SAS/C® Debugger User's Guide and Reference, Release 7.00

The Power to Know™

sas® | SAS Publishing

The correct bibliographic citation for this manual is as follows: SAS Institute Inc., SAS/C® Debugger User's Guide and Reference, Release 7.00, Cary, NC: SAS Institute Inc., 2001.

SAS/C® Debugger User's Guide and Reference, Release 7.00

1–58025–704–6

SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513.

1st Printing, April 2001

Contents

PART 1

Introduction to the SAS/C Debugger

CHAPTER

1

An Overview of the SAS/C Debugger

Introduction

This chapter introduces the methods of running the SAS/C Debugger, summarizes its capabilities, and provides a quick start to using the debugger under the CMS and OS/390 operating environments.

Methods of Running the Debugger

The SAS/C Debugger helps you find run-time errors, or bugs, in programs that have been compiled by the SAS/C Compiler. This process is commonly referred to as debugging. The debugger has been designed to help you determine the location of errors in your programs and the reasons why the errors are occurring.

There are two interactive methods of running the SAS/C Debugger under both CMS and OS/390: full-screen mode and line mode. If you are running under OS/390, you can also run the debugger noninteractively in batch mode.

Full-Screen Mode

In full-screen mode, the SAS/C Debugger must be used with an IBM 3270 type of terminal display system or an emulator. Today, most people work on a Windows operating environment that uses 3270 emulator software. If your terminal supports full-screen applications, the SAS/C Debugger runs in full-screen mode by default. In full-screen mode, the debugger provides an interactive windowing interface that enables you to view information and operate through a series of windows. Each window has a distinct function and displays logically related data. You will probably find that full-screen mode offers the most efficient interface.

Line Mode

You can override the default and run the debugger interactively in line mode. In line mode, you issue debugger commands at the **`Cdebug:`** prompt, and the screen displays information in response. The SAS/C Debugger supports all commands from previous releases as well as several new features for Release 7.00. You can switch to line mode after invoking the debugger, as described in “Switching to Line Mode” on page 9.

Batch Mode

The OS/390 data set can be routed to any system output device that is available at your site, including a direct access storage device or your terminal. The details of batch mode are presented in Chapter 5, “Running the Debugger under TSO,” on page 61 and in Chapter 6, “Running the Debugger under OS/390 Batch,” on page 65.

Capabilities of the SAS/C Debugger

Regardless of the method you use to run the debugger, you will find that it has many capabilities that will enhance your productivity. With the SAS/C Debugger, you can

- start execution of your SAS/C program.
- trace program flow.
- request *breakpoints* that are used to interrupt an executing program. For details see Chapter 12, “Using Debugger Commands,” on page 129.
- reexecute or bypass code, resuming execution after a specified location.
- single step through your program. You can either step into called functions or step over function calls.
- perform a variety of actions to access and modify expressions, including structures, unions, arrays, and classes. For details see Chapter 12, “Using Debugger Commands,” on page 129.
- change the *command scope* in order to perform actions on any expression in any function in the calling sequence.
- watch expressions in order to check for arbitrary changes in value.
- write customized debugger commands in the form of a CLIST or REXX EXEC.
- produce a storage analysis report.
- display the type information of an expression.
- obtain a traceback of active functions.
- generate a SIGINT signal in order to test program signal handling.
- resume execution after you correct the cause of a program check.
- disable debugger commands, reenable them, and drop them.
- perform an operating environment command.
- escape to your operating environment’s debugger.
- specify characteristics of debugger output.

- list program source.
- abort execution.
- execute an EXEC or a CLIST language command.
- execute a PROFILE upon entry to the debugger.
- add your own custom debugger input and output routines for specialized debugging needs.
- write debugger macros.
- install user-defined commands.

