN, LIBOPENN, LIBCLOSN, ...  until the last     */
/* close, which will be with libmode LIBCLOSL. What we doin  this     */
/* example is to perform a close on the physical file only with a     */
/* LIBCLOSL (or a NORMCLOS). For output members, we will always       */
/* write out an EOF record, since all output open/close operations    */
/* are with NORMOPEN/NORMCLOS. We also perform some freeing of        */
/* memory allocated during open time.                                 */
/*                                                                    */
/* Summary of required actions for ENGCLS:                            */
/* * close the member, and also the physical file if necessary        */
/* * perform any additional closing processing                        */
/*--------------------------------------------------------------------*
/*--------------------------------------------------------------------*/
rctype ENGCLS(fid,disp,libmode)
fidptr fid;                    /* fileid from engine middle mgr       */
int disp;                      /* disposition (ignored)               */
int libmode;                   /* libmode (explained above)           */
{

/*-----perform physical close if necessary----------------------------*/
if (libmode == NORMCLOS || libmode == LIBCLOSL) {
   if ((fid->opnmode & XO_OUTPUT) ||
            ((fid->opnmode & XO_UPDATE) && fid->specfid->newobs))
      fputs("EOF-EOF-EOF-EOF\n",fid->specfid->fp);
   fclose(fid->specfid->fp);
   }
```

```
/*-----indicate success-------------------------------------------------*/
return(SUCCESS);
}
```

## Example of ENGNAM

```
/*---------------------------------------------------------------------*
/*---------------------------------------------------------------------*/
/* The ENGNAM routine is responsible for creating the namestr struc-  */
/* tures for all input variables, and for filling in the xonlist      */
/* array with the variable names and types. The namestr array has     */
/* already been allocated by the middle mgr. ENGNAM also fills in     */
/* an array of namestr pointers, and an array of POSLNG structures    */
/* to be used later by the middle mgr.                                */
/*                                                                    */
/* The pos indicator we use is an offset into the buffer that we will */
/* be building in ENGRED. Pos is incremented based on thedesired      */
/* length of each variable.                                           */
/*                                                                    */
/* For each variable, ENGNAM                                          */
/* * Fills in a namestr structure, setting nname, ntype, nlng, nid,   */
/*   nsubtype, nfj, nlabel, npos, nvar0, nform, nfl, nfd,nifl,  and   */
/*   nifd. All fields are initialized to zero before ENGNAM is        */
/*   called, so all fields that can appropriately be zeroneed  not be */
/*   set.                                                             */
/* * Adds the namestr pointer into the ppn array of pointers.         */
/* * Puts the npos and nlng into the POSLNG array. The lng value      */
/*   should be negative if we're defining a character variable.       */
/* * Puts the nname and ntype into the xonlist array.                 */
/*                                                                    */
/* In addition, ENGNAM                                                */
/* * sets infostr.rec_len with the length of the input buffer that    */
/*   we will fill in within ENGRED. Note that this is theresulting    */
/*   buffer length. This buffer is better known as the program data   */
/*   vector, or PDV.                                                  */
/* * creates a special rid that corresponds to the RIDBODlogical      */
/*   value. After all, the next call to the engine code will be       */
/*   the first read. Therefore, we need to know the first observation */
/*   location beforehand. In this example, we get the position by     */
/*   calling the ftell routine.                                       */
/*---------------------------------------------------------------------*
/*---------------------------------------------------------------------*/

rctype ENGNAM(fid,pxonl)
fidptr fid;                 /* fileid from engine middle mgr         */
xonlptr  pxonl;             /* ptr to xonlist                        */
{
nameptr np;
nameptr *ppn;
long j;
long i,pos,s;
int l;
```

```
char *rp;
struct POSLNG *psl;
FILE *fp;
char *namelist;

/*-----get local copies of pointers that will be incremented----------*/
psl = fid->psl;
ppn = fid->ppn;

/*-----read through all variable records------------------------------*/
fp = fid->specfid->fp;
namelist = SAS_XMALLOC(fid->poolid,8*fid->num_vars,0);
for (i=pos=0;i<fid->num_vars;i++,psl++) {
   char record[82];

   rp = record;

   /*-----read a variable record-------------------------------------*/
   fgets(record,80,fp);
   l = strlen(record);

   /*-----copy in the name field into nname--------------------------*/
   j = SAS_ZSTRPOS(rp,l,' ');
   np = ppn(|i|);
   np->nname = namelist; namelist += 8;
   np->namelen = min(j,8);
   SAS_ZSTRMOV(rp,j,np->nname,8);

   /*-----set ntype from N or C indication---------------------------*/
   rp += j+1;
   np->ntype = (*rp == 'N') ? 1 : 2;

   /*-----set nlng from the length indicator-------------------------*/
   if (np->ntype == 2) {
      rp += 2; l -= j+3;
      SAS_ZSTOL(rp,l,0,&j,&s);
      np->nlng = s;
      }
   else np->nlng = sizeof(double);

   /*-----set other values as constants------------------------------*/
   np->nfj = 1;
   np->nlabel = BLANKS;
   np->nlablen = 0;
   SAS_ZFILLCI(' ',np->nform,8);
   SAS_ZFILLCI(' ',np->niform,8);

   /*-----set the two POSLNG fields----------------------------------*/
   psl->pos = np->npos = pos;
   psl->lng = np->nlng;
   if (np->ntype == 2)
      psl->lng = -(psl->lng);
```

```
   /*-----nvar0 is the 1-based variable number-----------------------*/
   np->nvar0 = i+1;

   /*-----nfl, nfd, nifl, nifd all 0 in this example-----------------*/

   /*-----fill in the xonlist element--------------------------------*/
   memcpy(pxonl->name,np->nname,8);
   pxonl++->type = np->ntype;

   /*-----increment our running position indicator-------------------*/
   pos += np->nlng;
   }

/*-----set the rec_len accordingly-----------------------------------*/
fid->rec_len = pos;

/*-----for UPDATE mode, we'll need a buffer--------------------------*/
fid->specfid->outbuf = SAS_XMALLOC(fid->poolid,
(long)(13*fid->num_vars),0);

/*-----note the location ofthe  first observation--------------------*/
fid->specfid->engrid1.loc = ftell(fp);
return(SUCCESS);
}
```

## Example of ENGDFV

```
/*-------------------------------------------------------------------*
/*-------------------------------------------------------------------*/
/* The ENGDFV routine is responsible for writing out the namestr     */
/* structures into the format that is correct for the output file.   */
/* The ppn field in the fileid points to an array of namestr pointers.*/
/* These namestrs are written out as appropriate. In our example, the */
/* variable name, N or C, and a length for character variables is    */
/* written for each namestr.                                         */
/*                                                                   */
/* For each variable, ENGDFV                                         */
/* * Writes out a namestr structure to the output file inwhatever    */
/*   manner is appropriate.                                          */
/* * Puts the npos and nlng into the POSLNG array. The lng value     */
/*   should be negative if we're defining a character variable.      */
/*                                                                   */
/* In addition, ENGDFV                                               */
/* * Optionally writes any header or trailer data                    */
/* * Sets prec_len to the output buffer record length. Note that     */
/*   this is the buffer whose pointer will be passed to the ENGWRT   */
/*   routine. It contains the data that will be convertedto  the     */
/*   proper format to be written out.                                */
/*-------------------------------------------------------------------*
/*-------------------------------------------------------------------*/
```

```
rctype ENGDFV(fid,prec_len)
fidptr fid;
long *prec_len;
{
long i,def_nvar,pos;
int j,l;
char temp[40];
struct POSLNG *psl;
nameptr *ppn;
char c;
FILE *fp;

/*-----set local variables-----------------------------------------*/
def_nvar = fid->num_vars;
ppn = fid->ppn;
psl = fid->psl;

/*---------------------------------------------------------------------*
/*---------------------------------------------------------------------*/
/* We allocate our output buffer here. This buffer is where we will   */
/* write the image to be written out via xxwrite. We allow 13 bytes   */
/* for each variable. We use BEST12 for numerics, and allow a max     */
/* of 12 for character, and allow a trailing blank, for a total of    */
/* 13 bytes for each variable.                                        */
/*---------------------------------------------------------------------*
/*---------------------------------------------------------------------*/

fid->specfid->outbuf = SAS_XMALLOC(fid->poolid,13*def_nvar,0);
fp = fid->specfid->fp;

/*-----write out the header record with the namestr count-------------*/
j = SAS_ZSTRIP(fid->memname,8);
memcpy(temp,fid->memname,j);
temp[j++] = ' ';
SAS_ZLTOS(def_nvar,temp+j,10);
SAS_ZSTRJLS(temp+j,10,'l',&l,NULL);
j += l;
temp[j++] = '\n';
temp[j] = 0;
fputs(temp,fp);

/*-----write namestr records for each variable------------------------*/
for (i=pos=0;i<def_nvar;i++,psl++) {

   /*-----fill in POSLNG structure-----------------------------------*/
   l = ppn[i]->nlng;
   psl->lng = MIN(l,12);                  /* allowing only 12 bytes */
   psl->pos = ppn[i]->npos = pos;
   pos += l;

   /*-----build the namestr record----------------------------------*/
   j = SAS_ZSTRIP(ppn[i]->nname,8)+1;
SAS_ZSTRMOV(ppn[i]->nname,8,temp,j);
```

```
   if (ppn[i]->ntype == 2) {
      psl->lng = -(psl->lng);
      c = 'C';
      }
   else c = 'N';
   temp[j++] = c;
   if (c == 'C') {
      SAS_ZLTOS((long)l,temp+j,4);
      j += 4;
      }

   /*-----write out the record----------------------------------------*/
   temp[j++] = '\n';
   temp[j] = 0;
   fputs(temp,fp);
   }

/*-----initialize prec_len with current buffer size-------------------*/
*prec_len = pos;

return(SUCCESS);
}
```

## Example of ENGRED

```
/*--------------------------------------------------------------------*
/*--------------------------------------------------------------------*/
/* The ENGRED routine is responsible for filling in an observation    */
/* buffer, using the POSLNG array.                                    */
/*                                                                    */
/* ENGRED should                                                      */
/* * read data from the input file and convert the data into the      */
/*   input buffer according to the POSLNG array                       */
/* * return X_WEOF if there are no more observations                  */
/* * set rptr to the address of the input buffer                      */
/*                                                                    */
/* optionally, ENGRED should                                          */
/* * take a note if the ENGNOT routine is supported.                  */
/*--------------------------------------------------------------------*
/*--------------------------------------------------------------------*/

rctype ENGRED(fid,rptr)
fidptr fid;                 /* fileid from engine middle mgr         */
char **rptr;                /* returned pointer to input buffer      */
{
long i;
long ll;
char *rp,*cp;
int j,k,l,m;
double temp;
struct POSLNG *psl;
```

```
char record[82];
FILE *fp;

rp = record;
fp = fid->specfid->fp;

/*-----take the note for subsequent ENGNOT calls----------------------*/
fid->specfid->curr_rid.loc = ftell(fp);

/*-----read the next record from the input file----------------------*/
i = (fgets(record,80,fp) == NULL);
ll = strlen(record) - 1;

/*-----set local variables-------------------------------------------*/
l = ll;
cp = *rptr = fid->currec;
psl = fid->psl;

/*-----indicate EOF if EOF record seen-------------------------------*/
if (ll >= 15 && !memcmp("EOF-EOF-EOF-EOF",rp,15))
   return(X_WEOF);

/*-----go througheach variable---------------------------------------*/
for (j=0;j<fid->num_vars;j++,temp++,psl++) {

   /*-----strip off next token----------------------------------------*/
   SAS_ZSTRJLS(rp,l,'l',&l,NULL);
   m = SAS_ZSTRPOS(rp,l,' ');
   k = (m != -1) ? MIN(l,m) : l;

   /*-----convert to numeric if appropriate---------------------------*/
   if (psl->lng > 0) {
      SAS_XFXIN(rp,k,0,0L,&temp);
      memcpy(cp,&temp,sizeof(double));
      cp += sizeof(double);
      }

   /*-----otherwise copy in the characters----------------------------*/
   else {
      SAS_ZSTRMOV(rp,k,cp,-(psl->lng));
      m = MIN(k,-psl->lng);
      SAS_ZSTRANC(cp,m,' ','_');           /* convert _ to blank ---*/
      cp -= psl->lng;
      }

   /*-----to next field in record-------------------------------------*/
   rp += k; l -= k;
   }

fid->specfid->lastio = LAST_IO_READ;
return(0);
}
```

## Example of ENGWRT

```
/*-------------------------------------------------------------------*
/*-------------------------------------------------------------------*/
/* The ENGWRT routine is responsible for converting an observation   */
/* buffer into the proper output record format and writing the record.*/
/*                                                                   */
/* ENGWRT should                                                     */
/* * convert all values to the output format and write the record    */
/* * increment the num_prec value in the fileid by 1 to indicate     */
/*   another observation has been read                               */
/*-------------------------------------------------------------------*
/*-------------------------------------------------------------------*/

rctype ENGWRT(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             */
{
long i;
char *p,*pp;
double d,temp;
long totlen;
int k;
nameptr np;
struct POSLNG *psl;

/*-----get address of our temporary output buffer---------------------*/
pp = p = fid->specfid->outbuf;

/*-----local variables being incremented------------------------------*/
np = fid->np;
psl = fid->psl;

/*-----loop through all variables to convert to output format---------*/
for (i=fid->num_vars,totlen=0;i > 0;i--,np++,psl++) {

   /*-----for numerics, convert to BEST12 format and left-justify----*/
   if (psl->lng > 0) {
      memcpy(&temp,rptr+psl->pos,psl->lng);
      SAS_ZFPAD((ptr)&temp,psl->lng,&d);
      SAS_XFXPN(d,12,99,0L,p);
      SAS_ZSTRJLS(p,12,'l',&k,NULL);
      }

   /*-----for character, strip trailing blanks and convert blk to _--*/
   else {
      k = -psl->lng;
      memcpy(p,rptr+psl->pos,k);
      k = SAS_ZSTRIP(p,k);
      SAS_ZSTRANC(p,k,'_',' ');
      }
```

```
   /*-----ensure a trailing blank----------------------------------*/
   p(|k++|) = ' ';

   p += k; totlen += k;
   }

/*-----increment observation count-----------------------------------*/
fid->num_prec++;

/*-----write out the converted record--------------------------------*/
pp[totlen] = '\n';
pp[totlen+1] = 0;
if (status && fid->specfid->lastio == LAST_IO_READ) {
   fseek(fid->specfid->fp,(fid->specfid)->curr_rid.loc,0);
   }
if (!status)
   fid->specfid->newobs = 1;
fputs(pp,fid->specfid->fp);
fid->specfid->lastio = LAST_IO_WRITE;
return(SUCCESS);
}
```

## Example of ENGNOT

```
/*-------------------------------------------------------------------*
/*-------------------------------------------------------------------*/
/* The ENGNOT routine is responsible for filling in an area with the  */
/* rid of the most recently read observation. This rid was built when */
/* ENGRED was most recently called.                                   */
/*-------------------------------------------------------------------*
/*-------------------------------------------------------------------*/
rctype ENGNOT(fid,ridp)
fidptr fid;
char *ridp;
{
memcpy(ridp,&fid->specfid->curr_rid,sizeof(struct RID));
return(SUCCESS);
}
```

## Example of ENGPNT

```
/*-------------------------------------------------------------------*
/*-------------------------------------------------------------------*/
/* The ENGPNT routine is responsible for repositioning inthe  file    */
/* based on a rid value. It must also handle these   special rid      */
/* values:                                                            */
/*                                                                    */
/* * RIDBOD    beginning of data                                      */
/*                                                                    */
/* These rid values must be handled if random access is supported:    */
```

```
/* * RIDCURR   the current observation                                  */
/* * RIDNEXT   the next observation                                    */
/* * RIDPREV   the previous observation                                */
/*                                                                     */
/* The first byte of the rid will indicate what kind of rid it is.    */
/* If the first byte is 0x00, this indicate a rid specific to this    */
/* engine. If the rid cannot be handled, return XHEBDCSQ.             */
/*---------------------------------------------------------------------*
/*---------------------------------------------------------------------*/

rctype ENGPNT(fid,ridp)
fidptr fid;
char *ridp;
{
struct RID *rid;
FILE *fp;
long curloc,ploc,loc;
char p(|82|);

fp = fid->specfid->fp;

/*-----handle special rids-------------------------------------------*/
if (*ridp == *RIDBOD) {
   fseek(fp,fid->specfid->engrid1.loc,0);
   }
else if (*ridp == *RIDNEXT) {
   /* no action necessary; we're already positioned there */
   }
else if (*ridp == *RIDPREV) {
   curloc = fid->specfid->curr_rid.loc;
   /* if we're already at beginning of data, we don't have
      to reposition */
   if (curloc == fid->specfid->engrid1.loc)
      return(X_WBOD);
   /* for this, we must point back to the first observation and
      read forward, recording each location before reading, until
      we match with the current record. Then, the previous read's
      location can be repositioned to. */
   fseek(fp,fid->specfid->engrid1.loc,0);
   while((loc = ftell(fp)) != curloc) {
      fgets(p,80,fp);
      ploc = loc;
      }
   fseek(fp,ploc,0);
   fid->specfid->curr_rid.loc = ploc;
   }
else if (*ridp == *RIDCURR) {
   fseek(fp,fid->specfid->curr_rid.loc,0);
   }

/*-----reject any other non-engine rids------------------------------*/
else if (*ridp != 0) {
   return(XHEBDCSQ);
```

```
   }

/*-----handle engine rids by calling fseek----------------------------*/
else {
   rid = (struct RID *)ridp;
   fseek(fid->specfid->fp,rid->loc,0);
   }
return(SUCCESS);
}

/*-------------------------------------------------------------------*
/*-------------------------------------------------------------------*/
/* The ENGTRM routine is responsible for terminating the engine      */
/* environment. Anything the engine must do to shut down the         */
/* environment goes here.                                            */
/*-------------------------------------------------------------------*
/*-------------------------------------------------------------------*/
```

### Example of ENGTRM

```
void ENGTRM() {
}
```

# Data Structures Used with Engines

This section describes the data structures you use when writing an engine. Each structure is illustrated and the fields of the structure are described.

# ENGSTAT Structure

**describes capabilities of engine**

The ENGSTAT structure contains values that indicate to the engine middle manager what special capabilities your engine supports. In your U_ENG code, first zero this structure and then set the flags for the capabilities your engine supports.

### Declaration

```
struct ENGSTAT {
char  *funcptrs;
short  support;
short  read;
short  write;
short  update;
short  random;
short  options;
```

## ENGSTAT Structure *continued*

```
short  assign;
short  note;
short  ridlen;
short  ridandn;
short  nopname;
char   engname[8];
short  expansion[20];
};
```

### Description

Each field is described in more detail below.

funcptrs
: Do not alter these pointers in any way. This is the array of function pointers that will be set by the engine middle manager.

support
: Set this to 1 in your main engine code, U_ENG. This indicates that the engine is supported on the host.

read
: Set this to 1 if your engine supports read access. If read=1, you must supply the ENGNAM, ENGRED, and ENGPNT routines. If read=0, the engine will be a write-only engine.

write
: Set this to 1 if your engine supports write access. If write=1 you must supply the ENGDFV and ENGWRT routines. If write=0, the engine will be a read-only engine.

update
: Set this to 1 if your engine supports update access. If update=1, you must also set read and write to 1. If update is not set to 1, PROC FSEDIT will only be able to access the logical data set in browse mode.

random
: Set this to 1 if your engine can support random access. This is the preferable access method used by PROC FSEDIT and FSBROWSE when opening the logical data set. If random is not set to 1, PROC FSEDIT and FSBROWSE will fail the open, and will instead reopen without the random capability available.

options
: Currently unused.

assign
: Set this to 1 if your engine supplies the ENGASG and ENGDAS routines. ENGASG is called by the engine middle manager when a LIBNAME statement is seen, after all syntax for the LIBNAME statement is parsed. You supply ENGASG if you need to perform some kind of initialization, or if you support options on the LIBNAME statement. If ENGASG is supplied, you must also supply ENGDAS, which is invoked when the libname is deassigned (when a LIBNAME CLEAR is specified, or at the end of the SAS job). If assign=0, no ENGASG or ENGDAS routine is expected and none will be called.

note

Set this to 1 if your engine supports noting and pointing. The note field must be set to 1 if random=1. If note=1, you must supply the ENGNOT and ENGPNT routines.

ridlen
Set this to the length of the record id (rid) that you build in ENGNOT and use in ENGPNT. You must define a structure that includes the fields needed to describe the record id. Then set engstat.ridlen to the size of that structure.

ridandn
Set this to 1 if your engine supports rid-to-n and/or n-to-rid conversion. If ridandn=1, you must supply the ENGN2R and ENGR2N routines. Most engines should be able to support rid-to-n, since the observation number can be embedded in the rid. However, it is often more difficult to implement n-to-rid conversion. For example, if your file structure employs compression or varying-length records, you may not be able to predict the rid given only the observation number. In this case, you have the option of creating an array of rids for all observations (by making a complete pass of the data). However, this is usually undesirable unless the input file is small.

If you supply the ENGR2N routine but cannot handle n-to-rid conversions, create the ENGN2R routine and simply return XHENOSUP, indicating that the operation is not supported. If your engine cannot support a true n-to-rid capability, users cannot

- use observation numbers in PROC FSEDIT and FSBROWSE to point to arbitrary observations.
- use the POINT= option in the SET statement in the DATA step.

nopname
Set this to 1 if the LIBNAME statement does not require a physical name to be present. This would be the case if you are able to derive the necessary information from the libref (obtainable from the fileid structure at ENGOPN time or from the calling sequence of ENGASG), or from the LIBNAME statement options. If nopname is 0, the engine middle manager will flag LIBNAME statements as erroneous if the physical name is omitted.

engname
Set this to the 8-byte engine name. This will be the engine name that appears in the log after the LIBNAME statement is processed. Here is an example of an MVS log where the ENGXMPL engine is used:

```
1          LIBNAME ATEST ENGXMPL 'your.test.data(testdata)';

NOTE: Libref ATEST was successfully assigned as follows:
Engine:        ENGXMPL
Physical Name: your.test.data(testdata)
```

It is easier for the user to understand if the module name (which is specified in the U_ENG declaration) and the engine name filled into the engstat.engname field are the same name.

expansion
This section is not currently used.

# FILEID structure

**describes structure of data file**

The FILEID structure contains many fields that enable your engine routines to communicate with the engine middle manager. This structure describes the data set your engine processes.

## Declaration

```
struct FILEID {
double          crdate;
double          modate;
ENTSTR          engmmptr;    /* anchor ptr for engine middle manager  */
nameptr        *ppn;         /* ptr to list of namestr ptrs           */
nameptr         np;          /* ptr to namestr memory block           */
struct POSLNG  *psl;         /* npos/nlng for all variables           */
struct X_INFSTR *x_infstr;   /* options info structure                */
char           *physname;    /* ptr to physical name                  */
SPECPTR         specfid;     /* ptr to user expansion area            */
char           *poolid;      /* pid for allocations/frees             */
char           *errmsg;      /* ptr to W_ESYSER error message         */
char           *label;       /* ptr to label (NULL if no label)       */
char           *currec;      /* current record pointer                */
long            num_prec;    /* number of physical records            */
long            num_lrec;    /* number of logical records             */
long            num_vars;    /* number of variables in data set       */
long            max_rc;      /* maximum allowable return code         */
long            rec_len;     /* length of input observation buffer    */
long            opnmode;     /* open mode                             */
short           lablen;      /* length of data set label              */
short           errmsgl;     /* length of W_ESYSER error message      */
char            libname[9];  /* libname for opens (null-terminated)   */
char            memname[9];/* memname for opens (null-terminated)   */
};
```

## Description

Each field of this structure is now discussed separately.

crdate
: is the creation date for the logical SAS data set created by the engine. Your ENGOPN routine must set this date to a valid SAS datetime value. For input data sets, this value can be either the current datetime value or the creation date of the data file your engine is reading. For output logical data sets, you should use the current datetime value. This creation date appears in PROC CONTENTS output.

modate
: is the date when the logical SAS data set was last modified. Your ENGOPN routine must set this date to a valid SAS datetime value. If your engine supports update mode, set this value to the current datetime value when the ENGOPN routine is called. For all

other modes, set this date equal to the `crdate` value.

engmmptr
is the anchor pointer for the engine middle manager. Do not modify this pointer or change any data pointed to by it.

ppn
points to a list of NAMESTR pointers. The engine middle manager allocates the list of NAMESTR pointers before calling ENGNAM (for input data sets) or ENGDFV (for output data sets). In your ENGNAM or ENGDFV routine you must fill in the NAMESTRs at the locations indicated in the pointer list. The number of NAMESTRs is determined by the variable count in the num_vars field of the fileid structure (see page 64).

np
points to the block of memory in which the NAMESTR structures will be stored. You can set this pointer to memory that you provide, or you can leave it NULL and the engine middle manager will allocate the memory. If you set np to a value, your engine will be responsible for freeing the memory.

psl
points to an array of POSLNG structures.

x_infstr
Currently unused.

physname
is a pointer to the physical name specified by the user on the LIBNAME statement. This is set by the engine middle manager before it calls the ENGOPN routine. The physical name string is null-terminated. If the first byte of the string is a null terminator, this means that the user did not specify a physical name in the LIBNAME statement. This condition can occur only if you set the `nopname` flag in the ENGSTAT structure to 1 (see page 61). In this case, your ENGOPN routine must determine how to handle the LIBNAME statement that does not include a physical name.

If the `nopname` flag was not set to 1, the engine middle manager rejects a LIBNAME statement when the physical name is omitted.

specfid
is a pointer reserved for your use. If your engine has special needs that require an anchor pointer of some kind, use this pointer. This pointer is not referenced in any way by the engine middle manager, so you have complete control over it.

You will need to #define SPECPTR to be the appropriate kind of pointer. If you do not need an anchor pointer, include this statement in your program:

```
#define SPECPTR ptr
```

If you do need an anchor pointer, define a structure (called for example, SPECFID) and specify the following:

```
struct SPECFID {
...
};
#define SPECPTR struct SPECFID *
```

poolid
is a pool pointer that the engine middle manager sets before calling any of your engine routines. Always use this pool pointer and call the SAS_XMALLOC routine to allocate

## FILEID structure *continued*

*(poolid continued)*

space. Use SAS_XMFREE to free space. Do not use SAS_XMEMEX, SAS_XMEMGET, SAS_XMEMZER, or SAS_XMEMFRE because you cannot pass a pool id to these routines.

errmsg
: points to an error message that you build. If any of your ENG*xxx* routines needs to signal an error condition, set this pointer to point to an appropriate error message and set `errmsgl` to the length of the message. In addition, return the W_ESYSER return code from the ENG*xxx* routine. The engine middle manager references this pointer to print the message on the log.

label
: is a pointer to the logical data set label. Your ENGOPN routine must set this pointer. If there is no such label, set this pointer to NULL. If there is a label, be sure that the pointer points to memory that will not be freed or overwritten before the engine middle manager can reference it (which would be the case with I/O buffers). In addition, set the `lablen` field the length of the label.

currec
: is the pointer to the observation buffer for the current observation. The buffer is allocated by the engine middle manager based on the `rec_len` value (explained below). The observation buffer is filled in by the ENGRED or ENGDFV routine. If your engine needs to reset the value of this pointer to point to a different buffer area, this is possible. For example, if you are simply writing the contents of a buffer created by reading a SAS data set, you can set the `currec` to point to that buffer instead of moving the data to the observation buffer pointed to by `currec`.

num_prec
: is the number of observations in the data set. Set this number in your ENGOPN routine. If you can't determine the number of observations when you open the data set, set this value to -1. The number you supply here is the value that the SAS System returns to procedures, such as PROC CONTENTS, that inquire about the number of observations in the data set.

num_lrecl
: Currently unused.

num_vars
: is the number of variables in the logical data set. Your ENGOPN routine sets this value for an input or update data set. The engine middle manager uses this number to compute the amount of memory needed for the NAMESTRs, the NAMESTR pointer array, and the POSLNG array. This number is also the value that the SAS System returns to procedures, such as PROC CONTENTS, that inquire about the number of variables in the data set.

max_rc
: is the maximum allowable return code from various I/O operations. Set this value to zero.

rec_len
: is the record length of the observation buffer. Your ENGNAM routine sets this value

for input data sets. The ENGDFV routine indirectly sets this value via its passed parameter, `prec_len`.

opnmode
is the open mode for the data set. This value is set by the engine middle manager (based on the `read, write` and `update` fields of the ENGSTAT structure) before it calls ENGOPN. The possible XO_*mode* values are:

- XO_INPUT for input
- XO_OUTPUT for output
- XO_UPDATE for update

Always use these XO_*mode* symbols when referring to the modes.

lablen
is the length of the label for the logical data set. Set this field to zero if there is no label. If this field is nonzero, be sure the `label` pointer points to a label of the proper number of bytes.

errmsgl
is the length of the error message pointed to by `errmsg`. If any ENG*xxx* routine encounters an error condition and builds an error message, the routine must set this field to the length of the error message and set `errmsg` to the address of the error message.

libname
is the libref specified by the user on the LIBNAME statement. The engine middle manager sets the libname field before calling ENGOPN. This is a null-terminated field.

memname
is the data set name specified by the user. The engine middle manager sets this field before calling ENGOPN. This is a null-terminated field.

This field can be updated by ENGOPN if necessary. A blank `memname` can be passed to ENGOPN if the member name is unknown. This is used in PROC CONTENTS situations, or in cases where there can only be one member, and its name is unknown or irrelevant. (Note that the SPSS, BMDP, and OSIRIS engines use _FIRST_ in such cases).

## POSLNG Structure

**positions and lengths of variables**

The engine middle manager allocates the POSLNG array so that your routines can use it to save the position and length of the variables that are to be read from or placed into the `currec` buffer (see page 64). You do not have to use this structure if the data base management system with which you interface provides another mechanism for positioning to variables within a record. The engine middle manager does not reference the structure array in any way except to allocate and free it. The information for these structures is copied from the NAMESTR structures. Refer to "NAMESTR" on page 278 of *SAS/TOOLKIT Software: Usage and Reference, Version 6, First Edition* for more information on the NAMESTR structure.

## POSLNG Structure *continued*

### Declaration

```
/*---npos/nlng save structure--------------------*/
struct POSLNG {
long   pos;              /* original npos                        */
short  lng;              /* original nlng                        */
};
```

### Description

The fields of this structure are described here:

pos
: is the position of the variable in the observation buffer. Set this value to the same value as the `npos` field of the NAMESTR structure for the variable.

lng
: is the length of the variable in the observation buffer. Set this value to the same value as the `nlng` field of the NAMESTR structure for the variable.

## XONLIST Structure

**names and types of variables**

The engine middle manager allocates the XONLIST structures so that your ENGNAM routine can use it to store the variable names and the variable types. These values are a subset of those that appear in the NAMESTR. These values also occur in the XONLIST structure because they may be used by the engine supervisor and the engine middle manager after the NAMESTRs are freed. Refer to "NAMESTR" on page 278 of *SAS/TOOLKIT Software: Usage and Reference, Version 6, First Edition* for more information on the NAMESTR structure.

### Declaration

```
struct XONLIST
{
char8    name;     /* variable name     */
char     type;     /* variable type     */
/*  1=num 2=char     */
};
typedef struct XONLIST *xonlptr;
```

### Description

The fields of this structure are described here:

name
: is the variable name. Set this value to the same value as the `nname` field of the NAMESTR structure for the variable.

type
: is the type of the variable. Set this value to the same value as the `ntype` field of the NAMESTR structure for the variable.

## ENG Routine Specifications

This section documents the routines that you must develop for a user-written engine. Only the following routines are required:

- ENGOPN is required for all engines.
- ENGNAM, ENGRED, and ENGPNT are required only if you are reading records from the file.
- ENGDFV and ENGWRT are required only if you are writing records to the file.
- ENGCLS is required for all engines.
- ENGTRM is required for all engines.

The other routines described in this section are needed only for special situations.

## Conventions for Using Parameters in Routines

Each routine described in this chapter has a section labeled "Declarations" that contains a table of variables you need to declare when you define the routine. The format of these tables is as follows:

- The Type column contains a C data type for each variable used by the routine. Note that the data types can be any of the native C types or the defined types for SAS/TOOLKIT software. Refer to Table 10.1 in *SAS/TOOLKIT Software: Usage and Reference, Version 6, First Edition* for a complete list of the defined types for SAS/TOOLKIT software.
- The Variable column lists each variable used by the routine. Note that in many cases a variable is declared as a pointer to another data type. You do not have to create a pointer variable and store the address of the other data. Instead, you can simply declare the data and use the address operator, the ampersand (&), in the call.
- The Use column of the table shows how the variable is used by the routine. The following values can appear in this column of the table:

  input
  : means that the engine middle manager passes this value to your routine.

  output
  : means that your routine passes the value back to the engine middle manager. Note that because the C language does not permit a function to actually change the value of a parameter, you actually pass the address of

a variable. The contents of the variable are changed, but the address remains the same. Therefore, the output is actually stored at the address of the variable. Keep in mind that the pointer listed in the table is not the actual output; the value pointed to is the output value.

returned — is the function return value (or the *l-value* in C).

- The Description column briefly describes the variable. Look for more details on how to use the variable in the "Description" section that follows the table.

# ENGASG Routine

**perform initialization for engine**

required if engstat.assign=1

## Usage

```
rc = ENGASG(&libhandl,libname,&physname,&namelen,n_nopts,
            p_nnames,p_nvals,n_copts,p_cnames,p_cvals);
```

## Declarations

| Type | Variable | Use | Description |
|---|---|---|---|
| ptr | libhandl | output | anchor pointer to be set by ENGASG |
| ptr | libname | input | pointer to 8-byte libname given in LIBNAME statement |
| ptr | physname | input/output | pointer to physical name given in LIBNAME statement |
| int | namelen | input/output | length of physical name |
| int | n_nopts | input | number of numeric options given |
| ptr* | p_nnames | input | pointer to list of pointers which point to null-terminated numeric option names |
| dblptr | p_nvals | input | pointer to list of doubles corresponding to the numeric values given |
| int | n_copts | input | number of character options given |
| ptr* | p_cnames | input | pointer to list of pointers which point to null-terminated character option names |
| ptr* | p_cvals | input | pointer to list of pointers which point to null-terminated character values |
| rctype | rc | returned | return codes:<br>SUCCESS<br>W_ESYSER (ensure errmsg and errmsgl are set) |

### Description

If your engine permits options on the LIBNAME statement or if you do not require a physical file name on the LIBNAME statement, you must define the ENGASG routine. The engine middle manager calls the ENGASG routine after the LIBNAME statement is parsed. The engine middle manager passes ENGASG all the information provided by the user on the LIBNAME statement.

If you want to permit your users to omit the physical filename on the LIBNAME statement, you must set `engstat.nopname` to 1. Then, your ENGASG can create a new physical name and set the `physname` pointer to point to the new name. Do not move data into the location referenced by the original `physname` pointer; instead set the pointer to the new location. You should also update the `namelen` value to the appropriate length.

If your engine permits options, your ENGASG routine must process the options. The engine middle manager passes the options to your ENGASG routine. The options are separated into arrays of numeric and character names and values. For each type of option, the names of the options are given in one array (either p_cnames or p_nnames), and the values are given in a second array (either p_cvals or p_nvals). For example, assume the user specifies this LIBNAME statement:

```
libname myref youreng 'abc.dat' mynum=5 mystg='hello' myname=john;
```

When the engine middle manager calls ENGASG, the following information about options is passed to your routine:

- n_nopts=1
- n_copts=2
- p_nnames[0]is a pointer to "MYNUM" (the character string MYNUM followed by a null terminator)
- p_nvals[0]=5
- p_cnames [0]is a pointer to "MYSTG"
- p_cnames[1]is a pointer to "MYNAME"
- p_cvals[0]is a pointer to "HELLO"
- p_cvals[1]is a pointer to "JOHN".

It is up to ENGASG to decide if the options and their values are valid.

**Note:** Since the `errmsg` and `errmsgl` fields of the FILEID structure are not available at the time that ENGASG is invoked, you must print a message using the SAS_XPSLOG routine if you need to produce error messages explaining the failure of the ENGASG routine.

## ENGCLS Routine

**close file used as input to engine**

required

# ENGCLS Routine *continued*

## Usage

```
rc = ENGCLS(fid,disp,libmode);
```

## Declarations

| Type | Variable | Use | Description |
|---|---|---|---|
| fidptr | fid | input | fileid pointer passed in by the engine middle manager |
| int | disp | | disposition |
| int | libmode | input | indicates how the file should be closed:<br>NORMCLOS - normal close<br>LIBCLOSN - a close corresponding to an ENGOPN for LIBOPEN1 or LIBOPENN<br>LIBCLOSL - the last requested close |
| rctype | rc | returned | return code:<br>SUCCESS<br>XHENOMEM - insufficient memory<br>W_ESYSER (ensure errmsg and errmsgl are set) |

## Description

You must define the ENGCLS routine for your engine. This routine closes the data file for either input or output operations. The ENGCLS routine must perform the following:

- close the member, and also the physical file if necessary
- perform any additional closing processing

If you defined the ENGOPN routine to be able to respond to this type of request from PROC CONTENTS:

```
proc contents data=libref._all_;
```

you need to also provide special processing in the ENGCLS routine. When the engine middle manager passes a `libmode` of LIBCLOSN, you close the member that was most recently opened by ENGOPN. When the engine middle manager passes LIBCLOSL, you close the entire library.

Any memory allocated by ENGOPN or other routines should be freed by ENGCLS if NORMCLOS or LIBCLOSL is the `libmode`.

# ENGDAS Routine

**deassign library**

required if engstat.assign=1

## Usage

```
rc = ENGDAS(libhandl);
```

## Declarations

| Type | Variable | Use | Description |
|---|---|---|---|
| ptr | libhandl | input | the anchor pointer set by ENGASG |
| rctype | rc | returned | return code:<br>SUCCESS<br>XHENOMEM - insufficient memory<br>W_ESYSER (ensure errmsg and errmsgl are set) |

## Description

If you define the ENGASG routine, you must also define the ENGDAS routine. The engine middle manager calls ENGDAS to deassign a library. A library is deassigned when the user

- issues a LIBNAME statement with the CLEAR option
- specifies a new LIBNAME statement with a libref currently in use
- ends the SAS job while the libref is still active.

# ENGDFV Routine

**describe output variables**

required if engstat.write=1 or engstat.update=1

## Usage

```
rc = ENGDFV(fid,prec_len)
```

# ENGDFV Routine *continued*

## Declarations

| Type | Variable | Use | Description |
|---|---|---|---|
| fidptr | fid | input | fileid pointer passed in by the engine middle manager |
| long* | prec_len | output | size of the output observation buffer. |
| rctype | rc | returned | return code:<br>SUCCESS<br>XHENOMEM - insufficient memory<br>W_ESYSER (ensure errmsg and errmsgl are set) |

## Description

If your engine writes to a file, you must define the ENGDFV routine. The ENGDFV routine writes variable descriptor records to the output file in the appropriate form. You can use fid→ppn to access an array of NAMESTR pointers. These NAMESTRs describe the variables stored in the observation buffer. For each variable, your ENGDFV routine must

- write the information from the NAMESTR structure to the output file in whatever manner is appropriate.
- set `prec_len` to the output buffer record length. Setting this value updates the `rec_len` field of the FILEID structure. Note that the pointer to the output buffer will be passed to the ENGWRT routine. The buffer contains the data that ENGWRT converts to the proper format to be written out.

  In addition, the ENGDFV routine might need to

- put the `npos` and `nlng` into the `pos` and `lng` fields of the POSLNG array. Make the `lng` value negative for a character variable. This step is necessary only if the data base management system accessed by your engine does not provide some mechanism for positioning to a specific variable within a record.
- write header or trailer data.

# ENGNAM Routine

**store information about variables**

required if engstat.read=1

## Usage

```
rc = ENGNAM(fid,pxonl);
```

## Declarations

| Type | Variable | Use | Description |
|---|---|---|---|
| fidptr | fid | input | fileid pointer passed in by the engine middle manager |
| xonlptr | pxonl | | <verify> |
| rctype | rc | returned | return code:<br>SUCCESS<br>XHENOMEM - insufficient memory<br>W_ESYSER (ensure errmsg and errmsgl are set) |

## Description

If your engine reads a file, you must define the ENGNAM routine. The ENGNAM routine fills in the NAMESTR structures for all input variables. The engine middle manager allocates the NAMESTR array before calling ENGNAM. In addition, the ENGNAM routine fills in the XONLIST array with the variable names and types and may also fill in the array of POSLNG structures.

For convenience, the NAMESTR structure is listed here. Refer to page 278 of the *SAS/TOOLKIT Software: Usage and Reference, Version 6, First Edition* for more information on the NAMESTR structure.

