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

Copyright © 1992 by SAS Institute Inc., Cary, NC, USA.

ISBN 1-55544-523-3

All rights reserved. Printed in the United States of America. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, or otherwise, without the prior written permission of the publisher, SAS Institute Inc.

1st printing, October 1992


The Institute is a private company devoted to the support and further development of its software and related services.

Other brand and product names are registered trademarks or trademarks of their respective companies.

Doc P19, 092892
Contents

Reference Aids v
Using This Book vii
Summary of Changes and Enhancements xi

Chapter 1 - Using the new TLKTDBG facility 1
Introduction 1
Example 2
Explanation of TLKTDBG Output 4

Chapter 2 - Using C Language Debuggers with SAS/TOOLKIT Programs 7
Introduction 7
Using the SAS/C Debugger Under MVS and CMS 7
Using the C Language Debugger under VMS 10
Using the dbx Debugger under AIX 11

Chapter 3 - Debugging Grammars 17
Introduction 17
Sample Session 17

Chapter 4 - Writing Engines 21
Introduction 22
Using Engines with SAS Programs 22
Writing the Engine 24
Example 29
Data Structures Used with Engines 47
ENG Routine Specifications 55

Chapter 5 - Writing Special DATA Step Functions 71
Introduction 71
The SAS_DSS Routines 71
Sample Program 74

Chapter 6 - Writing SAS/IML functions 83
Introduction 83
Example 83
SAS_IMxxxx Routine Reference 87
Contents

Using Subroutine Libraries with SAS/IML Functions  93

Chapter 7 - Other New Routines  95
Introduction  95
SAS_XDNAMCL Routine  95
SAS_ZCATMEM Routine  95
SAS_ZMISSVF Routine  96

Chapter 8 - PARMCARDS Processing  97
Introduction  97
Required SAS Statements  97
PARMCARDS Processing in Your Procedure  98
Example  98

Appendix 1 - Using SAS/TOOLKIT Software Under OS/2  103
Introduction  103
Accessing User-Written SAS Modules  104
Creating Executable Modules  106
Compiling and Linking without Using Make Files  113
Directory Structure  117
Note on Using the IBM Set/2 Debugger  124

Appendix 2 - Using SAS/TOOLKIT Software Under UNIX Systems  127
Introduction  127
Accessing User-written SAS Modules  127
Creating Executable Modules  128
Compiling and Linking C Programs without make Files  136
Compiling and Linking FORTRAN Programs without make Files  139
Reference Aids

Tables

4.1 Required Step for Writing Engines 26

Examples

4.1 CEXAMPL Example of Data Base Engine 29

Tables

A1.1 Contents of TEST Subdirectory 117
A1.2 Contents of GRM Subdirectory 118
A1.3 Contents of OBJ Subdirectory 119
A1.4 Contents of SRC Subdirectory 119
A1.5 Contents of OBJ Subdirectory 120
A1.6 Contents of LOAD Subdirectory 122
A1.7 Contents of CNTL Subdirectory 122
A1.8 Contents of MACLIB Subdirectory 123

Figures

A1.1 Summary of Directory Structures 124
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.


"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 Setl2 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.

<table>
<thead>
<tr>
<th>If you are ...</th>
<th>You should read ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>working with the OS/2 operating system</td>
<td>Appendix 1 to learn how to compile and link modules using the make facility under OS/2 2.0.</td>
</tr>
<tr>
<td>working with the UNIX operating system</td>
<td>Appendix 2 to learn how to compile and link modules using the make facility under UNIX.</td>
</tr>
<tr>
<td>writing a procedure</td>
<td>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.</td>
</tr>
<tr>
<td>writing an engine</td>
<td>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</td>
</tr>
<tr>
<td>writing a special DATA step function</td>
<td>Chapter 5 for instructions on doing this.</td>
</tr>
<tr>
<td>writing a SAS/IML function</td>
<td>Chapter 6 for instructions on doing this.</td>
</tr>
</tbody>
</table>
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.