Understanding Full-Screen Mode

The SAS/C Debugger must be used with an IBM 3270 type of terminal display system or emulator. An IBM mainframe environment handles screen display differently from a UNIX or Windows environment. If you are new to IBM mainframes, be aware of some of the differences, because they affect the way you interact with the SAS/C Debugger. A basic understanding of 3270 concepts helps you to understand why the debugger reacts in a manner that may seem unfamiliar to you. These differences are explained in the following section.

Basic 3270 Concepts

Typically, under the UNIX or Windows environments, input characters are available to an application program immediately as they are typed. In fact, the terminal may be equipped with keys that allow characters to be deleted from or inserted into a string of characters that is input to an application program. The application program processes all characters as they are input until a complete field is assembled.

Instead of single-character, unbuffered input and output as is used under the Windows or UNIX environments, the IBM 3270 terminal environment maps each character on the display to a storage area in the device. This storage area is known as the character buffer. The applications program, in this case the SAS/C Debugger, uses commands to send or receive data from this buffer. The data that is transferred between a terminal and an application is referred to as the data stream. An IBM 3270 data stream includes commands and control characters that specify the processing of the data in the character buffer. Most often, the buffered data are grouped into fields and formatted by using display attribute codes.

An application program can then display information by constructing a data stream and sending it to the device. Input is obtained by reading the buffered data and decoding the data stream that is returned. As a result, in IBM data stream programming, output to the terminal is the result of a command; input to an application program occurs only when the ENTER key or a program function key is pressed.

Once a data stream is constructed, an application program in an IBM mainframe environment must ask the operating environment to transmit it to the terminal.

Note: The ability to display attributes such as color or reverse video may be limited by the capabilities of your terminal. △

Table 1.1 on page 6 summarizes some important differences between the IBM 3270 full-screen communication and other environments. Be aware of these differences when you use the SAS/C Debugger. For additional information about the IBM 3270 data stream and full-screen programming, see the SAS/C Full-Screen Support Library User's Guide.

Table 1.1 Differences in Handling Characters

IBM 3270	Other Environments
buffered input	single character, unbuffered input
character available after ENTER or function key	character available every keystroke
character cannot be deleted after ENTER or function key	character can be deleted from assembled character string

Quick Start to Using the Debugger

This section explains the steps necessary to compile, link, and run programs with the SAS/C Debugger under CMS and OS/390. Part 3 of this book, "Using the Debugger in Your Environment," also describes these steps, but it includes more detailed information about using compiler options, allocating data sets that are needed by the debugger, saving a debugger session, setting up a debugger profile, and so on. The quick start procedures that are contained in this section are helpful to a new SAS/C Debugger user. If you are an experienced CMS or OS/390 user, familiar with previous releases of the debugger, see "Notes for Users with SAS/C Debugger Experience" on page 9.

Quick Start to Using the Debugger under CMS

This CMS quick-start procedure provides the essential information that you need in order to start using the debugger under CMS. Detailed information about compiling and linking your programs is provided in the SAS/C Compiler and Library User's Guide, Third Edition. Also, once you are comfortable with running the debugger, you can find more advanced information in Chapter 7, "Running the Debugger under CMS," on page 69.

CMS Quick-Start Procedure

These are the steps to using the debugger under CMS:

1 LC370 *program-name* (DEBUG:

This step compiles the source file for *program-name* with the compiler **debug** option. The name of the program is *program-name* C. The compiler output is placed in a file called*program-name* TEXT. The debugger symbol table file is placed in*program-name* DB. (The compiler outputs the debugger symbol table file, which the debugger uses at run time.)

2 COOL *program-name* (GENMOD *program-name*

This step creates a load module of your program. The output is placed in *program-name* MODULE. COOL is not always required. If you do not need to use COOL, you can issue the following commands:

LOAD *program-name*

GENMOD *program-name*

See the *SAS/C Compiler and Library User's Guide* for information about the circumstances that require the use of COOL.

3 *program-name* =DEBUG

This step runs your program. The **=debug** option invokes the debugger.