```
struct NAMESTR {
/*---fields from the data set---------------*/
long   ntype;      /* type of variable 1=num 2=char       */
long   nlng;       /* length of variable                  */
char  *nname;      /* name of variable                    */
long   namelen;    /* length of variable name             */
char  *nlabel;     /* ptr to label                        */
long   nlablen;    /* length of label                     */
char   nform[8];   /* format name                         */
long   nfl;        /* format width                        */
long   nfd;        /* format number of decimals           */
long   nfj;        /* 0=left justification, 1=right just  */
char   niform[8];  /* informat name                       */
long   nifl;       /* informat width                      */
long   nifd;       /* informat number of decimals         */
long   npos;       /* position of value in observation    */
long   nvar0;      /* number of variable on the file      */
/*---not from data set, derived fields via SAS_XFFILE--*/
long   nflen;      /* format width                        */
long   nfcode;     /* format code for variable            */
long   nifcode;    /* informat code for variable          */
};
```

## ENGNAM Routine *continued*

For each variable, your ENGNAM routine must

- fill in the values of the `ntype` through `nvar0` fields of the NAMESTR structure. The remaining three fields can also be set if appropriate. All fields are initialized to zero before ENGNAM is called, so all fields that can appropriately be zero need not be set.
- put the `nname` and `ntype` into the `name` and `type` fields of the XONLIST array.
- set fid->rec_len to the length of the input buffer that will be filled in by ENGRED. Note that this is the resulting buffer length. This buffer is better known as the program data vector, or PDV.

In addition, your ENGNAM routine might need to

- copy the values of `npos` and `nlng` into the `pos` and `lng` fields of the POSLNG array. Set the `lng` value to a negative number for a character variable. This step is necessary only if the data base management system accessed by your engine does not provide some mechanism for positioning to a specific variable within a record.
- create a special record id (rid) for the beginning of the file. This rid corresponds to the RIDBOD logical value. This rid will be needed as soon as the engine attempts to read from the file. You may need to determine the proper rid by performing some kind of read against your data file.

## ENGNOT Routine

**note position of record currently being read**

required if engstat.note=1

### Usage

```
rc = ENGNOT(fid,ridp);
```

### Declarations

| Type | Variable | Use | Description |
|---|---|---|---|
| fidptr | fid | input | fileid pointer passed in by the engine middle manager |
| char | *ridp | input | pointer to rid to fill in |
| rctype | rc | returned | return code:<br>SUCCESS<br>XHENOMEM - insufficient memory<br>XHENOSUP - noting not supported<br>W_ESYSER (ensure errmsg and errmsgl are set) |

### Description

If your engine supports direct access or permits the users to return to a previously read record, you must define the ENGNOT routine. The ENGNOT routine fills in an area with the rid of the most recently read observation. This rid is built by ENGRED.

Many SAS applications (such as procedures) need the capability of repositioning within a data set. Repositioning involves recording information about the position of a given observation, then returning to that observation at a later point after providing the positioning information. The positioning information resides in a record id (or rid). The positioning information is engine-dependent, and there are no qualifications placed on it by the engine middle manager. The engine simply supplies the length of the rid, then manipulates the rid by means of the ENGNOT and ENGPNT routines.

The ENGNOT routine records information about position of a record. The engine middle manager calls ENGNOT with a pointer to an area to contain the rid. ENGNOT fills in this rid with the appropriate information for the current observation. Then, when the application needs to reposition to the observation, the engine middle manager calls the ENGPNT routine, using the rid filled by a previous ENGNOT. The ENGPNT routine repositions to the desired observation so that the next ENGRED or ENGWRT routine reads from or writes to that observation.

The contents of the rid are engine-dependent. The only requirement is that the first byte of the rid must be reserved. This is because there are special rids understood by the engine supervisor that must also be recognized by ENGPNT. These special rids are identified by a lowercase letter as the first byte of the rid. The rid defined for your engine will contain the value 0x00 in the first character.

## ENGN2R Routine

**convert an observation number to a rid**

required if engstat.ridandn=1

### Usage

```
rc = ENGN2R(fid,recno,rid);
```

### Declarations

| Type | Variable | Use | Description |
|---|---|---|---|
| fidptr | fid | input | fileid pointer passed in by the engine middle manager |
| long | recno | input | observation number |
| ptr | rid | output | pointer to rid to fill in |
| rctype | rc | returned | return code:<br>SUCCESS<br>XHENOMEM - insufficient memory<br>XHENOSUP - n-to-rid not supported<br>W_ESYSER (ensure errmsg and errmsgl are set) |

## ENGN2R Routine *continued*

### Description

If your engine supports random access to the data file (for example, using the POINT= option of the SET statement or using observation numbers to position while using PROC FSEDIT or FSBROWSE), you must define the ENGN2R routine.

The ENGN2R routine converts an observation number into a rid. You are most likely to be able to implement this capability if the observation number is easily computed from the record information that you use for the rid. That is, this capability is more feasible if the data file your engine reads stores information about the location of its records in a form that can be easily equated to an observation number. Another possible implementation would be to read the entire file sequentially, storing the observation numbers and rids for subsequent random access.

If your engine cannot support n-to-rid conversion, you can return the XHENOSUP. Note that if your engine can support rid-to-n, but not n-to-rid, you must still set engstat.ridandn=1 and supply an ENGN2R routine, even if it simply returns XHENOSUP.

## ENGOPN Routine

**opens file as a SAS data set**

required

### Usage

```
rc = ENGOPN(fid,libmode)
```

### Declarations

| Type | Variable | Use | Description |
|---|---|---|---|
| fidptr | fid | input | fileid pointer passed in by the engine middle manager |
| int | libmode | input | Values are:<br>NORMOPEN - normal open<br>LIBOPEN1 - first input open for what will be a PROC CONTENTS series of opens<br>LIBOPENN - subsequent opens of a PROC CONTENTS series<br>LIBOPENY - second open request for a member, which can usually just be ignored |

| Type | Variable | Use | Description |
|---|---|---|---|
| rctype | rc | returned | return code:<br>SUCCESS<br>XHENOMEM - insufficient memory<br>W_ESYSER (ensure errmsg and errmsgl are set)<br>XHENOLIB - library can't be opened<br>XHENOMBR - member can't be opened<br>X_ENOMEM - insufficient memory<br>X_WEOF - for PROC CONTENTS-style opens |

## Description

If your engine reads from a data file, you must define the ENGOPN routine. The ENGOPN routine opens the data file as a logical SAS data set. You can perform whatever action is necessary to open the entity. The engine middle manager completes the `libname`, `memname`, and `opnmode` values of the FILEID structure before calling your ENGOPN routine so your routine will know what file to open and what mode to use.

If your engine permits this type of request from PROC CONTENTS:

```
proc contents data=libref._all_;
```

you need to also provide special processing in the ENGOPN routine. When the engine middle manager passes a `libmode` of LIBOPEN1 or LIBOPENN, you open the requested member of the library so that PROC CONTENTS can access the name of the member. PROC CONTENTS may also call ENGOPN with a `libmode` of LIBOPENY. This call can be ignored if the descriptor information that you just accessed when you opened the member is followed immediately by the data record. However, if the descriptor information is stored separately from the data, when your routine is called with a `libmode` of LIBOPENY, you should open the data record.

You ENGOPN routine should

- open the member with the mode requested in the `opnmode` field of the FILEID structure. If the open is unsuccessful, return the appropriate return code. If ENGOPN is called with a `libmode` of LIBOPENN and you have reached the end of the input data file, you can return a code of X_WEOF.
- set the fid->memname when the open mode is input. This is necessary in the case of a directory read (for PROC COPY or PROC CONTENTS) to determine the next member name. This field is also used if there can be a default member.
- set the number of observations in the fid->num_prec field when the open mode is input. Set this to -1 if the number of observations is unknown.
- set the number of variables in the file in fid->num_vars when the open mode is input.
- set the maximum permissible return code in `fid->max_rc`. This value is usually 0. If your application prefers to receive non-zero I/O return codes and issue messages or otherwise handle those errors, you can set the `max_rc` to a non-zero value.
- set the fid->crdate and fid->modate to the creation and modification dates.
- set the label pointer for the data set label.

# ENGPNT Routine

**point to previously noted record**

required if engstat.read=1

## Usage

```
rc = ENGPNT(fid,ridp);
```

## Declarations

| Type | Variable | Use | Description |
|---|---|---|---|
| fidptr | fid | input | fileid pointer passed in by the engine middle manager |
| char | *ridp | | pointer to rid to use in positioning |
| rctype | rc | returned | return code:<br>SUCCESS<br>XHENOMEM - insufficient memory<br>XHEBDCSQ - unacceptable rid<br>XHENOSUP - pointing not supported<br>W_ESYSER (ensure errmsg and errmsgl are set) |

## Description

The ENGPNT routine repositions in the file based on a rid value. This routine is required if your engine permits read access. For all engines except output-only, you must, at the very least, define this routine so it can position to the beginning of the file, indicated by `ridp` pointing to the value RIDBOD.

If your engine supports random access, you must also define the ENGNOT routine and a rid that enables you to directly access records in the file you are reading. The ENGNOT routine notes the current location of a record so you can return to that record. It stores the location in a record id or rid. The rid is a structure you define to describe the location of a record in the file your engine reads. You must reserve the first byte of the rid. The engine middle manager sets this byte to 0x00 when it passes ENGPNT a rid to a record in the data file. The engine middle manager also uses this first byte to indicate special rid values. Some of these special rid values must also be handled by your routine if you support random access:

- RIDCURR, the current observation
- RIDNEXT, the next observation
- RIDPREV, the previous observation

After testing for these rid values and the ones that are specific to your engine, you can set an error flag for any other rid values and return XHEBDCSQ.

Refer to “ENGNOT Routine” on page 62 for more information on how to enable direct access to the data file.

# ENGRED Routine

**read record into observation buffer**

required if engstat.read=1

## Usage

```
rc = ENGRED(fid,rptr);
```

## Declarations

| Type | Variable | Use | Description |
|---|---|---|---|
| fidptr | fid | input | fileid pointer passed in by the engine middle manager |
| char | **rptr | output | pointer to input buffer (returned) |
| rctype | rc | returned | return code:<br>SUCCESS<br>XHENOMEM - insufficient memory<br>X_WEOF - end-of-file<br>W_ESYSER (ensure errmsg and errmsgl are set) |

## Description

If your engine reads from a data file, you must create the ENGRED routine. The ENGRED routine fills in the observation buffer using the POSLNG array or some other mechanism provided by the data base management system your engine accesses. You must have also defined the ENGNAM routine to describe the variables that ENGRED moves from the data file to the observation buffer.

ENGRED should

- read records from the input data file and store the data in the input buffer according to the positions and lengths stored in the POSLNG array or some other mechanism for locating variables
- return X_WEOF when there are no more observations
- set `rptr` to the address of the input buffer

In addition, if your engine supports the ability to return to previously read records, ENGRED must note the rid of the current record.

# ENGR2N Routine

**convert a rid to an observation number**

required if engstat.ridandn=1

## Usage

```
rc = ENGR2N(fid,rid,&recno);
```

## Declarations

| Type | Variable | Use | Description |
|---|---|---|---|
| fidptr | fid | input | fileid pointer passed in by the engine middle manager |
| ptr | rid | input | pointer to rid |
| long | recno | output | observation number (returned) |
| rctype | rc | returned | return code:<br>SUCCESS<br>XHENOMEM - insufficient memory<br>XHENOSUP - rid-to-n not supported<br>W_ESYSER (ensure errmsg and errmsgl are set) |

## Description

You must create the ENGR2N routine if your engine supports random access. The engine middle manager calls the ENGR2N routine when it needs to convert a rid into an observation number. The simplest way to keep this information is to store the observation number as part of the rid.

ENGR2N is needed by procedures that print a number to be associated with the observation, as PROC PRINT does.

# ENGTRM Routine

**termination routine**

required

### Usage

```
ENGTRM();
```

### Description

You must create the ENGTRM routine. The ENGTRM routine terminates the engine environment. Anything the engine must do to shut down the environment should be included in this routine.

## ENGWRT Routine

**write record from observation buffer**

required if engstat.write=1 or engstat.update=1

### Usage

```
rc = ENGWRT(fid,rptr,status);
```

### Declarations

| Type | Variable | Use | Description |
|---|---|---|---|
| fidptr | fid | input | fileid pointer passed in by the engine middle manager |
| char | *rptr | input | pointer to output observation buffer |
| int | status | | 0=output observation<br>1=update observation |
| rctype | rc | returned | return code:<br>SUCCESS<br>XHENOMEM - insufficient memory<br>W_ESYSER (ensure errmsg and errmsgl are set) |

### Description

If your engine writes to a data file, you must create the ENGWRT routine. The ENGWRT routine converts an observation buffer into the proper output record format and writes the record.

ENGWRT should

- convert all values to the output format and write the record
- increment the `fid->num_prec` value by 1 to indicate another observation has been written

# Chapter 5 Writing Special DATA Step Functions



## Introduction

There are four new routines in Release 6.08 of SAS/TOOLKIT software to allow DATA step function writers to have access to the actual DATA step internal locations. Normally, function arguments are updated when passed to the function. However, some applications need to be able to directly update data locations without having to be passed the values to update.

## The SAS_DSS Routines

Direct updating is supported through the SAS_DSS routines. These routines operate on "symbol tables", which are the entities into which the DATA step stores information about each variable defined in the DATA step.

Currently, the SAS_DSS routines are supported for C, PL/I, and IBM Assembly Language for SAS/TOOLKIT software. Note that the SAS_DSS routines will work only when called from a DATA step function. Do not attempt to use these routines in an SCL program or in a SAS/IML user-written function.

Here is the C symbol table structure definition (found in the uwproc.h #include file):

```
struct SYMINFO
{

char                type;       /* Variable type          */
                                /*   1 - Numeric          */
                                /*   2 - Character        */

short               size;       /* Length of variable     */

ptr                 spelling;   /* Pointer to the name    */
short               spellen;    /* Length of the name     */

char40              slabel;     /* Label information      */

char8               formatn;    /* Format name            */
short               formatw;    /* Format width           */
short               formatd;    /* Format decimal         */
```

```
long                formatj;    /* Format justification   */
long                formatc;    /* Format code            */
ptr                 formatp;    /* Format data            */
int               (*ignore1)();
char8               iformatn;   /* Informat name          */
short               iformatw;   /* Informat width         */
short               iformatd;   /* Informat decimal       */
long                iformatj;   /* Informat justification */
long                iformatc;   /* Informat code          */
ptr                 iformatp;   /* Format data            */
int               (*ignore2)();

union
{
ptr             cloc;
double          *floc;
struct X_STRING {
short maxlen;   /* The allocated length             */
short curlen;   /* The current length               */
ptr   data;     /* The pointer to the string data   */
} *tloc;
} loc;                        /* Value storage location */

};
```

Note that the PL/I version of this structure is in the SYMINFO include file. The Assembler definitions are in the UWPROC include file.

What will be of most interest will be the loc union. The double pointer loc.floc will be the pointer to the variable if it is a numeric variable. The character pointer loc.tloc → data will be the pointer to the variable if it is a character variable. The value loc.tloc → curlen will be the length of the character data.

## SAS_DSSRSN Routine

The SAS_DSSRSN routine will obtain the total symbol table count.

```
i = SAS_DSSRSN();

where

i              total symbol table count
```

Call the SAS_DSSRSN routine once to determine the number of symbol table entries. You can then use this number to allocate an array of symbol tables that can be filled in by the SAS_DSSRSIF routine.

## SAS_DSSRSIF Routine

The SAS_DSSRSIF routine will obtain all symbol tables at one time.

```
SAS_DSSRSIF(syminfoptr,&n);

where

syminfoptr     ptr to SYMINFO array to fill in
n              number of SYMINFO elements available to be filled in
(value is updated by SAS_DSSRSIF)
```

The n value is modified by SAS_DSSRSIF to indicate the number of SYMINFO elements that were filled in. If the incoming n value is not the same as the outgoing n value, then there was some problem in filling in all elements.

Here is a code fragment that demonstrates how these two routines work together:

```
/*---get total symbol element count---*/
psyn = SAS_DSSRSN();

/*---allocate entire symbol table array---*/
psyptr = (struct SYMINFO *)SAS_XMEMEX(psyn * sizeof(struct SYMINFO));

/*---fill in this array with all variables' symbol elements---*/
SAS_DSSRSIF(psyptr,&psyn);
```

Your application may not need to obtain all symbol table elements. Instead, you may only need to obtain certain symbol table elements, based on the name of the variable. In this case, you can use the SAS_DSSRSI routine.

## SAS_DSSRSI Routine

The SAS_DSSRSI routine will determine if a specified variable does exist in the symbol tables.

```
rc = SAS_DSSRSI(varname,varnamel,&syminfo);

where

varname        a pointer to the variable to search for
varnamel       the length of the name
syminfo        a SYMINFO structure that will be filled in
rc             return code: 0=found non-zero=not found
```

If you have a select set of variables to search for, call the SAS_DSSRSI routine for each variable and extract the data location information from the symbol table returned by SAS_DSSRSI.

## SAS_DSSRSIT Routine

Your application may need to obtain information about many variables, based on their relative position, or by their prefix, or by any other criterion that doesn't allow you to create a specific list of variables. You can use the SAS_DSSRSIF routine to obtain the entire symbol table list, or you can use the SAS_DSSRSIT routine to obtain the symbol table elements one at a time, to avoid having to allocate all the symbol table elements at one time.

```
rc = SAS_DSSRSIT(&syminfo,&p);

where

syminfo        a SYMINFO structure that will be filled in
p              anchor pointer, set to NULL before first call
rc             return code: 0=variable found, 1=no more variables
```

# Sample Program

The sample source CDSFUNC contains an application using all four of the new SAS_DSS routines. You can examine that example to learn more about the functionality of the routines. Here is a listing of the CDSFUNC source included for convenience:

```
/*===================================================================*/
/* NAME:         cdsfunc                                             */
/* TYPE:         function                                            */
/* LANGUAGE:     C                                                   */
/* PURPOSE:      Example function package to demonstrate how to      */
/*               use the SAS_DSS routines                            */
/*-------------------------------------*/
/* This example demonstrates how the various SAS_DSS routines work.  */
/* We have 3 functions being defined in this example.                */
/*                                                                   */
/*=========================DSFUNC1===================================*/
/* The DSFUNC1 routine demonstrates how to use the SAS_DSSRSI routine*/
/* to obtain the symbol table for a specified variable.              */
/*                                                                   */
/*     CALL DSFUNC1(name,type,length,fmtname,fmtw,fmtd,infmtname,    */
/*             infmtw,infmtd,label);                                 */
/*                                                                   */
/* where                                                             */
/*                                                                   */
/* name            character expression indicating a variable name   */
/* type            (returned) type of the variable                   */
/* length          (returned) length of the variable                 */
/* fmtname         (returned) name of the format                     */
/* fmtw            (returned) width of the format                    */
/* fmtd            (returned) no. of decimals for the format         */
/* infmtname       (returned) name of the informat                   */
/* infmtw          (returned) width of the informat                  */
/* infmtd          (returned) no. of decimals for the informat       */
/* label           (returned) variable label                         */
```

```
/*                                                                  */
/* All arguments marked as 'returned' must be passed as variables.  */
/* However, any such argument can also be omitted via passing a     */
/* null argument. For example:                                      */
/*                                                                  */
/*      CALL DSFUNC1('X',,,,,,,,,label);                            */
/*                                                                  */
/* will obtain only the label for the variable X. Also, this call-  */
/* routine allows a variable number of arguments, so if you only    */
/* want the type and length:                                        */
/*                                                                  */
/*      CALL DSFUNC1('X',type,length);                              */
/*                                                                  */
/* Because of the data step variable typing restriction, you must   */
/* set the type for the character arguments before calling DSFUNC1. */
/* For example:                                                     */
/*                                                                  */
/*      DATA _NULL_;                                                */
/*         LENGTH LABEL $40;                                        */
/*         CALL DSFUNC1('X',,,,,,,,,LABEL);                         */
/*                                                                  */
/* If you omit the LENGTH statement, the DATA step will interpret   */
/* LABEL as a numeric variable and cause various warning messages.  */
/*                                                                  */
/* If the variable requested doesn't exist, an 'invalid argument to */
/* function' message will appear in the SAS log.                    */
/*                                                                  */
/*=========================DSFUNC2==================================*/
/* The DSFUNC2 routine demonstrates how to use the SAS_DSSRSIT      */
/* routine to loop through all the symbol table elements to find    */
/* variables of interest.                                           */
/*                                                                  */
/*     CALL DSFUNC2;                                                */
/*                                                                  */
/* There are no arguments to this call-routine. It will look for    */
/* the numeric variables A, B, C, D, and E in the symbol tables.    */
/* It will set the values to 0, 1, 2, 3, and 4, respectively. It    */
/* is OK if any or all of the variables are omitted or of the       */
/* wrong type; it will simply not set the value.                    */
/*                                                                  */
/*=========================DSFUNC3==================================*/
/* The DSFUNC3 routine demonstrates how to use the SAS_DSSRSN       */
/* and SAS_DSSRSIF routines to obtain all the symbol table elements */
/* at once in a local array and to subsequently use the symbol table */
/* to update data.                                                  */
/*                                                                  */
/*     CALL DSFUNC3;                                                */
/*                                                                  */
/* Like DSFUNC2, there are no arguments to this call-routine. It    */
/* will look for the numeric variables A, B, C, D, and E in the     */
/* symbol tables. It will use the current call counter and set      */
/* the variables A, B, C, D, and E to n, n+1, n+2, n+3, n+4, and    */
/* n+5, respectively, where n is the call counter. The "call        */
```

```
/* counter" is the number of times that DSFUNC3 has been called.     */
/* It is OK if any or all of the variables are omitted or of the     */
/* wrong type; it will simply not set the value.                     */
/*                                                                   */
/* Note that we don't perform the setup routine when request=2 is    */
/* called. This is because the variable locations have not yet been  */
/* determined by the time the request=2 call is made, so the loc.floc*/
/* fields in the symbol table elements are still NULL. We must wait  */
/* till the first DSFUNC3 invocation to call setup.                  */
/*-----------------------------------*/

/*--the one necessary #include file-----------------*/

#include "uwproc.h"

/*---number of variables (5 for A,B,C,D,E) for this example----*/
#define NVARS 5

/*---global elements initialized then later referred to-------*/
U_RWNSHR struct SYMINFO *psyptr;        /* symbol table array address*/
U_RWNSHR double *psyvptr[NVARS];        /* value location pointers   */
U_RWNSHR double dummyd;                 /* dummy double              */
U_RWNSHR long psyn;                     /* total symbols             */
U_RWNSHR long ptimes;                   /* count of DSFUNC3 calls    */
U_RWNSHR int isetup;                    /* indicates whether inited  */

/*---local routine prototype---------------------*/
void setup U_PARMS((void));
void IFFEXT U_PARMS((void));
int RTN1 U_PARMS((short*,short*,ptr*,double*));
int RTN2 U_PARMS((void));
int RTN3 U_PARMS((void));

/*-----------------------------------*/
/* The IFFMAI routine is required. It is called by the SAS supervisor*/
/* at least once for request=1 and exactly once for request 2.       */
/* Request 1 is to obtain the names and attributes of the functions. */
/* Request 2 is to obtain the addresses of the routines to be called.*/
/*-----------------------------------*/

ptr IFFMAI(request)
int *request;
{

/*-----------------------------------*/
/* With request 1, we call certain routines. The first routine we    */
/* call is UWPRCC. This routine is only necessary if you are going   */
/* to use SAS_ interface routines in your code. We do indeed in this */
/* example, so the UWPRCC call is necessary. Ensure a 0 is passed to */
/* it. The next call made is to FNCDFS. We pass this routine the     */
/* number of functions being defined. For each function, we provide  */
/* a FNCDFN call. This supplies the function number, function name,  */
/* the minimum and maximum number of arguments, and the return type  */
```