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 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 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, TLKDBG, 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 TLKIDBG facility

Introduction

The TLKIDBG 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 TLKIDBG=x. For example,

\texttt{OPTIONS DEBUG='TLKIDBG=1';}

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

When a nonzero TLKIDBG 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 TLKIDBG 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 TLKIDBG value does not require any recompilation or relinking of your code.

The TLKIDBG 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 TLKIDBG 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 TLKIDBG 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 TLKIDBG 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 shows 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 shows 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.
Using the new TLKDBG facility

NOTE: The data set WORK.NEW has 1 observations and 7 variables.
NOTE: The PROCEDURE SIMPLE used 0.07 CPU seconds and 32.54K.

The next example shows the results of running the same procedure with DEBUG='TLKDBG=3'.

10  * WITH TLKDBG 3 (ALSO PROVIDE DEREFERENCEC3 :  
11 OPTIONS DEBUG='TLKDBG=3';  
12 PROC SIMPLE DATA=TEMPOUT=NEW;RUN;  
NOTE: This is PROC SIMPLEC for the C Compiler.

TLKTDBG1: Calling SAS_XMEMEX (SAS027)...
TLKTDBG2: arg1= 176 (00000000)
TLKTDBG1: --Returning from SAS_XMEMEX (SAS027)...
TLKTDBG2: return value is: 05363CA8
TLKTDBG1: Calling SAS_ZMOVEI (SAS122)...
TLKTDBG2: arg1= 05363F00, arg2= 05363CA8, arg3= 176 8----------:
TLKTDBG1: --Returning from SAS_ZMOVEI (SAS122)...
TLKTDBG2: return value is: 05363CA8, arg2= 00000000, arg3= 176 8----------:
TLKTDBG1: Calling SAS_XSPARSE (SAS001)...
TLKTDBG2: arg1= 05363CA8, arg2= 00000000, arg3= 176 8----------:
TLKTDBG1: --Returning from SAS_XSPARSE (SAS001)...
TLKTDBG2: return value is: 00000000
TLKTDBG1: Calling SAS_XVGETI (SAS083)...
TLKTDBG2: arg1= 05364800, arg2= 1 (00000001), arg3= 8----------:
TLKTDBG1: --Returning from SAS_XVGETI (SAS083)...
TLKTDBG2: return value is: 00000000
TLKTDBG1: Calling SAS_XVPUTI2 (SAS095)...
TLKTDBG2: arg1= 05364800, arg2= 1 (00000001), arg3= 8----------:
TLKTDBG1: --Returning from SAS_XVPUTI2 (SAS095)...
TLKTDBG2: return value is: 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

TLKTDBG1: deref'd return ptr->00000000
TLKTDBG2: deref'd outgoing values arg1->00000000

NOTE: This is PROC SIMPLEC for the C Compiler.
Explanation of TLKTDBG Output

Here is a sample of output from the TLKTDBG facility.

TLKTDBG1: Calling SAS_XVGETI (SAS083)...
TLKTDBG2:  arg1= 053C4800, arg2= 1 (00000001), arg3= 000D5000
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= 000D5000
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.

The following paragraphs explain this output line by line.

TLKTDBG1: Calling SAS_XVNAME (SAS089)...
TLKTDBG2:  arg1= 053C4800, arg2= 1 (00000001), arg3= 000D5000
TLKTDBG1: --Returning from SAS_XVNAME (SAS089)...
TLKTDBG2:  return value is: 0 (00000000)
TLKTDBG3: deref'd outgoing values... arg1->05383800, arg3->0542E200