Note: Even though you compile with the **debug** option, you must invoke the debugger at run time also. △

Quick Start to Using the Debugger under TSO

If you are running under OS/390, then probably you want to use the debugger interactively under TSO. This section contains the basic information required to run the debugger under TSO.

Note: The SAS/C Debugger can be run in a batch job under OS/390. This procedure is explained in Chapter 5, "Running the Debugger under TSO," on page 61, and in Chapter 6, "Running the Debugger under OS/390 Batch," on page 65. △

Preparing Your Source and Debugger Data Sets

Before you can compile, link, and debug a program, you must create two data sets, one to hold your source code and one to hold the debugger symbol table file that is created when you compile with the **debug** option. The attributes of the data set that holds your source code are not critical; however, the debugger file does require specific attributes for record format and blocksize. For the debugger symbol table file, allocate a partitioned data set (PDS) with RECFM=U and BLKSIZE=4080. For the purpose of this quick-start procedure, use the data set names *userid.pdsname.*C and *userid.pdsname.*DBGLIB, as recommended in "TSO Quick-Start Procedure" on page 7.

Note: The object and load data sets, *userid.pdsname.*OBJ and *userid.pdsname.*LOAD, are automatically created when your program is compiled and are linked if they do not exist already. △

TSO Quick-Start Procedure

This procedure provides the essential information that you need in order to start using the debugger. Detailed information about compiling and linking your programs is provided in the SAS/C Compiler and Library User's Guide. Also, once you are accustomed to running the debugger, you can find more advanced information in Chapter 5, "Running the Debugger under TSO," on page 61.

When your source and debugger data sets are ready, you can compile, link, and run under TSO by using the following steps:

1 LC370 *pdsname* (*program-name*) DEBUG

For example, the following command compiles a member named HELLO that is contained in the PDS *userid*.MYWORK.C:

```
LC370 MYWORK(HELLO) DEB
```

The object code output from the compiler is placed in the PDS *userid*.MYWORK.OBJ(HELLO). Another output (which contains special tables that are used while running the debugger) is placed in the PDS *userid*.MYWORK.DBGLIB. The member name is the section name (SName).* The

* During compilation, the compiler creates names for various data objects in the compilation. In general, these names are based on the section name. The section name, in turn, can be specified by the **sname** compiler option. If no section name is specified, the compiler assigns one as described in the chapter about compiler options in the *SAS/C Compiler and Library User's Guide*.

debugger uses the section name in order to locate the member that contains debugging information for the compilation. For more information, see "SECTION-NAME and FUNCTION-NAME Arguments" on page 134.

2 COOL *pdsname* (*program-name*)

The following command directs the linker to use the object code in *userid.*MYWORK.OBJ in order to create a load module of your program:

```
COOL MYWORK(HELLO)
```

The output from the linker is placed in the partitioned data set *userid.* MYWORK.LOAD(HELLO). See the *SAS/C Compiler and Library User's Guide* for more information.

3 ALLOC DA(*pdsname*.DBGLIB) F(DBGLIB) SHR

The following ALLOCATE statement allocates *userid.*MYWORK.DBGLIB:

```
ALLOC DA(MYWORK.DBGLIB) F(DBGLIB) SHR
```

4 CALL *pdsname* (*program-name*)'=DEBUG'

The following command runs the program in *userid.*MYWORK.LOAD(HELLO):

```
CALL MYWORK(HELLO) '=DEBUG'
```

The **`'=debug'`** option invokes the debugger. The debugger prompts you for the debugger data set name if you omit step 3.