```
/* (1=numeric 2=character 0='call' routine). Following the FNCDFN    */
/* call, a call is made to FNCDFA as many times as there are minimum */
/* arguments to the function we're defining. FNCDFA specifies the    */
/* function number, the argument number, and the argument type (1    */
/* or 2 like above). The last call made is FNCDFE whose return code  */
/* is returned to the supervisor. For request 2, only the call to    */
/* FNCFNE needs to be called, and its return code is returned to the */
/* supervisor. Any additional initialization code that your appli-   */
/* cation may need should go in the request=2 code.                  */
/* The function rtn1 corresponds to the definitions for function 1   */
/* rtn2 for function 2, etc.                                         */
/*-----------------------------------*/

if (*request == 1) {
   UWPRCC(0);
   FNCDFS(3);

   /*---1. DSFUNC1:  from 2 to 10 args; 'call' routine-------*/
   FNCDFN(1,"DSFUNC1 ",2,10,0 + XFS_L);  /* XFS_L means nullargs OK */

   /*---argument types as appropriate-----------------*/
   FNCDFA(1,1,2);           /* var name        */
   FNCDFA(1,2,1);           /* var type        */
   FNCDFA(1,3,1);           /* var length      */
   FNCDFA(1,4,2);           /* format name     */
   FNCDFA(1,5,1);           /* format width    */
   FNCDFA(1,6,1);           /* format ndec     */
   FNCDFA(1,7,2);           /* informat name   */
   FNCDFA(1,8,1);           /* informat width  */
   FNCDFA(1,9,1);           /* informat ndec   */
   FNCDFA(1,10,2);          /* label           */

   /*---2. DSFUNC2:  no arguments; call routine-----------*/
   FNCDFN(2,"DSFUNC2 ",0,0,0);

   /*---3. DSFUNC3:  no arguments; call routine-----------*/
   FNCDFN(3,"DSFUNC3 ",0,0,0);

   return(FNCDFE());
   }
else if (*request == 2) {
   /*---indicate init needed with first DSFUNC3 call---*/
   isetup = 0;
   psyptr = NULL;
   return(FNCFNE());
   }
}

int RTN1(m1,c1,p1,type)
short *m1,*c1;
ptr *p1;
double *type;
{
```

```
struct SYMINFO syminfo;
short *cptr,maxlen;
ptr valuep;
int i,j,n,argtype;
char temp[8];

/*---copy over varname and upcase it---*/
SAS_ZSTRMOV(*p1,*c1,temp,8);
SAS_ZSTRUP(temp,8);

/*---verify the variable does exist---*/
if (SAS_DSSRSI(temp,SAS_ZSTRIP(temp,8),&syminfo))
   return(F_EIAF + 1);

/*---update type if argument given---*/
if (type != NULL)
   *type = syminfo.type;

/*---determine number of arguments specified (including nulls)---*/
FNCN(&n);

/*---loop through each argument---*/
for (i=3;i<=n;i++) {

   /*---obtain info on the argument---*/
   FNCARG(i,&argtype,&valuep,&cptr,&maxlen);

/*---skip processing if a null argument---*/
   if (argtype == 0)
      continue;

   /*---process based on argument number---*/
   switch(i) {
      case 3: /* length */
         *(double *)valuep = syminfo.size;
         break;
      case 4: /* format name */
         j = SAS_ZSTRIP(syminfo.formatn,8);
         *cptr = MIN(MIN(maxlen,8),j);
         memcpy(valuep,syminfo.formatn,*cptr);
         break;
      case 5: /* format width */
         *(double *)valuep = syminfo.formatw;
         break;
      case 6: /* format ndec */
         *(double *)valuep = syminfo.formatd;
         break;
      case 7: /* informat name */
         j = SAS_ZSTRIP(syminfo.iformatn,8);
         *cptr = MIN(MIN(maxlen,8),j);
         memcpy(valuep,syminfo.iformatn,*cptr);
         break;
      case 8: /* informat width */
```

```
         *(double *)valuep = syminfo.iformatw;
         break;
      case 9: /* informat ndec */
         *(double *)valuep = syminfo.iformatd;
         break;
      case 10: /* label */
         j = SAS_ZSTRIP(syminfo.slabel,40);
         *cptr = MIN(MIN(maxlen,40),j);
         memcpy(valuep,syminfo.slabel,*cptr);
         break;
      }
   }
return(F_OK);
}

int RTN2()
{
struct SYMINFO syminfo;
ptr p;
int i;
char temp[9];
U_RSHR static char names[1][46] =
"A        B        C        D        E        ";
p = NULL; /* for first SAS_DSSRSIT call */

/*---loop through symbol tables and search for variables---*/
while(SAS_DSSRSIT(&syminfo,&p) == 0) {

   /*---must be a numeric variable---*/
   if (syminfo.type != 1)
      continue;

   /*---copy variable name to 9-byte blank padded field---*/
   SAS_ZSTRMOV(syminfo.spelling,syminfo.spellen,temp,9);

   /*---search for it in our name list---*/
   i = SAS_ZSTRNDX(names[0],9*NVARS,temp,9);

   /*---if found, save proper value of 0 through 4 in variable---*/
   if (i >= 0)
      *syminfo.loc.floc = i / 9;
   }
return(F_OK);
}

int RTN3()
{
int i;

/*---perform initialization for first DSFUNC3 call---*/
if (!isetup)
setup();
```

```
/*---increment call counter---*/
ptimes++;

/*---loop through each variable and update its value---*/
for (i=0;i<NVARS;i++) {
   *psyvptr[i] = ptimes + i;
   }
return(F_OK);
}

/*---required termination routine---*/
void IFFEXT()
{
/*---free memory (possibly) allocated by setup---*/
SAS_XMEMFRE((ptr)psyptr);
}

/*---acquire entire symbol table array---*/
void setup() {
int i,j;
char temp[9];
struct SYMINFO *p;
U_RSHR static char names[1][46] =
"A        B        C        D        E        ";

/*---indicate setup done---*/
isetup = 1;

/*---get total symbol element count---*/
psyn = SAS_DSSRSN();

/*---allocate entire symbol table array---*/
p = psyptr = (struct SYMINFO *)
   SAS_XMEMEX(psyn * sizeof(struct SYMINFO));

/*---fill in this array with all variables' symbol elements---*/
SAS_DSSRSIF(psyptr,&psyn);

/*---initialize our pointer list to the dummy double---*/
for (i=0;i<NVARS;i++)
   psyvptr[i] = &dummyd;

/*---loop through symbol table elements for A-E variables---*/
for (i=0;i<psyn;i++,p++) {

   /*---must be numeric---*/
   if (p->type != 1)
      continue;

   /*---copy to local variable and blank pad to 9 characters---*/
   SAS_ZSTRMOV(p->spelling,p->spellen,temp,9);

   /*---search for variable in our name list---*/
```

```
      j = SAS_ZSTRNDX(names[0],9*NVARS,temp,9);

      /*---if found, save variable location in psyvptr array---*/
      if (j >= 0)
         psyvptr[j / 9] = p->loc.floc;
      }

   /*---initialize call counter to 0---*/
   ptimes = 0;
   }
```

The PL/I sample is located in the PDSFUNCx sources (PDSFUNC1-PDSFUNC6). The Assembler sample is in ADSFUNC.

# Chapter 6 Writing SAS/IML functions



## Introduction

With the SAS/TOOLKIT 6.07 release, you can write special functions that are called from within the SAS/IML environment. Standard DATA step functions (except for those using the SAS_DSS routines) can be invoked from within SAS/IML software, but functions that operate on vectors and matrices can also be written to be used exclusively with SAS/IML software.

## Example

A user-written SAS/IML function is implemented in a fashion similar to standard user-written functions. There is an IFFMAI routine defined with request values of 1 and 2, and the functions are named RTN1, RTN2, etc. The differences between standard user-written functions and SAS/IML functions are: 1) the FNCDFI routine is called instead of FNCDFN, and no FNCDFA routine is called; 2) the calling sequence for the RTNx routines is always the same (described below); and 3) additional interface routines (the SAS_IMxxxx routines described below) must be called.

Here is the IFFMAI routine from the sample SAS/IML function (called CIMLEXM1) supplied with the 6.07 production version of SAS/TOOLKIT Software:

```
ptr IFFMAI(request)
int *request;
{
if (*request == 1) {
   UWPRCC(0);
   FNCDFS(2);
   FNCDFI(1,"IMLEXCAL",1,1,0,2);
   FNCDFI(2,"IMLEXFUN",1,1,1,0);
   return(FNCDFE());
   }
else if (*request == 2) {
   return(FNCFNE());
```

```
   }
}
```

In this example, the difference between standard user-written functions and SAS/IML functions is shown in the use of FNCDFI instead of FNCDFN. The calling sequence for FNCDFI is as follows:

```
FNCDFI(funcnum,funcname,minarg,maxarg,return,nresult);

where

funcnum     int     function number being defined (1-based)
funcname    ptr     pointer to the function name as referenced by
                    the SAS/IML user; this pointer must point to
                    a character string padded with blanks to 8 bytes
minarg      int     the minimum number of input matrices
maxarg      int     the maximum number of input matrices
return      int     1=function (returns a matrix)
                    0=call routine (does not return a matrix)
nresult     int     the number of result matrices
```

Note that unlike standard user-written functions, you do not call FNCDFA to indicate the argument type for the input arguments. This is because a SAS/IML function permits numeric or character matrices to be passed interchangeably; it is the function's responsibility to determine if the matrix type is permissible.

Note also that arguments passed to a SAS/IML function fall into one of two categories: result matrices or input matrices. It is a requirement that all result matrices appear first in the calling sequence. It is then the responsibility of the function to ensure that SAS_IMWRES/SAS_IMWRESC (described below) is called for each result matrix. For example, our FNCDFI call above for IMLEXCAL indicates that there are two result matrices, and that there is a minimum and maximum of one input matrix. This means that when the SAS/IML user enters

```
CALL IMLEXCAL(b,c,a);
```

it will be assumed that b and c are result matrices, and a is an input matrix.

All arguments passed to a SAS/IML function are considered matrices, even if they are actually only scalar values (treated as a 1x1 matrix) or vectors (treated as a 1xN matrix).

Continuing our example, we present the code for the first defined SAS/IML function, the IMLEXCAL call routine, which will be RTN1. This SAS/IML function will accept one input matrix and create two result matrices. The first result matrix will contain a copy of the input matrix with an elementwise subtraction of 1. The second result matrix will also contain a copy of the input matrix with an elementwise addition of 1.

```
int RTN1(arg, to_n)

   ptr *arg;
   int *to_n;

{
   long    i, num;
   int     row, col, size;
   dblptr  r1, r2, a;
```

```
   SAS_IMWARG (arg[0],&row,&col,&size,&a);
   if (size != -8) return (IML_NOTNUMERIC);

   r1 = SAS_IMWRES(1,row,col);
   if (!r1) return (IML_MEMORY);

   r2 = SAS_IMWRES(2,row,col);
   if (!r2) return (IML_MEMORY);

   r1 = (dblptr) SAS_IMWRESP(1);

   SAS_IMWARG(arg[0],&row,&col,&size,&a);

   num = (long)row*col;

   for (i=0; i<num; i++,r1++,r2++,a++)
   {  *r1 = (*a)-1;
   *r2 = (*a)+1;
   }

   return(IML_OK);

}
```

All SAS/IML functions have the same calling sequence for the RTNx routines:

```
r = RTNx(arg,to_n);

where

arg     ptr*     pointer to the array of symbol table addresses
to_n    int*     pointer to the number of input matrices
r       int      return code (see below)
```

There is one symbol table address for each input matrix. In our example, there will be exactly one input matrix, so there is one symbol table address. The first symbol table address will be found at arg [0]. The symbol table address is passed to the SAS_IMWARG routine (see below) in order to obtain information about the input matrix, including the number of rows, columns, element size, and address of the matrix.

For any function, *to_n refers to the number of input matrices passed to the function. This is useful for those functions allowing different numbers of input matrices (indicated by minarg < maxarg in the FNCDFI call).

In the example, we call SAS_IMWARG to obtain information about our input matrix. If its element size is not -8 (indicating numeric), we know that the user entered the wrong calling sequence, so we indicate an error with the appropriate return code. We then allocate the memory for our two result matrices. Both result matrices will have the same number of rows and columns as the input matrix. If either allocation fails, we will return with the IML_MEMORY return code.

Now that all memory has been allocated, we need to re-resolve any result matrices allocated before the final memory allocation. In this case, result matrix 1 was allocated before result matrix 2, so we re-resolve it by calling SAS_IMWRESP with a value of 1 (for result matrix 1).

Since we know that memory allocation may relocate the input matrices as well, we'll

call SAS_IMWARG again to ensure we have the correct address for the input matrix.

The last part of the code is the algorithm to compute the two result matrices. The first result matrix contains the elementwise difference between the input matrix and 1. The second matrix contains the elementwise sum between the input matrix and 1.

Here is the code for the second example SAS/IML function, IMLEXFUN, which is a function (not a call routine). This function will sum up all the elements of the input matrix and return the sum as a scalar (that is, a 1x1 matrix).

```
int RTN2 (arg, to_n)

   ptr *arg;
   int *to_n;

{
   int      i, row, col, size;
   long     n;
   dblptr   r, a;
   double   sum;

   SAS_IMWARG(arg[0],&row,&col,&size,&a);
   if (size != -8) return (IML_NOTNUMERIC);

   r = SAS_IMWRES(1,1,1);
   if (!r) return (IML_MEMORY);

   SAS_IMWARG(arg[0],&row,&col,&size,&a);
   n = (long) row * col;
   sum = 0;
   for (i=0; i<n; i++,a++) sum += *a;
   *r = sum;

   return(IML_OK);
}
```

Note that the RTN2 calling sequence is the same as for RTN1.

As in RTN1, we check to ensure that the input matrix is numeric.

Functions consider their return value to be a result matrix. In this example, we want our return value to be a 1x1 matrix containing the sum of all elements in the input matrix, so we only have to call SAS_IMWRES for a 1x1 matrix, returning IML_MEMORY if memory is not available.

In this example, SAS_IMWRESP needn't be called since there were no additional allocations after SAS_IMWRES. We do, however, have to call SAS_IMWARG to obtain the possibly changed pointer for the input matrix.

The rest of the code is the implemented algorithm for the summing of the elements.

Note that the IFFEXT routine must also be included:

```
void IFFEXT()
```

It is not necessary for IFFEXT to do anything, but it must be present, or an unresolved reference will occur at link time, and your function package will fail upon termination of the SAS/IML environment.

## Other Notes

The linking of a user-written SAS/IML function is exactly the same as for any other user-written function. You supply a program constants object produced by PROC USERPROC (using MODTYPE=FUNCTION), you include your compiled objects, and you supply the other objects appropriate for a user-written function. SAS/IML functions use the same UWU prefix for module names, and use the same convention of requiring that the first five characters of the SAS/IML function names match with the first five characters after the UWU prefix in the module name.

All the implementation languages supported on a given operating system are available for implementing SAS/IML functions. All examples and documentation herein use the C language for consistency. Consult the sample SAS/IML function implemented in your language of choice to see how it may differ from the C implementation. The C sample source is called CIMLEXM1. The PL/I sample source is PIMLEXM1, PIMLEXM2, and PIMLEXM3. The FORTRAN sample source is FIMLEXM1. The Assembler sample source is AIMLEXM1.

Note that there is a sample jobstream for all supported languages. The C jobstream is called IMLJOBC1. The PL/I jobstream is called IMLJOBP1. The FORTRAN jobstream is called IMLJOBF1. The Assembler jobstream is called IMLJOBA1.

# SAS_IM*xxxx* Routine Reference

The user-written IML functions are characterized by the inclusion of several SAS_IMxxxx routines that interface to the SAS/IML supervisor. Each of these SAS_IMxxxx routines is documented below.

## SAS_IMWRES Routine

```
d = SAS_IMWRES(result,nrows,ncols);

where

result      int     result number (1-based)
nrows       int     number of rows
ncols       int     number of columns
d           ptr     pointer to allocated memory to hold doubles
```

SAS_IMWRES allocates memory to hold final numeric results from your computation. The result number is always 1 for a function call, since there is only one returned result. For a subroutine call, there may be more than one returned result, in which case each result must be separately allocated. The other two arguments are the number of rows and columns in the result matrix. For character results, use the routine SAS_IMWRESC. If the pointer returned from this routine is NULL, the requested memory could not be allocated and you must take necessary steps to handle the out-of-memory condition.

For example, if your SAS/IML code (not your function code) looks like this:

```
r = abc(x,y);
```

in your function code, you'll need to allocate 'r' using SAS_IMWRES(1,rows,cols); the pointer returned in this case points to an area in the memory of size

rows*cols*sizeof(double) bytes.

For a subroutine invocation in the SAS/IML code:

```
call abc(a,b,x,y);    where  a,b are result parameters
and    x,y are input  parameters
```

in your function code, you'll need to allocate 'a' using SAS_IMWRES(1,row1,col1) and 'b' using SAS_IMWRES(2,row2,col2).

Here we assume that r, a, and b are all numeric matrices.

## SAS_IMWRESC Routine

```
p = SAS_IMWRESC(result,nrows,ncols,size);

where

result       int      result number
nrows        int      number of rows
ncols        int      number of columns
size         int      element size
p            ptr      pointer to allocated memory
```

SAS_IMWRESC allocates memory to hold final character results from your computation. The result number is always 1 for a function call, since there is only one returned result. For a subroutine call, there may be more than one returned result, in which case each result must be separately allocated. The next two arguments are the number of rows and columns in the result matrix. In addition you also need to specify the element size for the result matrix. This is not needed for numeric results since all numbers are stored as sizeof(double). If the pointer returned from this routine is NULL, the requested memory could not be allocated and you must take necessary steps to handle the out-of-memory condition.

For example, in the SAS/IML code:

```
r = abc(x,y);
```

where r is a character matrix with each element of size 12 (bytes). In your function code, you'll need to allocate 'r' using SAS_IMWRESC(1,rows,cols,12). The pointer returned in this case points to an area in the memory of size rows*cols*12 bytes.

For a subroutine in the SAS/IML code:

```
call abc(a,b,x,y);    where  a,b are result parameters
and    x,y are input  parameters
```

in your function code, you'll need to allocate 'a' using SAS_IMWRESC(1,row1,col1,size1) and 'b' using SAS_IMWRESC(2,row2,col2,size2).

Here we assume that r, a, and b are all character matrices.

## SAS_IMWRESP Routine

```
p = SAS_IMWRESP(result);

where

result       int     result number
p            ptr     result pointer
```

SAS_IMWRESP refreshes or restores a result pointer obtained using SAS_IMWRES or SAS_IMWRESC calls. Restoring a result pointer is necessary only if you make any memory allocation after your SAS_IMWRES or SAS_IMWRESC call and before using the result pointer. The restoration is necessary due to the internal workspace management strategy that SAS/IML uses, that can cause objects to be displaced upon workspace compression. Any memory allocation can potentially cause a workspace compression and therefore a displacement of all allocated objects. This makes it necessary to restore a result pointer before using it, if any allocation was made since the result itself was allocated. The allocating routines are SAS_IMWRES, SAS_IMWRESC, and SAS_IMWALOC.

For example, in your function code:

```
out = SAS_IMWRES(1,10,5);    /* allocate a 10x5 result matrix */
SAS_IMWALOC((ptr *)&x,1000); /* allocate 1000 bytes of workarea */
out = SAS_IMWRESP(1);        /* restore the result */
out[0] = x[100];             /* use the result pointer */
```

or

```
out1 = SAS_IMWRES(1,10,5);   /* allocate a 10x5 result */
out2 = SAS_IMWRES(2,4,4);    /* allocate a  4X4 result  */
out1 = (dblptr) SAS_IMWRESP(1); /* restore the first result */
out1[0] = ...
out2[0] = ...
```

In this case it is necessary to restore the first result only, since an allocation was made for the second result after out1 was set. There is no need to restore out2, since no allocation is made subsequent to its own allocation.

It is generally a good practice to allocate results only after all other allocations, such as for temporary work area, are made. This eliminates the need to restore the result pointer if there is only one result.

A SAS_IMWRESP call issued when not necessary is harmless. So when in doubt, it is best to call it.

## SAS_IMWALOC Routine

```
SAS_IMWALOC(&p,size);

where

p            ptr    pointer value to be set by SAS_IMWALOC
size         long   allocation size
```

SAS_IMWALOC allocates memory for scratch area or work area for either numeric or character data. The memory allocated using imwaloc must be freed using SAS_IMWFREE, before you return control to SAS/IML.

For example, for doing some character operation, you need work area 3 times the size of the matrix, your function code would look like this:

```
SAS_IMWALOC(&x, 3L*rows*cols*size);
```

If the pointer x is NULL after this call, memory could not be allocated. You must handle this as an out-of-memory condition (that is, returning with the IML_MEMORY value).

## SAS_IMWFREE Routine

```
SAS_IMWFREE(&p);

where

p        ptr       ptr returned by SAS_IMWALOC
```

The SAS_IMWFREE routine is used to free up memory allocated using the SAS_IMWALOC routine. Every successful SAS_IMWALOC allocation must be freed using SAS_IMWFREE before control is returned to SAS/IML.

## SAS_IMWARG Routine