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= 000D5000

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. Errors 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 pagen='=-d';
3?
proc multiply data=temp out=new;run;
Set system breakpoint at OFABAC to activate ESCAPE command
SAS/C Source Debugger Release 4.50H
TRACE: (LSMAIN) -> MAIN(MAIN)
Listing source from dsn: your.sas.proc.source.library(CMULTIPL)
MAIN(MAIN) 32: void U_MAIN(MULTIPLY){}
bear on entry - context of MAIN
CDEBUG:
  b print_obs e
CDEBUG:
  b print_sum e
CDEBUG:
  g
SAS/TOOLKIT SOFTWARE ** MULTIPLY PROCEDURE **.
This procedure is written in C

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

The SAS System

<table>
<thead>
<tr>
<th>OBS</th>
<th>VARIABLE</th>
<th>PREVIOUS VALUE</th>
<th>NEW VALUE</th>
<th>LENGTH</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

NOTE: The data set WORK.NEW has 1 observations and 2 variables.
NOTE: The PROCEDURE MULTIPLY used 0.28 CPU seconds and 3267K.
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
//BEGIN 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 OBEBAC to activate ESCAPE command
SAS/C Source Debugger Release 4.50N
TRACE: (LSCHAIN) -> 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  00230000  002420FF

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
```
Using C Language Debuggers with SAS/TOOLKIT Programs  □ Using the dbx Debugger under AIX

1987: double obser,

DBG> g

break at routine CMULTIPL\Prnt_sum
2002: double sum;

DBG> g

The SAS System
       1
       (current date)

<table>
<thead>
<tr>
<th>OBS</th>
<th>VARIABLE</th>
<th>PREVIOUS VALUE</th>
<th>NEW VALUE</th>
<th>SUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X</td>
<td>1</td>
<td>1</td>
<td>$1.00</td>
</tr>
</tbody>
</table>

NOTE: The dataset WORK.DT1 has 1 observations and 2 variables.

? 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 Oxd000e638
0xd000e638 (load-0x40) 80410014 1 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 I? 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 Oxd000e5f8
0xd000e5f8 (load) 7c0802a6 mf1r 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 $stkpt: 0x2010b648 $toc: 0x2001bdf4 $r3: 0x20084f46
$r4: 0x00000000 $r5: 0x00000000 $r6: 0x20052c20 $r7: 0x00b88621
$r8: 0x2002ac00 $r9: 0x00000000 $r10: 0x20000000 $r11: 0x20000000
$r12: 0x2001ab18 $r13: 0xdeadbeef $r14: 0xdeadbeef $r15: 0xdeadbeef
```
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 Ox2008f46e would be

```
Ox2008f46e/50c
```

The response from the command is something like this:

```
2008f46e: ' ' 's' 'a' 's' 'l' 'h' 'b' 'c' 'm' 'e' '/ ' 's' 'a' 's' 'x' 'y' 'z'
2008f47e: ' ' 't' '0' '0' 'i' 'k' 't' 't' 'e' 't' '0' 'a' 'd' '/ '
2008f48e: '0' 'p' 'e' 'n' 't' 's' 't' 's' 't' '\0' '\0' '\0' '\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 /saslhomesasxyz/toolkitc/loadlopentst, 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 1 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:
Using the dbx Debugger under AIX

Chapter 2

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 UWRCC(&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 _MAIN macro in the procedure source. For SAS/TOOLKIT informats/functions/formats/call routines (IFFCs), use IFFMAI. For SAS/TOOLKIT engines, use ENGMAL. 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 :.
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 17

Sample Session 17

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.

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

NOTE: The dataset 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 DSINIT 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 DSFLT 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)
```
The following lists explains sections of the output.

**.SEM.** identifies it as a sentinel for semantic action debugging.

**actn** 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 that 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=6:6 ts= 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=6:1 xzploc=7:1ts= 8 TEMP
   .tok. type= 38 subtype=-1776 ident=1 ss=0 loc=5:24 xzploc=6:24 ts= SELL
   .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= 11 subtype= 0 ident=0 ss=1 loc=5:35 xzploc=6:35 ts= ;
   .tok. type= 2 subtype=-111 ident=0 ss=0 loc=5:37 xzploc=6:37 ts= NOT
   .tok. type= 11 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(I) 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.