Naming your data sets Name your partitioned data sets (PDSs) as follows:

source data set	*userid.pdsname*.C
object data set	*userid.pdsname*.OBJ
load data set	*userid.pdsname*.LOAD
debugger data set	*userid.pdsname*.DBGLIB

where *userid* is your TSO user prefix. *pdsname* can be any name, but it must be the same for all four data sets, as in the following:

*userid.*MYWORK.C

*userid.*MYWORK.OBJ

*userid.*MYWORK.LOAD

*userid.*MYWORK.DBGLIB

The LC370 and COOL CLISTs assume these naming conventions. If you follow the naming conventions, you do not have to override data set names.

If you do not format the names of the data sets as shown above, then you must fully qualify all of your data set names. For example, suppose you name a partitioned data set *userid*.MYWORK.SOURCE, using its members to contain your source code. Then, to compile a member that is named PROGRAM, the command is

```
   LC370 'userid.MYWORK.SOURCE(PROGRAM)' DEBUG
OBJECT(, , , userid.MYWORK.OBJ (PROGRAM)''')
DBGLIB(, , , userid.MYWORK.DBGLIB (PROGRAM)''')
```

As you can see, this compile step is much more complicated than the compile step (step 1) given earlier in this section.

The same thing happens with the COOL CLIST. For example, if your object module data set is *userid*.MYWORK.OBJECT, then the command is

```
   COOL 'userid.MYWORK.OBJECT(PROGRAM)'
LOAD(, , , userid.MYWORK.LOAD (PROGRAM)''')
```

Notes for Users with SAS/C Debugger Experience

The full-screen windowing interface provides a useful addition to your productivity tools. These notes enable you to run the new release while using your existing knowledge of the debugger.

Invoking the Debugger

As with previous releases of the software, you invoke the SAS/C Debugger by adding the **`=debug`** option to the command that runs your program.

Switching to Line Mode

As previously mentioned, full-screen mode is usually the default method of running the debugger. After invoking the debugger, you can use the **`window off`** command to switch from full-screen mode to line mode. Issue this command from the command window, which is located at the bottom of the screen and is indicated by the **`Cdebug:`** prompt.

Once you enter line mode, you can issue all of the debugger commands that you used with previous releases of the SAS/C Debugger.

Using the Windowing Interface to Enter Line-Mode Commands

Instead of switching to line mode, you may prefer to issue your commands from the command window while in full-screen mode. This gives you the advantage of viewing your source code in the Source window and your previously entered commands in the Log window. Commands that you issue from the **`Cdebug:`** prompt in the Command window use the same syntax as line-mode commands do. Another major advantage to this approach is that you begin to familiarize yourself with the windowing interface.

CHAPTER

2

The Windowing Interface

Introduction

This chapter provides an introduction to the windowing interface that you use to control the SAS/C Debugger during full-screen mode. In full-screen mode you can use the Command window or PF keys to issue commands that control your debugging session. As a result of these commands, the SAS/C Debugger displays information about your program and the session. The windowing interface is composed of windows in which you can send commands and other information to the debugger.