```
SAS_IMWARG(p,&nrows,&ncols,&size,&mp);

where

p      ptr       symbol table address
nrows  int       number of rows (returned)
ncols  int       number of columns (returned)
size   int       size of element (returned)
mp     ptr       address of matrix (returned)
```

The pointer to the array of symbol table addresses is passed as the first parameter to your RTNx routine. You obtain the appropriate element of the symbol table array and pass that as the first argument to SAS_IMWARG. SAS_IMWARG can then tell you the number of rows and columns in the matrix, and the element size. Note that if size is negative, the matrix is numeric, and ABS(size) is the size of each element, which should always be sizeof(double). SAS_IMWARG also returns the actual address of the matrix. This pointer will point to an array of doubles if the matrix is numeric, and will point to an array of character strings for a character matrix.

Note that the address of the matrix may change once you make calls to SAS_IMWALOC, SAS_IMWRES, or SAS_IMWRESC, so you'll need to call SAS_IMWARG again after all allocations are complete. Be careful not to save the address returned by SAS_IMWARG if subsequent SAS_IMWALOC/SAS_IMWRES/SAS_IMWRESC calls are made. Using this saved address after these calls can cause unpredictable results, since the original matrices may have been relocated.

## SAS/IML Function Return Codes

Here are the return codes that you can use to return indicating successful completion or any warning or error condition. All of them are defined in the UWPROC include file, which must be included in your program.

| CODE | ASSOCIATED ERROR OR WARNING MESSAGE |
|---|---|
| IML_OK | (Successful completion) |
| IML_WARN | (Warning message printed by the library routine) |
| IML_MEMORY | ERROR: (execution) Unable to allocate sufficient memory. At least xxx more bytes required. |
| IML_NULLMATRIX | ERROR: (execution) Matrix has not been set to a value |
| IML_NOTCONFORM | ERROR: (execution) Matrices do not conform to the operation. |
| IML_NOTSQUARE | ERROR: (execution) Matrix should be square. |
| IML_NOTPOSDEF | ERROR: (execution) Matrix should be positive definite. |
| IML_SINGULAR | ERROR: (execution) Matrix should be non-singular. |
| IML_NOTSYMMETRIC | ERROR: (execution) Matrix should be symmetric. |
| IML_INVALIDARG | ERROR: (execution) Invalid argument to function. |
| IML_NOTNUMERIC | ERROR: (execution) Character argument should be numeric. |
| IML_NOTCHARACTER | ERROR: (execution) Numeric argument should be character. |
| IML_NOTSCALAR | ERROR: (execution) Argument should be a scalar. |
| IML_MISSVAL | ERROR: (execution) Invalid argument or operand; contains missing values. |
| IML_GTMACINT | ERROR: (execution) Result matrix dimension cannot be greater than MACINT. (MACINT is the largest integer on your machine). |
| IML_LIBERROR | (Error message printed by the library routine) |
| IML_STOPERROR | (STOP return code, stops executing already submitted statements, does not quit IML) |
| IML_ABORTERROR | (ABORT return code, stops executing, quits PROC IML) |

## Using Subroutine Libraries with SAS/IML Functions

You may find it useful to include subroutines from commercial subroutine libraries (such as IMSL or ESSL). There are two example SAS/C functions, CIMLEXM2 and CIMLEXM3 (for the IBM MVS operating system only) that refer to ESSL and IMSL routines, respectively. You can use these examples as models for developing your own applications.

Note that most subroutine library functions do expect vector and/or matrix arguments, which would not be supported outside of the SAS/IML environment.

# Chapter 7 Other New Routines



## Introduction

The routines described in this chapter update the SAS_X and SAS_Z routines described in *SAS/TOOLKIT Software: Usage and Reference, Version 6, First Edition.*

## SAS_XDNAMCL Routine

The SAS_XDNAMCL routine is called to terminate processing initiated by SAS_XDNAME.

```
rc = SAS_XDNAMCL(x);

where

x       struct MEMLIST *     ptr returned by SAS_XDNAME
rc      int                  0=successful nonzero=otherwise
```

You can call SAS_XPRLOG with rc upon return from SAS_XDNAMCL to provide the user with more information when SAS_XDNAMCL is unsuccessful. Passing SAS_XPRLOG a value of 0 (successful) will not cause any message to be printed on the log.

You should call SAS_XDNAMCL once you are finished with the structures created by SAS_XDNAME. Until SAS_XDNAMCL is called, or until the end of your procedure, the libref associated with the memlist cannot be cleared. Note that if you are calling SAS_XDNAME from within a data step function, automatic clearing does not occur until your function is deleted. Therefore, you may want to call SAS_XDNAMCL as soon as possible to ensure it does not interfere with a user wanting to clear the libref.

## SAS_ZCATMEM Routine

The SAS_ZCATMEM routine is used to obtain the list of members in a catalog.

```
rc = SAS_ZCATMEM(libname,catname,&listptr,&listn);

where

libname  ptr                 libref to use
catname  ptr                 catalog name
listptr  ptr                 pointer to member list (returned)
```

```
listn    long                number of members (returned)
rc       long                0=successful nonzero=otherwise
```

You can call SAS_XPRLOG with rc upon return from SAS_ZCATMEM to provide the user with more information when SAS_ZCATMEM is unsuccessful. Passing SAS_XPRLOG a value of 0 (successful) will not cause any message to be printed on the log.

The member list consists of 16-byte elements. Each element contains a member name and a member type, each 8 bytes in length.

When you are done with the list, you can free it by calling the SAS_XMEMFRE routine.

Example of use:

```
long rc;
ptr listptr,p;
long listn,i;
/* list the members of the current format catalog */
rc = SAS_ZCATMEM("WORK    ","FORMATS ",&listptr,&listn);
SAS_XPRLOG(rc);
if (rc == 0) {
   for (i=0,p=listptr;i<listn;i++,p += 16) {
      SAS_XPSLOG("Member=%8s type=%8s",p,p+8);
      }
   SAS_XMEMFRE(listptr);
   }
```

## SAS_ZMISSVF Routine

The SAS_ZMISSVF routine fills in a location with a float (not a double) missing value. This routine is used in conjunction with graphics procedures, and should not otherwise be used.

```
SAS_ZMISSVF(&value);

where

value    float               value to be set to standard missing
```

# Chapter 8 PARMCARDS Processing



## Introduction

The primary purpose of the PARMCARDS processing is for external applications that perform their own parsing of input statements and/or data. You can use PARMCARDS processing with your SAS procedure if you do the following:

- include statements in your procedure that access and parse the PARMCARDS file
- use the SAS system option, PARMCARDS=, in the SAS program that calls your procedure
- use the PARMCARDS statement with the SAS statements that invoke your procedure.

This chapter illustrates PARMCARDS processing.

## Required SAS Statements

If your procedure uses PARMCARDS, the user invoking your procedure must specify the SAS system option, PARMCARDS= . In addition, the user must include the PARMCARDS or PARMCARDS4 statement in the statements that invoke your procedure.

```
options parmcards=fileref;
proc your-proc;
   parmcards;
   data-line-1
   data-line-2
   data-line-3
   ;
```

or

```
options parmcards=fileref;
proc your-proc;
   parmcards4;
   data-line-1
   data-line-2
   data-line-3
;;;;
```

When the SAS System encounters either of these statements, it writes the data lines following the statement to the file referenced by the PARMCARDS= system option until it reaches a line with a semicolon (for the PARMCARDS statement) or a line with four semicolons beginning in column 1 (for the PARMCARDS4 statement). Keep in mind that the line containing the semicolons is not written to the file. Note also that the lines are not parsed, so no macro processing is performed on the data lines.

# PARMCARDS Processing in Your Procedure

To access the data lines stored by the user in the PARMCARDS file, your procedure must call the SAS_XOPTCGT routine to obtain the fileref. Then, after you return from the call to SAS_XSPARSE that parses all the procedure statements (and also writes all PARMCARDS data lines to the file), you can read the data from the PARMCARDS file, using the fileref returned by SAS_XOPTCGT.

There is no specific mechanism in place to verify whether a user has actually specified a PARMCARDS statement or whether any lines have been provided. Your procedure needs to check for two possible errors:

- If the user omits the PARMCARDS statement, you can detect this error by opening the PARMCARDS file and writing a single line to it. After calling SAS_XSPARSE, your procedure can then test to see if the PARMCARDS file contains only that line. If that is all the file contains, this means that the user did not specify the PARMCARDS statement at all.
- If the user specifies the PARMCARDS statement but does not provide any data lines, the PARMCARDS file will be completely empty. This will cause the attempt to open the file to fail.

If the user correctly supplies the PARMCARDS statement and the data lines you expect, your procedure must then parse the data lines.

**Note:** For the SAS/C compiler, a fileref (DDname) should be prefixed with `ddn:` to distinguish it from a data set name.

# Example

The following example includes the C source code for a procedure, PROC PMCDTEST, that tests the existence of PARMCARDS processing. The example also shows the grammar for this procedure as well as sample statements and the resulting log from testing the procedure.

## Procedure Source Code

```
#define MAINPROC 1
#define SASPROC  1
#include "uwproc.h"
#undef NULL
#include <stdio.h>
ptr     PMCDG U_PARMS(( void ));
void    U_MAIN(PMCDTEST)()
{
```

```
char filename[13];
char record[81];
FILE *fid;
ptr p;
int rc,i;

/*---init the proc environment and announce---*/
UWPRCC(&proc);
SAS_XPSLOG("PROC PMCDTEST PARMCARDS testing...");

/*---get the fileref name and append it to ddn: for SAS/C---*/
strcpy(filename,"ddn:");
p = SAS_XOPTCGT("PARMCARDS",&rc);
strcat(filename+4,p);
SAS_XPSLOG("ddname to test is '%s'...",filename);

/*---open the PARMCARDS file and write out our single record---*/
if ((fid = fopen(filename,"w")) == NULL) {
   SAS_XPSLOG("Cannot open PARMCARDS file to initialize...");
   SAS_XEXIT(XEXITERROR,0);
   }
fputs("DUMMY RECORD",fid);
fclose(fid);

/*---now go parse the statements, including PARMCARDS data---*/
SAS_XSPARSE(PMCDG(),NULL,&proc);

/*---complain if we can't open (no PARMCARDS data lines)---*/
if ((fid = fopen(filename,"r")) == NULL) {
   SAS_XPSLOG("No PARMCARDS data lines given...");
   SAS_XEXIT(XEXITERROR,0);
   }

/*---read each data line---*/
for (i=0;fgets(record,80,fid) != NULL;i++) {

   /*---verify that first record isn't our dummy one---*/
   if (i == 0 && strcmp(record,"DUMMY RECORD\n") == 0) {
      SAS_XPSLOG("No PARMCARDS statement given...");
      break;
      }

   /*---announce the data line---*/
   SAS_XPSLOG("PARMCARDS record: '%*s'.",strlen(record)-1,record);
   }

/*---announce the total---*/
SAS_XPSLOG("Total of %d PARMCARDS records seen.",i);

/*---close up and terminate---*/
fclose(fid);
SAS_XEXIT(XEXITNORMAL,0);
}
```

## Grammar

```
#----test grammar for proc pmcdtest--------------#
%INCLUDE STUBGRM.
PROGRAM      = ANYSTMT ENDJB ,
ANYSTMT      = PMCDTESTSTMT ,
PMCDTESTSTMT = @PROCINIT @STMTINIT(1) "PMCDTEST" @STMTEND .
#----end of grammar-----------------------#
```

Note that PARMCARDS is a global statement, and does not appear in the grammar.

## SAS Statements to Test Procedure

The following examples show the log output from various invocations of the procedure.

In this example, the user invokes the procedure but omits the PARMCARDS statement:

```
proc pmcdtest;
run;
```

The log output from these statements follow:

```
3          PROC PMCDTEST; RUN;
PROC PMCDTEST PARMCARDS testing...
ddname to test is 'ddn:SASPARM'...

No PARMCARDS statement given...
Total of 0 PARMCARDS records seen.
```

These statements include the PARMCARDS statement, but omit the data lines that should follow PARMCARDS.

```
proc pmcdtest;
   parmcards;
run;
```

The log follows:

```
4          PROC PMCDTEST;
PROC PMCDTEST PARMCARDS testing...
ddname to test is 'ddn:SASPARM'...
4                        PARMCARDS;

No PARMCARDS data lines given...
NOTE: The SAS System stopped processing this step because of errors.

5          RUN;
```

This example correctly invokes the procedure, includes the PARMCARDS statement, and provides data lines:

```
proc pmcdtest;
```

```
   parmcards;
first line
last line
;
```

The log follows:

```
6          PROC PMCDTEST;
PROC PMCDTEST PARMCARDS testing...
ddname to test is 'ddn:SASPARM'...
6                         PARMCARDS;
7          FIRST LINE
8          LAST LINE
9          ;

PARMCARDS record: 'FIRST LINE'.
PARMCARDS record: 'LAST LINE'.
Total of 2 PARMCARDS records seen.
```

This example uses the PARMCARDS4 statement and ends the data lines with four semicolons:

```
proc pmcdtest;
   parmcards4;
one line
another line
with semicolon;
we're done
;;;;
```

The log follows:

```
10         PROC PMCDTEST;
PROC PMCDTEST PARMCARDS testing...
ddname to test is 'ddn:SASPARM'...
10                        PARMCARDS4;
11         ONE LINE
12         ANOTHER LINE
13         WITH SEMICOLON;
14         WE'RE DONE
15         ;;;;

PARMCARDS record: 'ONE LINE'.
PARMCARDS record: 'ANOTHER LINE'.
PARMCARDS record: 'WITH SEMICOLON;'.
PARMCARDS record: 'WE'RE DONE'.
Total of 4 PARMCARDS records seen.
```

# Appendix 1 Using SAS/TOOLKIT® Software Under OS/2®



## Introduction

This appendix provides sample `make` files for compiling and linking your procedure, function, CALL routine, informat, format, or engine. Under OS/2 2.0 you can create procedures, IFFCs, and engines using the IBM C Set/2 compiler. In addition, this appendix discusses how to use user-written SAS modules after they are compiled and linked and how to make those modules available to other sites.

## Note on Supported Languages

At the current time, SAS/TOOLKIT software supports only the IBM Set/2 C compiler for the OS/2 2.0 environment. SAS Institute hopes to support the Borland C++ compiler for the OS/2 2.0 environment when that compiler becomes available. There are currently no plans to support any other C compilers in this environment.

Currently, SAS/TOOLKIT software supports only C as an implementation language in the OS/2 2.0 environment. SAS Institute hopes to support FORTRAN as another implementation language in the future. However, IBM currently does not provide a FORTRAN compiler for OS/2 2.0. Instead, IBM refers customers to the MicroWay and Waterloo FORTRAN products. SAS Institute continues to investigate the viability of supporting either of these compilers. If you are interested in using FORTRAN as an implementation language, please contact Technical Support at SAS Institute to discuss your compiler preferences.

# Accessing User-Written SAS Modules

This section discusses how to access user-written modules and how to make these modules available to other sites.

## Specifying Where to Find Modules

Procedures, IFFCs, and engines are executable modules that are stored as DLL files. Images supplied by SAS Institute usually have a SAS or SAB prefix, but user- written images will not.

In order to access your user-written module, you must indicate its location with a -path option that is provided to the SAS command. This option can be specified on the command line, or placed into a config.sas file. The value of the -path option should be the name of the directory in which the executable resides. If you have multiple executables in multiple directories, you can issue the -path option multiple times.

For example, suppose your user-written procedure ABC is located in the directory c:\tlkt\c\load. To access the procedure, use the following statement:

```
sas -path c:\tlkt\c\load
```

If you have procedure DEF in c:\tlkt\c\loadx, and wish to access both ABC and DEF in the same SAS job, the command would be:

```
sas -path c:\tlkt\c\load -path c:\tlkt\c\loadx
```

Files in the c:\tlkt\c\load directory will be accessed before files in c:\tlkt\c\loadx.

Setting up the -path options in the config.sas file will be especially convenient if there are several paths to include. Refer to the discussion of the -PATH option in Chapter 7 of *SAS Companion for the OS/2 Environment, Version 6, Second Edition.*

## Distributing Modules to Other SAS Sites

You can distribute a module you have written to other SAS Sites that are running OS/2 2.0. The site receiving your module does not have to license SAS/TOOLKIT Software in order to run your module.

You can provide the module to your off-site users by any traditional file transferal mechanism, such as diskette or by the file transfer software. However, you must send the module as a DLL file, because the site will not be permitted to recompile or relink your code unless it also licenses SAS/TOOLKIT Software.

## SAS System Options for Successful Testing

This section describes two options that will help you as you test your programs. The second option is needed only until a bug in the IBM code can be corrected.

### Option for Saving the LOG During an Abend

The SAS/System under OS/2 provides a special configuration option that enables you to more easily test user-written modules. During the development phase of a module, your program may occasionally receive a GPF (general protection fault) or some other failure that causes the SAS System to terminate. With such a termination, no LOG or LIS file is created, causing difficulty in retrieving debugging messages or other indicators of the problem cause. To ensure the LOG file is created (and the LIS file if appropriate), you must specify the WXFILEFLUSH configuration option when you invoke the SAS System to ensure that the log is not lost if the system abends. You can specify this option on the SAS command or in the CONFIG.SAS file. This example illustrates using the option on the SAS command:

```
sas -wxfileflush
```

### Option Required for C Subroutine

If your C code uses certain C subroutines, such as `fopen` or `fclose`, you may encounter a bug in the IBM code when you invoke your program the second time. The problem occurs because the C termination routines do not properly shut down the environment for certain C subroutines, so that a failure occurs with reinvocation. IBM Technical Support is aware of this bug, and a fix is forthcoming, but at the time of publication of this document, the fix was not yet available. The workaround for this problem is as follows:

1. Invoke your module the first time as usual.
2. Before invoking it subsequent times, enter this SAS option:

   ```
   OPTIONS CDE=P;
   ```

   This option purges any loaded module that has been marked as purgeable. Any SAS/TOOLKIT application module will be purged at that time.
3. Reinvoke your module. Since a fresh copy is being used, the IBM bug is avoided.

Please note that you must specify OPTIONS CDE=P every time that you wish a module to be purged. Specifying the option once does not cause automatic purging.

# Creating Executable Modules

Executable modules are created under OS/2 2.0 by using the IBM C Set/2 compiler to compile the necessary components, then using LINK386 to link the objects into a DLL file. This process can be handled by the `nmake` command, which greatly simplifies the software development effort. Most of the discussion in this appendix concerns the use of `nmake` (hereafter referred to as `make`) to create your executables. This appendix assumes that you have a working knowledge of `make`. If you are not familiar with the `make` facility, we recommend that you consult OS/2 programming documentation to learn more about it before attempting to build your SAS/TOOLKIT executables. "Compiling and Linking C Programs without make Files" on page 136 lists the commands that `make` invokes to create the objects and executables. If you choose to avoid using `make`, you can model your commands after those discussed.

This example illustrates using `nmake`:

```
nmake /f prcjobc6.mak
```

## The Make Files

Each sample procedure, IFFC, and engine has a `make` file associated with it, with the extension MAK. All the sample `make` files first include the global `make` file TOOLKIT.MAK.

**Note:** As you review the contents of the `make` files illustrated in this appendix, keep in mind that \ is a special character used with `make` to continue lines. To use \ as a literal character in a path name, you must precede it with the escape character ^.

## TOOLKIT.MAK File

The TOOLKIT.MAK file contains all the macro definitions that are specific to PC operating systems, but are not dependent on a specific operating system. The TOOLKIT.MAK file is as follows:

```
#########################################################################
# Copyright (C) 1992 by SAS Institute Inc., Cary NC USA 27512-8000    #
# NAME:    toolkit.mak                                                  #
# PRODUCT: SAS/TOOLKIT                                                  #
# PURPOSE: primary make file included by all applications               #
#########################################################################

!include tlkthost.mak

# PC-oriented macros

PRCINTCV        = $(TLKTDIR)^\prcintcv.obj
PRCINTCA        = $(TLKTDIR)^\prcintca.obj
IFFINT          = $(TLKTDIR)^\iffint.obj
ENGINTCV        = $(TLKTDIR)^\engintcv.obj
ENGINTCA        = $(TLKTDIR)^\engintca.obj
OBJLIB          = $(TLKTDIR)^\toolkit.lib
```

```
SASCMDGRM       = $(SASCMD) $(TLKTSAMP)grmfunc.sas -log nul:
                  -path $(GLOBAL_LOAD) -sysparm $(SYSPARM)
MAKE_PGMCON     = $(SASCMD) $(TLKTSAMP)pgmcon.sas
                  -path $(GLOBAL_LOAD) -sysparm $(SYSPARM)
FTNPREP_CMD     = $(SASCMD) $(TLKTSAMP)ftnprep.sas
                  -path $(GLOBAL_LOAD) -log nul: -sysparm
MAKE_GRMSRCD    = $(GRM)$(MODNAME).grm
MAKE_GRMSRCC    = $(SASCMDGRM)
MAKE_GRMOBJD    = $(SRC)$(FUNCFILE).c
MAKE_GRMOBJC    = $(COMPILE_GRMFUNC)
MAKE_GRMOBJ_FTND= $(FUNCFILE).f
MAKE_GRMOBJ_FTNC= $(COMPILE_GRMFUNC_FTN)
SYSPARM1        = *$(WHICH)*$(MODNAME)*$(PGMCON)
SYSPARM2        = *@$(FUNCFILE)*@$(GRAMFUNC)*@$(GRM)
SYSPARM3        = *@$(SRC)*@$(INCLLIB)*@$(OBJ)*$(LANGUAGE)
SYSPARM         = $(SYSPARM1)$(SYSPARM2)$(SYSPARM3)
PGMCONIFF       = -o $(LOAD)$(MODNAME) $(UWLINKIFF)
PGMCONENG       = -o $(LOAD)$(MODNAME) $(UWLINKENG)
PGMCONPROC      = -o $(LOAD)$(MODNAME) $(UWLINKPROC)
COMPILE_GRMFUNC = $(UWC) -Fo$(OBJ)$(FUNCFILE).obj
                  $(SRC)$(FUNCFILE).c $(UWCOPTS)
COMPILE_GRMFUNC_FTN = $(UWFORT) $(FUNCFILE).f $(UWFORTOPTS)
```

## TLKTHOST.MAK File

The TOOLKIT.MAK file includes the TLKTHOST.MAK file, which contains the information specific to OS/2 2.0. You must alter this TLKTHOST.MAK file to reflect the correct information for your specific machine.

The TLKTHOST.MAK file supplied for OS/2 2.0 is as follows:

```
#########################################################################
# Copyright (C) 1992 by SAS Institute Inc., Cary NC USA 27512-8000    #
# NAME:    tlkthost.mak                                               #
# PRODUCT: SAS/TOOLKIT                                                #
# PURPOSE: make file used for IBM C Set/2 Compiler for OS/2 2.0       #
#########################################################################

# host-specific macros for OS/2 32-bit using IBM C Set/2 Compiler

# C compiler options set:
# -O-  no optimize (recommended if -Ti used)
# -Ti  general debugger symbols (recommended if debugger used)
# -Ge- build a DLL file (required)
# -Gn+ do not search libraries during compilation (required)
# -Gs+ omit stack checking (required)
# -C   don't link after compile (required)
# -I   directory for #include files

(1) UWC             = icc -O- -Ti -Ge- -Gn+ -Gs+ -C -I$(MACLIB)
OSNAME          = OS2_32
VENDOR          = IBM
```

```
# site-specific

❷ INCLLIB         = toolkit-directory^\global^\grm^\
TLKTDIR         = toolkit-directory^\global^\obj32^\
GLOBAL_LOAD     = toolkit-directory^\global^\load32^\
TLKTSAMP        = toolkit-directory^\global^\test^\
MACLIB          = toolkit-directory^\c^\maclib
SASCMD          = sas
VENDOR_LIBS     =
u:^\ibmc_32^\lib^\dde4sbs+u:^\ibmc_32^\lib^\dde4sbso+u:^\sdk_20^\os2libos2386
```

1. The first section of the TLKTHOST.MAK file contains system-specific information. You seldom need to change this section. One example of when you might change it would be if the UWC macro did not provide the correct command for the C compiler. If your C compiler is located in a path not specified by your PATH command, or if you have renamed it to something other than ICC, then you would have to change TLKTHOST.MAK to reflect this. Otherwise, do not change anything else in the first section.
2. The second section consists of directory names that are site-specific. Check with your SAS Software Representative to determine where these files are installed at your site. The macros and their meanings are:

| Macros | Description |
|---|---|
| INCLLIB | Location of the stub.grm grammar file |
| TLKTDIR | Location of the SAS/TOOLKIT global object directory |
| GLOBAL_LOAD | Location of the SAS/TOOLKIT preproc/sasp/postproc modules |
| TLKTSAMP | Location of the SAS/TOOLKIT ftnprep, pgmcon, and grmfunc SAS programs |
| SASCMD | Your local SAS command |
| MACLIB | This macro should not be changed. |

## Setting up a Make File to Use

The best way to understand how to set up a `make` file for building a user-written module is to use a sample `make` file. Consider this `make` file for creating PROC MULTIPLY using the C language:

```
#########################################################################
# Copyright (C) 1992 by SAS Institute Inc., Cary NC USA 27512-8000    #
# NAME:    prcjobc6.mak                                               #
# PRODUCT: SAS/TOOLKIT                                                #
# PURPOSE: sample make file for creating PROC MULTIPLY on OS2 2.0     #
#########################################################################
❶ !include toolkit.mak
```

```
❷ # local macros
  UWOBJ           = $(OBJ)cmultipl.obj
  MODNAME         = multiply
  FUNCFILE        = cmultg
  GRAMFUNC        = multg
  PGMCON          = mcbmulti
  WHICH           = PROC
  LANGUAGE        = c
  PRCINT          = $(PRCINTCV)

❸ !include where.mak

❹ # local MAKE definitions
  ALL             : $(LOAD)$(MODNAME).dll
❺ $(OBJ)cmultipl.obj      : $(SRC)cmultipl.c ; \
              $(UWC) -Fo$(OBJ)cmultipl.obj $(UWCOPTS) $(SRC)cmultipl.c

❻ !include grmfunc.mak
  !include pgmcon.mak
  !include uwlnkprc.mak
```

1. The first statement of the `make` file is the include statement for TOOLKIT.MAK. This include statement **must** be present for the `make` file to work correctly.
2. The next section of the `make` file consists of local macro definitions. These macros provide the necessary information to allow the packaged macros of toolkit.mak to work correctly. Here is a list of the macros that can be defined:

| Macro | Used By | Description |
|---|---|---|
| UWOBJ | all | lists all objects for which you will be providing the source (not including the grammar function) |
| MODNAME | all | the name of the executable and the name of the grammar file (which will have a .GRM extension) |
| FUNCFILE | procs | the name of the C file that will contain your grammar function |
| GRAMFUNC | procs | the name of the function as it is called from your proc source |
| PGMCON | all | the name of your program constants object |
| WHICH | all | describes which type of module is being created: PROC, FUNCTION, FORMAT, INFORMAT, ENGINE |
| LANGUAGE | all | currently only C is accepted |
| PRCINT | all | currently only $(PRCINTCV) is accepted |
| ENGINT | engines | currently only $(ENGINTCV) is accepted |

3. The next statement after the macro definitions is an include for WHERE.MAK. The WHERE.MAK file describes the location of the source, object, maclib, and load (executable) directories. For example:

```
SRC             = ..^\src^\
OBJ             = ..^\obj^\
MACLIB          = ..^\maclib^\
LOAD            = ..^\load^\
GRM             = $(INCLLIB)
```

   Providing SRC, OBJ, MACLIB, LOAD, and GRM allows you to separate your differing file types. If you don't see a need for this, simply omit the !INCLUDE statement for WHERE.MAK, or provide a WHERE.MAK file with no definitions.

4. The next item in the sample `make` file is the local definitions. The first definition is ALL, and it should always refer to $(LOAD)$(MODNAME).DLL. This will expand into the executable name that is to be created by `make`. (Note that if you choose to not provide a definition for LOAD in the WHERE.MAK file, it will reduce to null and only the executable name will appear in the ALL definition, meaning that the resulting executable will be placed in the current directory).

5. After the ALL definition, specify the definitions and dependencies for all objects that were listed in the UWOBJ macro. This example has one object:

```
$(OBJ)cmultipl.obj      : $(SRC)cmultipl.c ; \
          $(UWC) -Fo$(OBJ)cmultipl.obj $(UWCOPTS) $(SRC)cmultipl.c
```

   This statement indicates that the CMULTIPL.OBJ object is dependent on the CMULTIPL.C source. The object is generated by the C compilation of the source using the required options.

6. After your object definitions, you should provide !INCLUDE statements depending upon what your module is doing. If you are writing a procedure, you must add the following line:

```
!include grmfunc.mak
```

   This will include the definitions and dependencies to allow the grammar function to be created and compiled from the grammar source. This line is used only for procedures.

   For all applications, you must add the following line:

```
!include pgmcon.mak
```

   This will include the definitions and dependencies to create the program constants object.

   The last include line depends on the type of module being created:

| **To include ...** | **Specify this statement:** |
|---|---|
| procs | `!include uwlnkprc.mak` |
| IFFCs | `!include uwlnkiff.mak` |
| engines | `!include uwlnkeng.mak` |

The uwlnkxxx.mak file is responsible for ensuring that the executable is linked using the program constants object, the grammar function object (procs only), the objects in the UWOBJ list, and the SAS/TOOLKIT objects and library.

## Sample Make File for Functions

```
########################################################################
# Copyright (C) 1992 by SAS Institute Inc., Cary NC USA 27512-8000    #
# NAME:    iffjobuc.mak                                               #
# PRODUCT: SAS/TOOLKIT                                                #
# PURPOSE: sample make file for creating sample functions on OS2 2.0  #
########################################################################
!include toolkit.mak

# local macros
UWOBJ           = $(OBJ)cfunc.obj
MODNAME         = uwutestf
PGMCON          = mcbuwute
WHICH           = FUNCTION
LANGUAGE        = c
PRCINT          = $(PRCINTCV)
IFFINT          = $(IFFINTCV)

!include where.mak

# local MAKE definitions
ALL             : $(LOAD)$(MODNAME).dll
$(OBJ)cfunc.obj         : $(SRC)cfunc.c ; \
                  $(UWC) -Fo$(OBJ)cfunc.obj $(UWCOPTS) $(SRC)cfunc.c

!include pgmcon.mak
!include uwlnkiff.mak
```

## Sample Make File for Informats

```
########################################################################
# Copyright (C) 1992 by SAS Institute Inc., Cary NC USA 27512-8000    #
# NAME:    iffjobic.mak                                               #
# PRODUCT: SAS/TOOLKIT                                                #
# PURPOSE: sample make file for creating sample informats on OS2 2.0  #
########################################################################
!include toolkit.mak

# local macros
UWOBJ           = $(OBJ)cinfmt.obj
MODNAME         = uwitesti
PGMCON          = mcbuwite
WHICH           = INFORMAT
LANGUAGE        = c
PRCINT          = $(PRCINTCV)
IFFINT          = $(IFFINTCV)
```

```
!include where.mak

# local MAKE definitions
ALL             : $(LOAD)$(MODNAME).dll
$(OBJ)cinfmt.obj         : $(SRC)cinfmt.c ; \
                  $(UWC) -Fo$(OBJ)cinfmt.obj $(UWCOPTS) $(SRC)cinfmt.c

!include pgmcon.mak
!include uwlnkiff.mak
```

### Sample Make File for Engines

```
########################################################################
# Copyright (C) 1992 by SAS Institute Inc., Cary NC USA 27512-8000    #
# NAME:    engjobc1.mak                                               #
# PRODUCT: SAS/TOOLKIT                                                #
# PURPOSE: sample make file for creating sample engine    on OS2 2.0  #
########################################################################
!include toolkit.mak

# local macros
UWOBJ           = $(OBJ)cengxmpl.obj
MODNAME         = engxmpl
PGMCON          = mcbengxm
WHICH           = ENGINE
LANGUAGE        = c
PRCINT          = $(PRCINTCV)
ENGINT          = $(ENGINTCV)

!include where.mak

# local MAKE definitions
ALL             : $(LOAD)$(MODNAME).dll
$(OBJ)cengxmpl.obj         : $(SRC)cengxmpl.c ; \
                  $(UWC) -Fo$(OBJ)cengxmpl.obj $(UWCOPTS) $(SRC)cengxmpl.c

!include pgmcon.mak
!include uwlnkeng.mak
```

## Running Your Make File

After you create your `make` file, run it in your development directory. First, copy TOOLKIT.MAK, TLKTHOST.MAK, PGMCON.MAK, and UWLINK.MAK from the C\CNTL directory under the top-level SAS/TOOLKIT directory. You will also need copies of any other `make` files you are including, such as WHERE.MAK and UWLNKPRC.MAK. When you have all the `make` files in place, you can invoke `make` as follows:

```
nmake /f yourfile.mak
```

If all is correct, `make` recreates any of the components that have changed since you last ran `make`, and finally links the executable.

For a list of the actual commands produced by `make`, see the discussion in the next sections.

# Compiling and Linking without Using Make Files

This section lists all of the commands used by the `make` facility. If you are using `make`, you do not need to know the detailed information listed in this part of the appendix. However, if you are not using `make`, you can use the contents of these command lists to model your own process for compiling and linking.

**Note:** The term *toolkit-directory* is used in these command lists to represent the top-level directory where the SAS/TOOLKIT product is stored on your PC.

## Command List for Compiling and Linking Procedures

This section contains the set of commands generated by a `make` of the sample procedure, cmultipl.c.

```
❶ icc -O- -Ti -Ge- -Gn+ -Gs+ -C -Itoolkit-directory\c\maclib\ -Focmultipl.obj
      toolkit-directory\c\src\cmultipl.c
❷ sas toolkit-directory\global\test\pgmcon.sas
      -path toolkit-directory\global\load32\
      -sysparm *PROC*multiply*mcbmulti*@cmultg*@multg*@*
               @toolkit-directory\c\src\*@toolkit-directory\global\grm\*@*c
❸ sas toolkit-directory\global\test\grmfunc.sas
      -path toolkit-directory\global\load32\
      -sysparm *PROC*multiply*mcbmulti*@cmultg*@multg*@*
               @toolkit-directory\c\src\*@toolkit-directory\global\grm\*@*c
❹ icc -O- -Ti -Ge- -Gn+ -Gs+ -C -Itoolkit-directory\c\maclib\ -Focmultg.obj
      toolkit-directory\c\src\cmultg.c
❺ lnkdef c IBM OS2_32 PROC  @
         @\ibmc_32\lib\dde4sbs+
         \ibmc_32\lib\dde4sbso+
         \sdk_20\os2lib\os2386 @ @toolkit-directory\global\obj32\
         @ multiply mcbmulti cmultipl.obj cmultg.obj
❻ link386 @multiply.lnk
```

The sample is explained in detailed here:

1. Command 1 is the C compilation for the cmultipl.c source. The options specified for the compiler are:

| Compiler Options | Description |
|---|---|
| -O- | no optimize (recommended if -Ti used) |
| -Ti | general debugger symbols (recommended if debugger used) |
| -Ge- | build a DLL file (required) |
| -Gn+ | do not search libraries during compilation (required) |

| Compiler Options | Description |
|---|---|
| -Gs+ | omit stack checking (required) |
| -C | don't link after compile (required) |
| -I | directory for #include files |

No other options than those marked are required. Note also that the ordering of the options and operands is important and should be the same as in this example.

2. Command 2 is the SAS invocation that creates the program constants object. The global SAS program pgmcon.sas is used to generate the object. The -sysparm option contains a rather lengthy string of characters (broken up over several lines here for readability's sake) that instructs pgmcon.sas on generating the object. If you choose to avoid using `make` files, we recommend that you write your own SAS program to invoke PROC USERPROC for generating a program constants object.
3. Command 3 is the SAS command to build a grammar function. As with the program constants object, you can provide your own PROC USERPROC invocation in your own file if you do not wish to use `make`.
4. Command 4 compiles the grammar function generated by command 3. As in step 1, use the options shown in this example.
5. Command 5 is the LNKDEF command. This command is provided by SAS/TOOLKIT Software, and is responsible for dynamically building a LNK and DEF file for the LINK386 command. LNKDEF is provided with several arguments so that it has the sufficient information to create the LNK and DEF files. The files are created using the name of the module followed by the extension. In the above examples, the created files are MULTIPLY.LNK and MULTIPLY.DEF.
6. Command 6 is the LINK386 command that will use the LNK and DEF files. Note that LINK386 must be used. If you use LINK instead, the link will not be successful.

## Contents of the MULTIPLY.LNK file

The last statement in the command list for procedures refers to the MULTIPLY.LNK file, which is illustrated here:

```
/packdata +
/codeview +
/map +
/nod +
/linenumbers +
MCBMULTI +
CMULTIPL.OBJ +
CMULTG.OBJ +
toolkit-directory\GLOBAL\OBJ32\\prcintcv +
 +
,
.\MULTIPLY.DLL,
.\MULTIPLY.MAP,
\IBMC_32\LIB\DDE4SBS+\IBMC_32\LIB\DDE4SBSO+\SDK_20\OS2LIB\OS2386 +
toolkit-directory\GLOBAL\OBJ32\\toolkit,
```

```
MULTIPLY.DEF
```

### Contents of the MULTIPLY.DEF file

The last statement in the MULTIPLY.LNK file refers to the MULTIPLY.DEF file, which is illustrated here:

```
LIBRARY MULTIPLY INITINSTANCE
PROTMODE
EXPORTS MCB_MULTIPLY
```

## Command List for Compiling and Linking IFFCs

This section contains the set of commands generated by a `make` of the sample function, cfunc.c. This command set is very similar to the procedure command set, except that no grammar function is generated, compiled, or linked in. Note also that the IFFINT.OBJ object is linked, which is required of IFFCs written in C.

```
icc -Ge- -Gn+ -Gs+ -C -Itoolkit-directory\c\maclib\ -Focfunc.obj
    toolkit-directory\c\src\cfunc.c
sas toolkit-directory\global\test\pgmcon.sas
    -path toolkit-directory\global\load32\
    -sysparm *FUNCTION*uwutestf*mcbuwute*@*@*@*@toolkit-directory\c\src\*
             @toolkit-directory\global\grm\*@*c
lnkdef c IBM OS2_32 FUNCTION  @
    @\ibmc_32\lib\dde4sbs+
     \ibmc_32\lib\dde4sbso+
     \sdk_20\os2lib\os2386 @ @toolkit-directory\global\obj32\
    @ uwutestf mcbuwute cfunc.obj
link386 @uwutestf.lnk
```

### Contents of UWUTESTF.LNK

The last statement in the command list for IFFCs refers to UWUTESTF.LNK, which is illustrated here:

```
 /packdata +
 /codeview +
 /map +
 /nod +
 /linenumbers +
 MCBUWUTE +
 CFUNC.OBJ +
  +
 toolkit-directory\GLOBAL\OBJ32\\prcintcv +
 toolkit-directory\GLOBAL\OBJ32\\iffint +
  +
 ,
 .\UWUTESTF.DLL,
 .\UWUTESTF.MAP,
 \IBMC_32\LIB\DDE4SBS+\IBMC_32\LIB\DDE4SBSO+\SDK_20\OS2LIB\OS2386 +
 toolkit-directory\GLOBAL\OBJ32\\toolkit,
```

```
UWUTESTF.DEF
```

### Contents of UWUTESTF.DEF

The last statement in UWUTESTF.LNK refers to UWUTESTF.DEF, which is illustrated here:

```
LIBRARY UWUTESTF INITINSTANCE
PROTMODE
EXPORTS MCB_UWUTESTF
```

## Command List for Compiling and Linking Engines

This section contains the set of commands generated by a `make` of the sample engine, cengxmpl.c. This command set appears very similar to the procedure command set, except that no grammar function is generated, compiled, or linked in. Note also that the ENGINTCV.OBJ object is linked, which is required of engines written in C.

```
icc -Ge- -Gn+ -Gs+ -C -Itoolkit-directory\c\maclib\ -Focengxmpl.obj
    toolkit-directory\c\src\cengxmpl.c
sas toolkit-directory\global\test\pgmcon.sas
    -path toolkit-directory\global\load32\
    -sysparm *ENGINE*engxmpl*mcbengxm*@*@*@*@toolkit-directory\c\src\*
             @toolkit-directory\global\grm\*@*c
lnkdef c IBM OS2_32 ENGINE  @
      @\ibmc_32\lib\dde4sbs+
       \ibmc_32\lib\dde4sbso+
       \sdk_20\os2lib\os2386 @ @toolkit-directory\global\obj32\
      @ engxmpl mcbengxm cengxmpl.obj
link386 @engxmpl.lnk
```

### Contents of ENGXMPL.LNK

The last statement in the command list for engines refers to ENGXMPL.LNK, which is illustrated here:

```
/packdata +
/codeview +
/map +
/nod +
/linenumbers +
MCBENGXM +
CENGXMPL.OBJ +
 +
toolkit-directory\GLOBAL\OBJ32\\prcintcv +
toolkit-directory\GLOBAL\OBJ32\\engintcv +
 +
,
.\ENGXMPL.DLL,
.\ENGXMPL.MAP,
\IBMC_32\LIB\DDE4SBS+\IBMC_32\LIB\DDE4SBSO+\SDK_20\OS2LIB\OS2386 +
toolkit-directory\GLOBAL\OBJ32\\toolkit,
```

```
ENGXMPL.DEF
```

### Contents of ENGXMPL.DEF

The last statement in ENGXMPL.LNK refers to ENGXMPL.DEF, which is illustrated here:

```
LIBRARY ENGXMPL INITINSTANCE
PROTMODE
EXPORTS MCB_ENGXMPL
```

## Directory Structure

The SAS/TOOLKIT directories are cascaded from a single directory. The top level directory name for SAS/TOOLKIT software is determined by the person installing the software at your site. In most cases, the names of the directories below the top directory are the same as the ones described in the remainder of this appendix.

In the top-level SAS/TOOLKIT directory are two subdirectories, GLOBAL and C. GLOBAL contains all the subdirectories that contain global material used by any language (at this time, C is the only supported language). The C directory contains all the subdirectories that contain code specific to the C language or #include files.

## GLOBAL Directory

The GLOBAL directory contains the following subdirectories:

- TEST
- GRM
- OBJ

The following tables describe the contents of each subdirectory.

The TEST directory contains all the SAS programs used to test the sample executables. Also included are SAS programs invoked during the `make` process. Table A1.1 lists the files in this directory.

**Table A1.1**
*Contents of TEST Subdirectory*

| SAS Program | Description |
|---|---|
| DSFUNC.SAS | Tests UWUDSFUN |
| ENGXMPL.SAS | Tests ENGXMPL (read-only) |
| ENGXMPLW.SAS | Tests ENGXMPLW (write) |
| EXAMPLE.SAS | Tests PROC EXAMPLE |
| EXTRACT.SAS | Tests PROC EXTRACT |
| FTNPREP.SAS | FORTRAN preprocessor (not used) |
| GEXAMPLE.SAS | Tests PROC GEXAMPLE |
| GFRAC.SAS | Tests PROC GFRAC |

| SAS Program | Description |
|---|---|
| GPOP.SAS | Tests PROC GPOP |
| GRMFUNC.SAS | Used by `make` for building grammar functions |
| IMLFUNC.SAS | Tests UWUIMLEX |
| MULTIPLY.SAS | Tests PROC MULTIPLY |
| OPENTST.SAS | Tests PROC OPENTST |
| PGMCON.SAS | Used by `make` for building program constants |
| SIMPLE.SAS | Tests PROC SIMPLE |
| SUBLIB.SAS | Tests PROC SUBLIB |
| TLKTFMT.SAS | Tests UWFONEOF |
| TLKTFUNC.SAS | Tests UWUTESTF |
| TLKTIFMT.SAS | Tests UWITESTI |
| UPARSE.SAS | Tests PROC UPARSE |

The GRM directory contains all the sample grammars, plus the stub that's used by any user-written procedure. Table A1.2 list the files in this directory.

**Table A1.2**
*Contents of GRM Subdirectory*

| Grammar File | Description |
|---|---|
| EXAMPLE.GRM | Grammar for PROC EXAMPLE |
| EXTRACT.GRM | Grammar for PROC EXTRACT |
| GEXAMPLE.GRM | Grammar for PROC GEXAMPLE |
| GFRAC.GRM | Grammar for PROC GFRAC |
| GPOP.GRM | Grammar for PROC GPOP |
| MULTIPLY.GRM | Grammar for PROC MULTIPLY |
| OPENTST.GRM | Grammar for PROC OPENTST |
| SIMPLE.GRM | Grammar for PROC SIMPLE |
| SUBLIB.GRM | Grammar for PROC SUBLIB |
| UPARSE.GRM | Grammar for PROC UPARSE |
| STUBGRM.GRM | Grammar stub |

The OBJ directory contains all the objects necessary to compile and link SAS/TOOLKIT applications. Table A1.3 lists the files in this directory.

**Table A1.3**
*Contents of OBJ Subdirectory*

| Object File | Description |
|---|---|
| ENGINTCV.OBJ | Object linked in for engines |
| IFFINT.OBJ | Object linked in for IFFCs |
| PRCINTCV.OBJ | Proc interface object (always linked) |
| TOOLKIT.LIB | Object library (always referenced) |

## C Directory

The C subdirectory contains the following subdirectories:

- SRC
- OBJ
- LOAD
- CNTL
- MACLIB

The contents of each of these subdirectories is discussed below.

The SRC subdirectory contains all the source code for the sample applications written in the C language. Table A1.4 lists the files in this directory.

**Table A1.4**
*Contents of SRC Subdirectory*

| Source File | Description |
|---|---|
| CDSFUNC.C | Data step function example |
| CENGXMPL.C | Engine example |
| CEXAMPLE.C | PROC EXAMPLE |
| CEXMPLG.C | PROC EXAMPLE grmfunc |
| CEXTRACG.C | PROC EXTRACT grmfunc |
| CEXTRACT.C | PROC EXTRACT |
| CFMT.C | Sample formats |
| CFUNC.C | Sample functions |
| CGEXAMPG.C | PROC GEXAMPLE grmfunc |
| CGEXAMPL.C | PROC GEXAMPLE |
| CGFRAC.C | PROC GFRAC |
| CGFRACG.C | PROC GFRAC grmfunc |
| CGPOP.C | PROC GPOP |
| CGPOPG.C | PROC GPOP grmfunc |
| CIMLEXM1.C | IML function example |

| Source File | Description |
|---|---|
| CINFMT.C | Sample informats |
| CMULTG.C | PROC MULTIPLY grmfunc |
| CMULTIPL.C | PROC MULTIPLY |
| COPENTSG.C | PROC OPENTST grmfunc |
| COPENTST.C | PROC OPENTST |
| CSIMPLE.C | PROC SIMPLE |
| CSIMPLEG.C | PROC SIMPLE grmfunc |
| CSUBLIB.C | PROC SUBLIB |
| CSUBLIBG.C | PROC SUBLIB grmfunc |
| CUPARSE.C | PROC UPARSE |
| CUPARSEG.C | PROC UPARSE grmfunc |

The OBJ subdirectory contains the compiled versions of the source code in the SRC directory, plus the program constants objects. Table A1.5 lists the files in this directory.

**Table A1.5**
*Contents of OBJ Subdirectory*

| Object File | Description |
|---|---|
| CDSFUNC.OBJ | DATA step function |
| CENGXMPL.OBJ | Engine example |
| CEXAMPLE.OBJ | PROC EXAMPLE |
| CEXMPLG.OBJ | PROC EXAMPLE grmfunc |
| CEXTRACG.OBJ | PROC EXTRACT grmfunc |
| CEXTRACT.OBJ | PROC EXTRACT |
| CFMT.OBJ | Sample formats |
| CFUNC.OBJ | Sample functions |
| CGEXAMPG.OBJ | PROC GEXAMPLE grmfunc |
| CGEXAMPL.OBJ | PROC GEXAMPLE |
| CGFRAC.OBJ | PROC GFRAC |
| CGFRACG.OBJ | PROC GFRAC grmfunc |
| CGPOP.OBJ | PROC GPOP |
| CGPOPG.OBJ | PROC GPOP grmfunc |
| CIMLEXM1.OBJ | IML function example |
| CINFMT.OBJ | Sample informats |
| CMULTG.OBJ | PROC MULTIPLY grmfunc |

| Object File | Description |
|---|---|
| CMULTIPL.OBJ | PROC MULTIPLY |
| COPENTSG.OBJ | PROC OPENTST grmfunc |
| COPENTST.OBJ | PROC OPENTST |
| CSIMPLE.OBJ | PROC SIMPLE |
| CSIMPLEG.OBJ | PROC SIMPLE grmfunc |
| CSUBLIB.OBJ | PROC SUBLIB |
| CSUBLIBG.OBJ | PROC SUBLIB grmfunc |
| CUPARSE.OBJ | PROC UPARSE |
| CUPARSEG.OBJ | PROC UPARSE grmfunc |
| MCBENGXM.OBJ | program constants object for ENGXMPL |
| MCBEXAMP.OBJ | program constants object for PROC EXAMPLE |
| MCBEXTRA.OBJ | program constants object for PROC EXTRACT |
| MCBGEXAM.OBJ | program constants object for PROC GEXAMPLE |
| MCBGFRAC.OBJ | program constants object for PROC GFRAC |
| MCBGPOP.OBJ | program constants object for PROC GPOP |
| MCBMULTI.OBJ | program constants object for PROC MULTIPLY |
| MCBOPENT.OBJ | program constants object for PROC OPENTST |
| MCBSIMPL.OBJ | program constants object for PROC SIMPLE |
| MCBSUBLI.OBJ | program constants object for PROC SUBLIB |
| MCBUPARS.OBJ | program constants object for PROC UPARSE |
| MCBUWFON.OBJ | program constants object for UWFONEOF |
| MCBUWITE.OBJ | program constants object for UWITESTI |
| MCBUWUDS.OBJ | program constants object for UWUDSFUN |
| MCBUWUIM.OBJ | program constants object for UWUIMLEX |
| MCBUWUTE.OBJ | program constants object for UWUTESTF |

The LOAD subdirectory contains the DLL files that are the result of linking the objects in OBJ. Table A1.6 lists the files in this directory.

**Table A1.6**
*Contents of LOAD Subdirectory*

| DLL File | Description |
|---|---|
| ENGXMPL.DLL | Sample engine |
| EXAMPLE.DLL | PROC EXAMPLE |
| EXTRACT.DLL | PROC EXTRACT |

| DLL File | Description |
|---|---|
| GEXAMPLE.DLL | PROC GEXAMPLE |
| GFRAC.DLL | PROC GFRAC |
| GPOP.DLL | PROC GPOP |
| MULTIPLY.DLL | PROC MULTIPLY |
| OPENTST.DLL | PROC OPENTST |
| SIMPLE.DLL | PROC SIMPLE |
| SUBLIB.DLL | PROC SUBLIB |
| UPARSE.DLL | PROC UPARSE |
| UWFONEOF.DLL | Sample formats |
| UWITESTI.DLL | Sample informats |
| UWUDSFUN.DLL | Sample data step functions |
| UWUIMLEX.DLL | Sample IML functions |
| UWUTESTF.DLL | Sample functions |

The CNTL subdirectory contains the `make` files for compiling and linking the examples. Files described as "general" `make` files are not specific to the samples and can be used by your application. Table A1.7 lists the files in this directory.

**Table A1.7**
*Contents of CNTL Subdirectory*

| **`make` File** | Description |
|---|---|
| ENGJOBC1.MAK | sample engine |
| GRMFUNC.MAK | general `make` file for grammar functions |
| IFFJOB2C.MAK | sample data step functions |
| IFFJOBFC.MAK | sample formats |
| IFFJOBIC.MAK | sample informats |
| IFFJOBUC.MAK | sample functions |
| IMLJOBC1.MAK | sample IML functions |
| PGMCON.MAK | general `make` file for program constants |
| PRCJOBC1.MAK | PROC EXAMPLE |
| PRCJOBC2.MAK | PROC EXTRACT |
| PRCJOBC3.MAK | PROC GEXAMPLE |
| PRCJOBC4.MAK | PROC GFRAC |
| PRCJOBC5.MAK | PROC GPOP |
| PRCJOBC6.MAK | PROC MULTIPLY |
| PRCJOBC7.MAK | PROC SIMPLE |

| **make File** | **Description** |
|---|---|
| PRCJOBC8.MAK | PROC SUBLIB |
| PRCJOBC9.MAK | PROC UPARSE |
| PRCJOBCA.MAK | PROC OPENTST |
| TLKTHOST.MAK | general `make` file with OS/2 2.0 information |
| TOOLKIT.MAK | general `make` file not OS/2 specific |
| UWLINK.MAK | general `make` file for linking all |
| UWLNKENG.MAK | general `make` file for linking engines |
| UWLNKIFF.MAK | general `make` file for linking IFFCs |
| UWLNKPRC.MAK | general `make` file for linking procs |
| WHERE.MAK | sample `make` file defining directories |

The MACLIB subdirectory contains the `#include` files used by SAS/TOOLKIT applications. Table A1.8 lists the files in this directory.

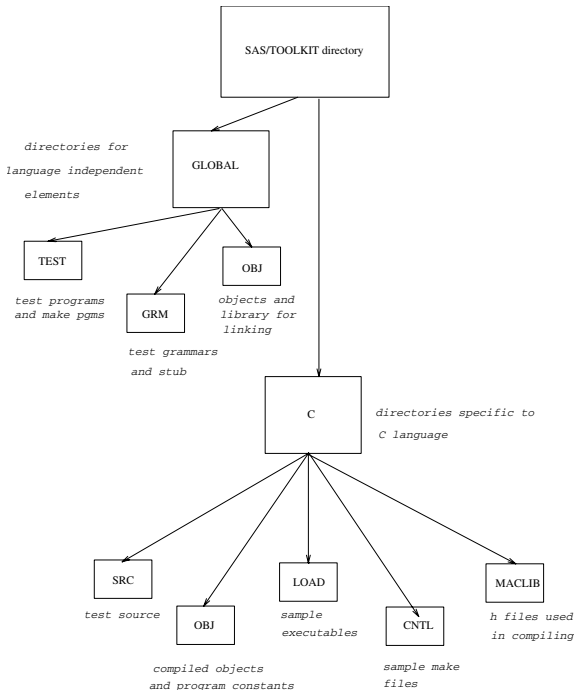
**Table A1.8**
*Contents of MACLIB Subdirectory*

| **Include File** | **Description** |
|---|---|
| ENGDEF.H | used by engines |
| GRAMFUNC.H | used by grammar functions |
| UWHOST.H | host-specific information included by uwproc.h |
| UWPROC.H | used by ALL applications |

## Summary of Directory Structure

Figure A1.1 illustrates the directory structure described in the previous sections.

**Figure A1.1**
*Summary of Directory Structures*




## Note on Using the IBM Set/2 Debugger

You are likely to experience problems if you use the IBM Set/2 debugger for OS/2 2.0 with user-written SAS modules. Current testing at SAS Institute indicates that if you set breakpoints to debug a user-written SAS module, when you exit from the debugger, your system will lock. You will have to reboot the system.

# Appendix 2 Using SAS/TOOLKIT® Software Under UNIX Systems



## Introduction

This appendix provides sample `make` files for compiling and linking your procedure, function, CALL routine, informat, format, or engine. On the UNIX implementations (HP-UX, AIX, and SunOS) that SAS/TOOLKIT software supports, you can create procedures, IFFCs, and engines using the native C compiler or native FORTRAN compiler. In addition, this appendix discusses how to use user-written SAS modules after they are compiled and linked and how to make those modules available to other sites.

## Accessing User-written SAS Modules

This section discusses how to access user-written modules and how to make these modules available to other sites.

## Specifying Where to Find Modules

Procedures, IFFCs, and engines are executable modules that are stored as executable images. They have a one-level name. Images supplied by SAS Institute usually have a `sas` or `sab` prefix, but user-written images will not.

In order to access your user-written module, you must indicate its location with a -path option that is provided to the SAS command. This option can be specified on the command line, or placed into a config.sas file. The value of the -path option should be the name of the directory in which the executable resides. If you have multiple executables in multiple directories, you can issue the -path option multiple times.

For example, suppose your user-written procedure abc is located in the directory /usr/written/procs. To access the procedure, use the following command:

```
sas -path /usr/written/procs
```

If you have procedure def in /usr/written/procs/special, and wish to access both abc and def in the same SAS job, the command would be:

```
sas -path /usr/written/procs -path /usr/written/procs/special
```

Files in `/usr/written/procs` will be accessed before files in `/usr/written/procs/special`.

Setting up the -path options in the config.sas file will be especially convenient if there are several paths to include. Refer to the discussion of the -PATH SAS system option in Chapter 8 of *SAS Companion for the UNIX Environment and Derivatives, Version 6, First Edition*.

## Distributing Modules to Other SAS Sites

You can distribute a module you have written to other SAS Sites that are running the same version of the SAS System under the same UNIX operating system. The site receiving your module does not need to license SAS/TOOLKIT Software in order to run your module.

You can provide the module to your off-site users by any traditional file transferral mechanism, such as tape or by the compress/uuencode/email/uudecode/uncompress process. However, you must send the the module as an executable, because the site will not be permitted to recompile or relink your code unless it also licenses SAS/TOOLKIT software.

# Creating Executable Modules

Executable modules are created on the UNIX platforms by using the compiler of choice to compile the necessary components, then using the system linker (ld) to link the objects into an executable image. On UNIX platforms, this process is usually handled by the `make` facility, which greatly simplifies the software development effort. Most of the discussion in this appendix concerns the use of `make` to create your executables. This appendix assumes that you have a working knowledge of `make`. If you are not familiar with the `make` facility, we recommend that you consult your UNIX documentation to learn more about it before attempting to build your SAS/TOOLKIT executables. "Compiling and Linking C Programs without make Files" on page 136 and "Compiling and Linking FORTRAN Programs without make Files" on page 139 list the commands that `make` invokes to create the objects and executables. If you choose to avoid using `make`, you can model your commands after

those discussed.

This example illustrates using `make` to compile a C program:

```
make -f prcjobc6.mak
```

▶ ***Caution*** . . . . . ***Review warning messages carefully.***
There are many warnings that come out of the FORTRAN compiler on some UNIX platforms when a routine is declared but not used. Including the fprotosf.txt file causes the declarations for all interface routines that return values (vs. call routines that don't return anything). Therefore, if your program includes this file, you will receive a large number of warnings about all of the routines that are declared but not used. These warnings will not affect your program, but they may obscure other warnings that you need to see to correct an actual problem in the program. ▲

## The make Files

Each sample procedure, IFFC, and engine has a `make` file associated with it. All the sample `make` files first include the global `make` file toolkit.make.

## The toolkit.make File

The toolkit.make file contains all the macro definitions that are UNIX-specific, but are system independent.

```
########################################################################
# Copyright (C) 1992 by SAS Institute Inc., Cary NC USA 27512-8000    #
# NAME:    toolkit.make                                               #
# PRODUCT: SAS/TOOLKIT                                                #
# PURPOSE: standard include file for all make files on UNIX           #
########################################################################
#
include tlkthost.make

# UNIX-portable macros

PRCINTCV        = $(TLKTDIR)/prcintcv.o
PRCINTCA        = $(TLKTDIR)/prcintca.o
IFFINT          = $(TLKTDIR)/iffint.o
ENGINTCV        = $(TLKTDIR)/engintcv.o
ENGINTCA        = $(TLKTDIR)/engintca.o
OBJLIB          = $(TLKTDIR)/toolkit.a
SASCMDGRM       = $(SASCMD) $(TLKTSAMP)grmfunc.sas
                  -sysparm $(SYSPARM)
MAKE_PGMCON     = ; $(SASCMD) $(TLKTSAMP)pgmcon.sas
                  -sysparm $(SYSPARM)
FTNPREP_CMD     = $(SASCMD) $(TLKTSAMP)ftnprep.sas
                  -sysparm
MAKE_GRMSRC     = $(GRM)$(MODNAME).grm ; $(SASCMDGRM)
MAKE_GRMOBJ     = $(SRC)$(FUNCFILE).c $(UWH) ; $(COMPILE_GRMFUNC)
MAKE_GRMOBJ_FTN = $(FUNCFILE).$(FSUF) ; $(COMPILE_GRMFUNC_FTN)
SYSPARM         = $(SYSPARM1)$(SYSPARM2)$(SYSPARM3)
```

```
SYSPARM1        = $(WHICH)*$(MODNAME)*$(PGMCON)*@
SYSPARM2        = $(FUNCFILE)*@$(GRAMFUNC)*@$(GRM)*@
SYSPARM3        = $(SRC)*@$(INCLLIB)*@$(OBJ)*$(LANGUAGE)
PGMCONIFF       = -o $(LOAD)$(MODNAME) $(UWLINKIFF)
PGMCONENG       = -o $(LOAD)$(MODNAME) $(UWLINKENG)
PGMCONPROC      = -o $(LOAD)$(MODNAME) $(UWLINKPROC)
PROCLINKCMD     = $(UWLINK) $(PGMCONPROC) $(BASEOBJ)
                  $(UWPROCOBJ) $(PRCINT) $(OBJLIB) $(UWLINKOPTS)
IFFLINKCMD      = $(UWLINK) $(PGMCONIFF) $(BASEOBJ)
                  $(UWIFFOBJ) $(PRCINT) $(IFFINT) $(OBJLIB)
                  $(UWLINKOPTS)
ENGLINKCMD      = $(UWLINK) $(PGMCONENG) $(BASEOBJ)
                  $(UWENGOBJ) $(PRCINT) $(ENGINT) $(OBJLIB)
                  $(UWLINKOPTS)
COMPILE_GRMFUNC = $(UWC) $(SRC)$(FUNCFILE).c $(UWCOPTS) ;
                  $(MOVE_GRMFUNC_OBJ)
COMPILE_GRMFUNC_FTN = $(UWFORT) $(FUNCFILE).$(FSUF) $(UWFORTOPTS) ;
                  $(MOVE_GRMFUNC_OBJ)
PROC_LINK       = $(UWPROCOBJ) ; $(PROCLINKCMD)
IFF_LINK        = $(UWIFFOBJ) ; $(IFFLINKCMD)
ENG_LINK        = $(UWENGOBJ) ; $(ENGLINKCMD)
UWPROCOBJ       = $(OBJ)$(PGMCON).o $(PROCOBJ) $(OBJ)$(FUNCFILE).o
UWIFFOBJ        = $(OBJ)$(PGMCON).o $(IFFOBJ)
UWENGOBJ        = $(OBJ)$(PGMCON).o $(ENGOBJ)
```

## The tlkthost.make File

The toolkit.make file includes the tlkthost.make file, which contains the system-specific information. You must alter the tlkthost.make file to reflect the correct system-specific information.

**Note:** The term *toolkit-dir* is used in these command lists to represent the top-level directory where the SAS/TOOLKIT product is stored on your system and *your-dir* represents your development directory.

The tlkthost.make file supplied for the HP-UX system is as follows:

```
########################################################################
# Copyright (C) 1992 by SAS Institute Inc., Cary NC USA 27512-8000    #
# NAME:    tlkthost.make                                              #
# PRODUCT: SAS/TOOLKIT                                                #
# PURPOSE: host-specific make file for HP-UX                          #
########################################################################

(1) UWC             = /bin/cc
UWFORT          = f77 -U
UWLINK          = /usr/local/bin/ld \
                  -u calloc -u malloc -u realloc -u free
UWCOPTS         = -g -c -I$(MACLIB) -DHPUX
UWFORTOPTS      = -g -c -I$(MACLIB)
UWLINKOPTS      = +n -R80000 -HF -N -z -a archive -lm -lc -lf
UWLINKIFF       = -e iffint_pgmcon
UWLINKENG       = -e engint_pgmcon
UWLINKPROC      = -e $(MODNAME)_pgmcon
```

```
BASEOBJ         = $(TLKTDIR)/vicrt.o
BASEOBJF        = /lib/crt0.o $(TLKTDIR)/main.o
FSUF            = f

# Institute-specific names
# should be modified per site

❷ INCLLIB         = toolkit-dir/global/grm
TLKTDIR         = toolkit-dir/global/obj
TLKTSAMP        = toolkit-dir/global/test/
SASCMD          = sas

MACLIB = .
```

1. The first section of the tlkthost.make file contains system-specific information that you will most likely not change. For example, the UWC macro provides the correct command for the C compiler. If your C compiler is located somewhere other than /bin/cc, then you would have to change tlkthost.make to reflect this. You may possibly need to change the UWFORT definition, the location of the ld command in UWLINK, and/or the location of crt0.o in BASEOBJF if it is not located in /lib. Otherwise, do not change anything else in the first section.
2. The second section consists of directory names that will be site- specific. Check with your SAS Software Representative to determine where these files are installed at your site. The macros and their meanings are:

| Macros | Description |
|---|---|
| INCLLIB | Location of the stub.grm grammar file |
| TLKTDIR | Location of the SAS/TOOLKIT global object directory |
| TLKTSAMP | Location of the SAS/TOOLKIT ftnprep, pgmcon, and grmfunc SAS programs |
| SASCMD | Your local SAS command |
| MACLIB | This macro should not be changed. |

## Setting Up a make File to Use

The best way to understand how to set up a `make` file for building a user-written module is to use a sample `make` file. Consider this `make` file for creating PROC MULTIPLY using the C language:

```
######################################################################
# Copyright (C) 1992 by SAS Institute Inc., Cary NC USA 27512-8000    #
# NAME:    prcjobc6.make                                              #
# PRODUCT: SAS/TOOLKIT                                                #
# PURPOSE: sample make file for creating PROC MULTIPLY on UNIX        #
######################################################################
❶ include toolkit.make
```

```
❷ # local macros
  PROCOBJ         = $(OBJ)cmultipl.o
  MODNAME         = multiply
  FUNCFILE        = cmultg
  GRAMFUNC        = multg
  PGMCON          = mcbmulti
  WHICH           = PROC
  LANGUAGE        = c
  PRCINT          = $(PRCINTCV)

❸ include where.make

❹ # local MAKE definitions
  ALL             : $(LOAD)$(MODNAME)
❺ $(OBJ)cmultipl.o      : $(SRC)cmultipl.c ; \
                    $(UWC) $(SRC)cmultipl.c $(UWCOPTS) ;\
                    mv cmultipl.o $(OBJ)

❻ include grmfunc.make
  include pgmcon.make
  include uwlnkprc.make
```

1. The first statement of the `make` file is the include for toolkit.make. This include **must** be present for the `make` file to work correctly.
2. The next section of the `make` file consists of local macro definitions. These macros provide the necessary information to allow the packaged macros of toolkit.make to work correctly. Here is a list of the macros that can be defined:

| Macro | Used by | Description |
|---|---|---|
| PROCOBJ | procs | lists all objects for which you will be providing the source |
| IFFOBJ | IFFCs | lists all objects for which you will be providing the source |
| ENGOBJ | engines | lists all objects for which you will be providing the source |
| MODNAME | all | the name of the executable and the name of the grammar file (which will have a .grm extension) |
| FUNCFILE | procs | the name of the C file that will contain your grammar function |
| GRAMFUNC | procs | the name of the function as it is called from your proc source |
| PGMCON | all | the name of your program constants object |
| WHICH | all | describes which type of module is being created: PROC, FUNCTION, FORMAT, INFORMAT, ENGINE |
| LANGUAGE | all | c or fortran |

| Macro | Used by | Description |
|---|---|---|
| PRCINT | all | $(PRCINTCV) if using C $(PRCINTCA) if using FORTRAN |
| ENGINT | engines | $(ENGINTCV) if using C $(ENGINTCA) if using FORTRAN |

3. The next statement after the macro definitions is an include for where.make. The where.make file describes the location of the source, object, maclib, and load (executable) directories. For example:

```
SRC             = your-dir/src/
OBJ             = your-dir/obj/
MACLIB          = your-dir/maclib/
LOAD            = your-dir/load/
GRM             = $(INCLLIB)/
MOVE_GRMFUNC_OBJ= mv $(FUNCFILE).o $(OBJ)
```

   Providing SRC, OBJ, MACLIB, LOAD, and GRM allows you to separate your differing file types. If you don't see a need for this, simply omit the include of where.make, or provide a where.make file with no definitions. If you do use definitions in where.make, be sure to use the MOVE_GRMFUNC_OBJ definition as seen above.

4. The next item in the sample `make` file is the local definitions. The first definition is ALL, and it should always refer to $(LOAD)$(MODNAME). This will expand into the executable name that is to be created by `make`. (Note that if you choose to not provide a definition for LOAD in the where.make file, it will reduce to null and only the executable name will appear in the ALL definition, meaning that the resulting executable will be placed in the currect directory).

   The ALL target is used to build all the pieces of the module; this target's dependency is the load module itself. Further target definitions (like the ones stating dependencies on grammar files) are defined when the other `make` files are included. See number 6.

5. After the ALL definition, specify the definitions and dependencies, that is the `make` rules, for all objects that were listed in the PROCOBJ, IFFOBJ, or ENGOBJ macro. This example has one object:

```
$(OBJ)cmultipl.o      : $(SRC)cmultipl.c ; \
                  $(UWC) $(SRC)cmultipl.c $(UWCOPTS) ;\
                  mv cmultipl.o $(OBJ)
```

   This statement indicates that the cmultiple.o object is dependent on the cmultipl.c source. The object is generated by two commands. The first is the C compilation of the source using the required options. The second command is a move to the proper object directory. (Omit this second command if the where.make file does not provide a definition for OBJ).

   For FORTRAN programs, this command should be be as follows:

```
$(OBJ)fmultipl.o      : $(SRC)fmultipl.for ; \
                  $(UWFORT) $(SRC)fmultipl.for $(UWFORTOPTS) ;\
                  mv fmultipl.o $(OBJ)
```

6. After your object definitions, you should provide include statements depending upon what your module is doing. If you are writing a procedure, you must add the following line:

```
include grmfunc.make
```

This will include the definitions and dependencies to allow the grammar function to be created and compiled from the grammar source.

For all applications, you must add the following line:

```
include pgmcon.make
```

This will include the definitions and dependencies to create the program constants object.

The last include line depends on the type of module being created:

| To Include . . . | Specify this statement |
|---|---|
| procs | `include uwlnkprc.make` |
| IFFCs | `include uwlnkiff.make` |
| engines | `include uwlnkeng.make` |

The uwlnkxxx.make file is responsible for ensuring that the executable is linked using the program constants object, the grammar function object (procs only), the objects in the PROCOBJ/IFFOBJ/ENGOBJ list, and the SAS/TOOLKIT objects and library.

## Sample make File for Functions

```
########################################################################
# Copyright (C) 1992 by SAS Institute Inc., Cary NC USA 27512-8000    #
# NAME:    iffjobuc.make                                              #
# PRODUCT: SAS/TOOLKIT                                                #
# PURPOSE: sample make file for creating the sample functions         #
########################################################################
include toolkit.make

# local macros

IFFOBJ          = $(OBJ)cfunc.o
UWH             = $(UWPROCH)
MODNAME         = uwutestf
PGMCON          = mcbuwute
WHICH           = FUNCTION
LANGUAGE        = c
PRCINT          = $(PRCINTCV)

include where.make

# local MAKE definitions
```

```
ALL             : $(LOAD)$(MODNAME)
$(OBJ)cfunc.o   : $(SRC)cfunc.c $(UWH); \
                  $(UWC) $(SRC)cfunc.c $(UWCOPTS) ;\
                  mv cfunc.o $(OBJ)

include pgmcon.make
include uwlnkiff.make
```

## Sample make File for Informats

```
########################################################################
# Copyright (C) 1992 by SAS Institute Inc., Cary NC USA 27512-8000    #
# NAME:    iffjobic.make                                              #
# PRODUCT: SAS/TOOLKIT                                                #
# PURPOSE: sample make file for creating the sample informats         #
########################################################################
include toolkit.make

# local macros

IFFOBJ          = $(OBJ)cinfmt.o
MODNAME         = uwitesti
PGMCON          = mcbuwite
WHICH           = INFORMAT
LANGUAGE        = c
PRCINT          = $(PRCINTCV)

include where.make

# local MAKE definitions
ALL             : $(LOAD)$(MODNAME)
$(OBJ)cinfmt.o  : $(SRC)cinfmt.c ; \
                  $(UWC) $(SRC)cinfmt.c $(UWCOPTS) ;\
                  mv cinfmt.o $(OBJ)

include pgmcon.make
include uwlnkiff.make
```

## Sample make File for Engines

```
########################################################################
# Copyright (C) 1992 by SAS Institute Inc., Cary NC USA 27512-8000    #
# NAME:    engjobc1.make                                              #
# PRODUCT: SAS/TOOLKIT                                                #
# PURPOSE: sample make file for creating the sample engine            #
########################################################################
include toolkit.make

# local macros

ENGOBJ          = $(OBJ)cengxmpl.o
MODNAME         = engxmpl
```

```
PGMCON          = mcbengxm
WHICH           = ENGINE
LANGUAGE        = c
PRCINT          = $(PRCINTCV)
ENGINT          = $(ENGINTCV)

include where.make

# local MAKE definitions
ALL             : $(MODNAME)
$(OBJ)cengxmpl.o   : $(SRC)cengxmpl.c ; \
                  $(UWC) $(SRC)cengxmpl.c $(UWCOPTS) ;\
                  mv cengxmpl.o $(OBJ)

include pgmcon.make
include uwlnkeng.make
```

## Running Your make File

After you have created your `make` file, run it in your development directory. First, copy tlkthost.make from the c/cntl directory under the top-level SAS/TOOLKIT directory in your directory. You will also need copies of any other `make` files your are including, such as where.make and pgmcon.make. When you have all the `make` files in place, you can invoke `make` as follows:

```
make -f yourfile.make
```

If all is correct, `make` recreates any of the components that have changed since you last ran `make`, and finally links the executable.

For a list of the actual commands produced by `make`, see the discussion in {lqd}Compiling and Linking C Programs without make Files{rdq} and "Compiling and Linking FORTRAN Programs without make Files" on page 139 depending on which language you are using.

# Compiling and Linking C Programs without make Files

This section lists all of the commands generated by `make` for the C version of the MULTIPLY procedure under HP-UX. If you are using `make`, you do not need to know the detailed information listed in this part of the appendix. However, if you are not using `make`, you can use the contents of these command lists to model your own process for compiling and linking.

## Command List for Procedures in C

**Note:** The linker command in this command list is specifically for the HP-UX platform. Alternate linker commands for AIX and SunOS are provided in the description of number 7.

❶ `sas` *`toolkit-dir`*`/global/test/pgmcon.sas`
`-sysparm PROC*multiply*mcbmulti*@cmultg*@multg*`

```
                @toolkit-dir/global/grm/*@your-dir/src/*
                @toolkit-dir/global/grm*@your-dir/obj/*c
❷ /bin/cc your-dir/src/cmultipl.c -g -c -Iyour-dir/maclib/ -DHPUX
❸ mv cmultipl.o your-dir/obj/
❹ sas toolkit-dir/global/test/grmfunc.sas
      -sysparm PROC*multiply*mcbmulti*@cmultg*@multg*
                @toolkit-dir/global/grm/*@your-dir/src/*
                @toolkit-dir/global/grm*@your-dir/obj/*c
❺ /bin/cc your-dir/src/cmultg.c -g -c -Iyour-dir/maclib/ -DHPUX
❻ mv cmultg.o your-dir/obj/
❼ /usr/local/bin/ld  -u calloc -u malloc -u realloc -u free
      -o your-dir/load/multiply -e multiply_pgmcon
❽     toolkit-dir/global/obj/vicrt.o
❾     your-dir/obj/mcbmulti.o your-dir/obj/cmultipl.o your-dir/obj/cmultg.o
❿     toolkit-dir/global/obj/prcintcv.o
⓫     toolkit-dir/global/obj/toolkit.a
⓬     +n -R80000 -HF -N -z -a archive -lm -lc -lf
```

1. This command is the SAS invocation that creates the program constants object. The global SAS program pgmcon.sas is used to generate the object. The -sysparm option contains a rather lengthy string of characters (broken up over several lines here) that instructs pgmcon.sas on generating the object. If you choose to avoid using `make` files, you need to write your own SAS program to invoke PROC USERPROC for generating a program constants object.

2. This command compiles the sample C source code. Note that the options -g (debug), -c (no link), -I (include file directory) and -DHPUX (define HPUX symbol) are provided. The -g option can be omitted, since the HP-UX debugger currently does not support dynamically loaded modules (as all user-written procedures are characterized). The -c option **must** be specified, since the link must be performed after all other components are built. The -I option can be omitted if all #include files are in the same directory. The -DHPUX should be provided so that some HP-UX-specific items can be properly set during the compile.

3. This command moves the object from the current directory to the object directory. If you are performing all processing in the same directory, this command can be omitted. Some UNIX platforms support the -o option in their cc command for writing out an object to a specified file (optionally in another directory). But since some do not, this command provides the capability for all platforms.

4. This command is the SAS command to build a grammar function. As with the program constants object, you can provide your own PROC USERPROC invocation in your own file if you do not wish to use `make`.

5. This command compiles the grammar function generated by command 4.

6. This command copies the object as in command 3.

7. This command is the link. The four -u options must be present. The -o is required and specifies the output object. The -e option supplies the entry name, which will be xxxxxxxx_pgmcon, where xxxxxxxx is your module name.

   The comparable command for AIX systems is shown here:

```
/bin/ld -u calloc -u malloc -u realloc -u free \
       -H512 -T512 -b import:/lib/syscalls.exp \
       -b glink:/lib/glink.o -b noloadmap -b h:4
```

```
        objects and libraries \
         -emcn_main -lm -lc -lxlf
```

This is the comparable linker command for SunOS systems:

```
/bin/ld -r -p -u _calloc -u _malloc -u _free -u _realloc \
       objects and libraries \
       -e _mcn_main -d -dc -dp -X -lm /usr/lib/libc.a \
       /usr/lang/SC1.0/libF77.a
```

8. This line includes a required object that is supplied as part of SAS/TOOLKIT Software.
9. This line contains the program constants object, the procedure object, and the grammar function object.
10. This line contains the required procedure interface object, prcintcv (PRoCedure INTerface Call-by-Value).
11. This line is the object library for additional resolutions (also supplied with SAS/TOOLKIT software).
12. These options to the linker are all required. They should not be altered. The ordering of the options and operands is important, and should be as indicated.

## Command List for IFFCs in C

This section contains the set of commands generated by a `make` of the sample function, cfunc.c. This command set is very similar to the procedure command set, except that no grammar function is generated, compiled, or linked in. Note also that the iffint.o object is linked, which is required of IFFCs written in C.

```
sas toolkit-dir/global/test/pgmcon.sas
       -sysparm FUNCTION*uwutestf*mcbuwute*@*@*
                @toolkit-dir/global/grm/*@your-dir/src/*
                @toolkit-dir/global/grm*@your-dir/obj/*c
/bin/cc your-dir/src/cfunc.c -g -c -Iyour-dir/maclib/ -DHPUX
mv cfunc.o your-dir/obj/
/usr/local/bin/ld  -u calloc -u malloc -u realloc -u free
       -o your-dir/load/uwutestf -e iffint_pgmcon
       toolkit-dir/global/obj/vicrt.o
       your-dir/obj/mcbuwute.o your-dir/obj/cfunc.o
       toolkit-dir/global/obj/prcintcv.o
       toolkit-dir/global/obj/iffint.o
       toolkit-dir/global/obj/toolkit.a
       +n -R80000 -HF -N -z -a archive -lm -lc -lf
```

## Command List for Engines in C

This section contains the set of commands generated by a `make` of the sample engine, cengxmpl.c. This command set appears very similar to the procedure command set, except that no grammar function is generated, compiled, or linked in. Note also that the engintcv.o object is linked, which is required of C engines.

```
sas toolkit-dir/global/test/pgmcon.sas
       -sysparm ENGINE*engxmpl*mcbengxm*@*@*
                @toolkit-dir/global/grm/*@your-dir/src/*
                @toolkit-dir/global/grm*@your-dir/obj/*c
/bin/cc your-dir/src/cengxmpl.c -g -c -Iyour-dir/maclib/ -DHPUX
mv cengxmpl.o your-dir/obj/
/usr/local/bin/ld  -u calloc -u malloc -u realloc -u free
       -o your-dir/load/engxmpl -e engint_pgmcon
       toolkit-dir/global/obj/vicrt.o
       your-dir/obj/mcbengxm.o your-dir/obj/cengxmpl.o
       toolkit-dir/global/obj/prcintcv.o
       toolkit-dir/global/obj/engintcv.o
       toolkit-dir/global/obj/toolkit.a
       +n -R80000 -HF -N -z -a archive -lm -lc -lf
```

# Compiling and Linking FORTRAN Programs without make Files

This section lists all of the commands generated by `make` for the FORTRAN version of the MULTIPLY procedure under HP-UX. If you are using `make`, you do not need to know the detailed information listed in this part of the appendix. However, if you are not using `make`, you can use the contents of these command lists to model your own process for compiling and linking.

## Command List for Procedures in FORTRAN

```
❶ sas toolkit-dir/global/test/pgmcon.sas
       -sysparm PROC*multiply*mcbmulti*@fmultg*@multg*
                @toolkit-dir/global/grm/*@your-dir/src/*
                @toolkit-dir/global/grm*@your-dir/obj/*fortran
❷ sas toolkit-dir/global/test/ftnprep.sas
       -sysparm  your-dir/src/fmultipl.for
❸ f77 -U -g -c -Iyour-dir/maclib/ fmultipl.f
❹ mv fmultipl.o your-dir/obj/
❺ sas toolkit-dir/global/test/grmfunc.sas
       -sysparm PROC*multiply*mcbmulti*@fmultg*@multg*
                @toolkit-dir/global/grm/*@your-dir/src/*
                @toolkit-dir/global/grm*@your-dir/obj/*fortran
❻ sas toolkit-dir/global/test/ftnprep.sas
       -sysparm  your-dir/src/fmultg.for
❼ f77 -U fmultg.f -g -c -Iyour-dir/maclib/
❽ mv fmultg.o your-dir/obj/
❾ /usr/local/bin/ld  -u calloc -u malloc -u realloc -u free
       -o your-dir/load/multiply -e multiply_pgmcon
```

```
⑩  /lib/crt0.o toolkit-dir/global/obj/main.o
⑪  your-dir/obj/mcbmulti.o your-dir/obj/fmultipl.o your-dir/obj/fmultg.o
⑫  toolkit-dir/global/obj/prcintca.o
⑬  toolkit-dir/global/obj/toolkit.a
⑭  +n -R80000 -HF -N -z -a archive -lm -lc -lf
```

1. This command is the SAS invocation that creates the program constants object. The global SAS program pgmcon.sas is used to generate the object. The -sysparm option contains a rather lengthy string of characters (broken up over several lines here) that instructs pgmcon.sas on generating the object. If you choose to avoid using `make` files, you need to write your own SAS program to invoke PROC USERPROC for generating a program constants object.
2. This command invokes the ftnprep.sas program. This SAS program is a FORTRAN preprocessor that converts your input xxx.for file into an output xxx.f file. This preprocessing is necessary in order to ensure that the FORTRAN file meets the proper criteria for successful compilation and linkage. Preprocessing is necessary for both the procedure source and the grammar function source.
3. This command compiles the sample FORTRAN source code. The -U option indicates that there is to be no automatic lowercasing of source. This option is required so that the external names used by the source will not be lowercased, causing unresolved references. Ensure that all the keywords in your FORTRAN program (integer, if, goto, etc.) are lowercased before preprocessing. Note that the options -g (debug), -c (no link), and -I (include file directory) are specified. The -c option **must** be specified, since the link must be performed after all other components are built. The -I option can be omitted if all #include files are in the same directory.
4. This command moves the object from the current directory to the object directory. If you are performing all processing in the same directory, this command can be omitted.
5. This command is the SAS command to build a grammar function. As with the program constants object, you can provide your own PROC USERPROC invocation in your own file if you do not wish to use `make`.
6. See step 2 for an explanation of this FORTRAN preprocessing step.
7. This command compiles the grammar function generated by command 4.
8. This command copies the object as in command 3.
9. This command is the link. The four -u options must be present. The -o is required and specifies the output object The -e option supplies the entry name, which will be xxxxxxxx_pgmcon, where xxxxxxxx is your module name. 3) The linker is given
10. This command includes /lib/crt0.o and main.o. These files are required for successful linkage of FORTRAN code. /lib/crt0.o should be available on your UNIX system, and main.o is supplied as part of SAS/TOOLKIT Software.
11. This command contains the program constants object, the procedure object, and the grammar function object.
12. This command contains the required procedure interface object prcintca.o. The prcintca (PRoCedure INTerface Call-by-Address) object is used by FORTRAN.
13. This command is the object library for additional resolutions (also supplied with SAS/TOOLKIT software).
14. This command contains additional options that are all required. They should not be

altered. The ordering of the options and operands is important, and should be as indicated.

Here is the set of commands generated by a `make` of the sample functions:

## Command List for IFFCs in FORTRAN

This section contains the set of commands generated by a `make` of the sample function, ffunc.for. This command set is very similar to the procedure command set, except that no grammar function is generated, compiled, or linked in. Note also that the iffint.o object is linked, which is required of IFFCs written in FORTRAN.

```
sas toolkit-dir/global/test/pgmcon.sas \
       -sysparm FUNCTION*uwutestf*mcbuwute*@*@*
                @toolkit-dir/global/grm/*@your-dir/src/*
                @toolkit-dir/global/grm*@your-dir/obj/*fortran
sas toolkit-dir/global/test/ftnprep.sas
        -sysparm  your-dir/src/ffunc.for
f77 -U -g -c -Iyour-dir/maclib/ ffunc.f
mv ffunc.o your-dir/obj/
/usr/local/bin/ld  -u calloc -u malloc -u realloc -u free
        -o your-dir/load/uwutestf -e iffint_pgmcon
        /lib/crt0.o toolkit-dir/global/obj/main.o
        your-dir/obj/mcbuwute.o your-dir/obj/ffunc.o
        toolkit-dir/global/obj/prcintca.o
        toolkit-dir/global/obj/iffint.o
        toolkit-dir/global/obj/toolkit.a
        +n -R80000 -HF -N -z -a archive -lm -lc -lf
```

## Note on Engines in FORTRAN

At the current time, SAS/TOOLKIT software supports only the C language under UNIX for writing engines.

blank