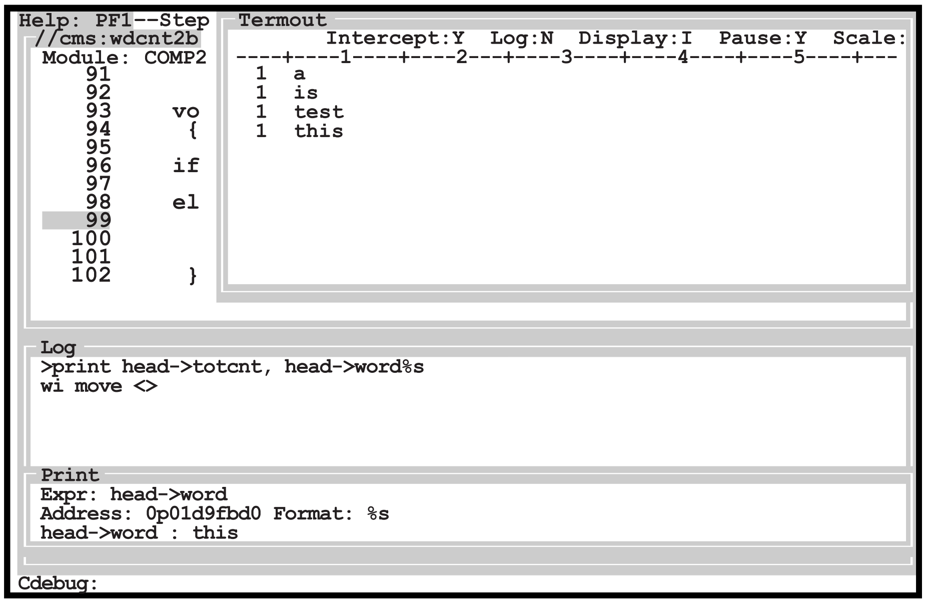
Some Window Basics

The windowing interface consists of a series of windows through which you issue commands to perform a variety of tasks. Each window consists of an optional *border* and one or more *fields*. The border defines the location and display area for the window on the terminal screen. The fields are areas inside the window that are used to view and type information. The field can be extended beyond the window borders. That is, the field can contain information that is not currently visible within the borders. You scroll the window horizontally or vertically to view this information.

Multiple Windows Can Be Displayed

You can have several windows displayed simultaneously, which enables you to issue a command in one window and view its results in a second window. For example, Display 2.1 on page 13 shows the Status, Source, Termout (terminal output), Log, Print, and Command windows. The functions of these and other windows that are not shown in this example are explained in “Types of Windows” on page 14.

Display 2.1 Displaying Several Windows Simultaneously



```
Help: PF1--Step  Termout
 //cms:wdcnt2b          Intercept:Y  Log:N  Display:I  Pause:Y  Scale:
 Module: COMP2   ----+----1----+----2---+----3----+----4----+----5----+---
     91            1  a
     92            1  is
     93     vo     1  test
     94      {     1  this
     95
     96     if
     97
     98     el
     99
    100
    101
    102      }

 Log
 >print head->totcnt, head->word%s
 wi move <>

 Print
 Expr: head->word
 Address: 0p01d9fbd0 Format: %s
 head->word : this

Cdebug:
```

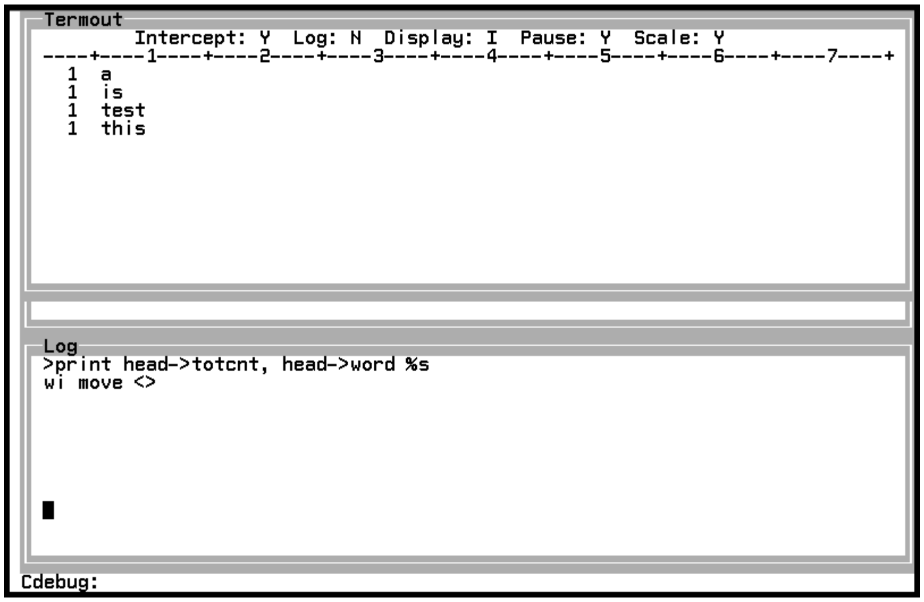
Some windows display all of the information that is associated with their function at one time. For example, the Status window consists of one line of status information, as shown in Display 2.1 on page 13; therefore, all of the information that is associated with the Status window is displayed simultaneously.

Other windows, such as the Source and Termout windows in this example, might display only a portion of the information that is associated with their function at one time. You may have to scroll through a field to view information that is hidden.

Windows Can Be Moved and Resized

A window's size affects how much information can be displayed at one time. Most windows can be resized, moved, opened, closed, and zoomed. (When you zoom a window, it either fills the entire display or shows the maximum information available for that window.) For example, Display 2.2 on page 14 shows the initial position of the Termout window. In the example shown in Display 2.1 on page 13, the Print window has been moved, and the Termout window has been moved and resized. Both of these windows filled the top half of the display when they were originally displayed. See "Controlling the Windowing Interface" on page 24 for complete descriptions of how to open, close, move, resize, and zoom windows.

Display 2.2 Initial Position of the Termout Window



```
Termout
         Intercept: Y  Log: N  Display: I  Pause: Y  Scale: Y
----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+
   1  a
   1  is
   1  test
   1  this

Log
>print head->totcnt, head->word %s
wi move <>

Cdebug:
```

Windows Can Overlap

When several windows are displayed, you can choose to have them overlap each other. This allows you to display the maximum size for the window in which you are working, while keeping the other windows readily available. Depending on the size and position of the overlapped windows, parts of them may be visible, or they may be completely hidden. In Display 2.2 on page 14, the Termout window overlaps the Source window.

The debugger uses a stack to keep track of open windows. You can move through this stack of windows in either direction, or you can jump directly to a specific window in the stack. The current window is called the top window because it is on top of the stack. It overlays any other windows that happen to be located on the same portion of the display.

Windows Have Logical Cursors

Each window has a logical cursor that is associated with it. Although you cannot see the logical cursor, the debugger uses it to keep track of your location in a window. The current window displays a physical cursor that can be seen at the location of the logical cursor. Furthermore, the position of the logical cursor is used for certain operations (such as scrolling by cursor amount), which may cause the logical cursor to move.

Types of Windows

The windowing interface is composed of 15 different types of windows. (You can open multiple instances of some of these window types.) Reference information for each window is provided in Chapter 13, "Window Directory," on page 153. The following list provides a summary of their functions:

Browse window
  is used to browse text files and to display output from the **browse** command.

Command window
  is used to issue debugger commands.

Config window
: displays current configuration information such as default window locations, colors, and display attributes. This window can also be used to change window attributes and location.

Dump window
: displays a dump of memory in character and hexadecimal format.

Find window
: is used to enter a search string for the **`windows find`** command.

Help window
: provides context-sensitive help information. You can enter the help system from any location on the display by pressing the PF1 key. (By default, the PF1 key is used to access the help system. The Keys window is used to change the help key assignment.)

Keys window
: displays current PF key settings. This window can also be used to redefine a PF key.

Log window
: displays a log of the debugger commands that you enter during your session, as well as other useful information.

Message window
: displays a message in response to invalid input. You can close this window by pressing the ENTER key.

Pop-up window
: request for information in response to invalid input. The Pop-up window displays the invalid input, and you must either provide a valid value or delete the invalid value before you can continue.

Print window
: displays the value of an expression.

Register window
: displays the contents of general purpose and floating-point registers and the current instruction address in hexadecimal format.

Source window
: displays source code, highlighting the current position in the code. You can also enter prefix-area commands from the Source window to control your debugging session. Prefix area commands are described in “Source Window” on page 173.

Status window
: displays status information for your debugging session.

Termin window
: is used to type terminal input that your program requests.

Termout window
: displays terminal output from your program.

Watch window
: is used to monitor the changing value of an expression. As with the Print window, the expression can be of either the scalar or the aggregate type.

The four primary windows (Status, Source, Log, and Command) are always open. With the exception of the Message or Pop-up windows, you can use the **`window open`** command to take a quick look at the other windows that are listed in the previous section. To open a window, type the following command, where *name* is the name of the window:

```
    window open name
```

To close the window, move the cursor into the window and press PF15, which executes the **window close** command.

Using the Help Window

The Help window is used to access the debugger's hypertext help system. You can use the following formats of the **help** command to open the Help window:

help
displays the help system index when you enter it in the Command window.

help <>
provides context-sensitive help based on the cursor position. This command is normally assigned to the PF1 key; however, you can also enter it from the Command window. (The help key assignment is shown in the Status window.)

help*WINDOW-NAME | COMMAND-NAME | TOPIC*
allows direct access to information about a particular window, command, or topic. For example, **help print** opens the Help window and displays help information about the **print** command.

Note: The pair of angle brackets <> is used as a placeholder to specify the name of the window in which the cursor is currently located. When you press the PF key, the name of your current window is substituted for <> and the command is executed. For example, pressing the PF1 key when the cursor is in the Log window issues the following command:

```
window help <>
```

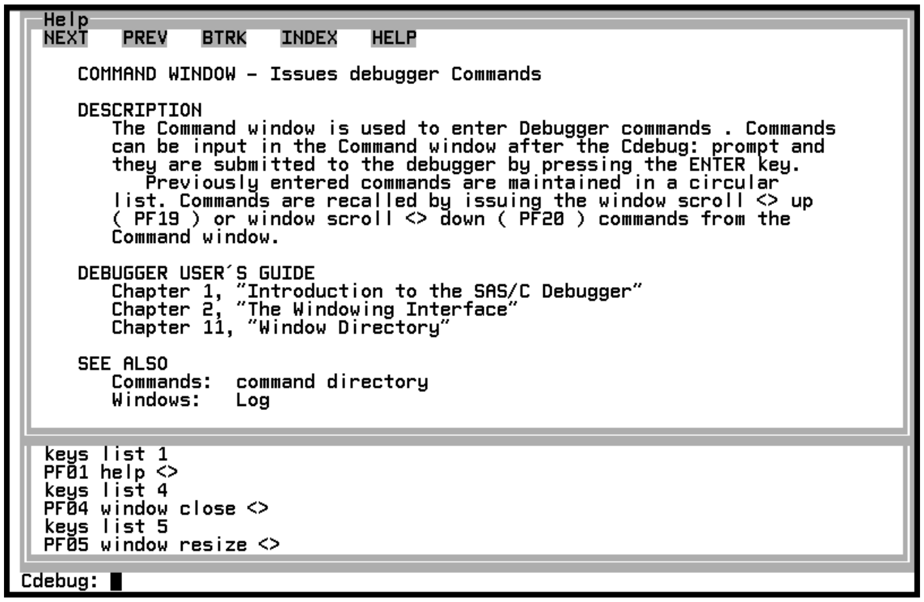
Since the cursor is in the Log window, **log** is substituted for the <> placeholder symbol when the command is executed. See "Using Placeholders in Window Subcommands" on page 150 for additional information about the <> placeholder. △

Help Key

The left side of the Status window identifies the PF key assigned to the **help** <> command. Normally, the PF1 key is assigned to this command. If no key is assigned, the Status window displays HELP:HELP, which indicates that you must issue the **help** command from the Command window in order to access the help system.

Pressing the help key opens the Help window, which then displays context-sensitive help information based on the current location of the cursor. For example, Display 2.3 on page 17 shows the Help window that is displayed by pressing the PF1 key while the cursor is located in the Command window.

Display 2.3 Help Window, Command Window Card



```
Help
NEXT   PREV   BTRK   INDEX   HELP

   COMMAND WINDOW - Issues debugger Commands

   DESCRIPTION
      The Command window is used to enter Debugger commands . Commands
      can be input in the Command window after the Cdebug: prompt and
      they are submitted to the debugger by pressing the ENTER key.
         Previously entered commands are maintained in a circular
      list. Commands are recalled by issuing the window scroll <> up
      ( PF19 ) or window scroll <> down ( PF20 ) commands from the
      Command window.

   DEBUGGER USER'S GUIDE
      Chapter 1, "Introduction to the SAS/C Debugger"
      Chapter 2, "The Windowing Interface"
      Chapter 11, "Window Directory"

   SEE ALSO
      Commands:  command directory
      Windows:   Log

keys list 1
PF01 help <>
keys list 4
PF04 window close <>
keys list 5
PF05 window resize <>
Cdebug:
```

Hypertext Cards and Links

The Help window displays information in hypertext format on hypertext cards. For example, Display 2.3 on page 17 shows the card for the Command window. These cards are displayed one at a time. You move between cards by a networks of links. A link has special display attributes that distinguish it from the surrounding text in a card. When you are running the debugger, links are displayed in either green or bold, depending on the characteristics of your terminal. You select a link by moving the cursor onto the link and pressing the ENTER key.

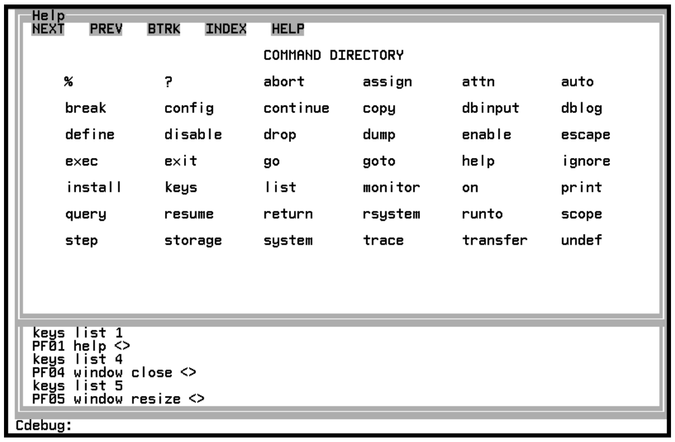
Navigation Links

Five navigation links always appear in the upper-left corner of the Help window. The navigation links are labeled **`NEXT`**, **`PREV`**, **`BTRK`**, **`INDEX`**, and **`HELP`**. As described in "Help Window" on page 164, these links are used to move rapidly through the help system following a predefined path or to move back through the cards you have selected.

Links within the Text of a Card

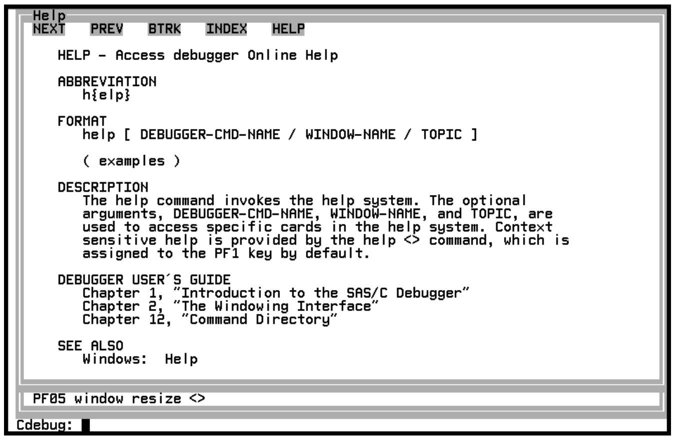
Most cards also display links that are embedded in the text of the card. For example, the Command window card shown in Display 2.3 on page 17 contains a link called **`debugger commands`** on the first line of the description. Selecting this link causes the COMMAND DIRECTORY card, shown in Display 2.4 on page 18, to be displayed. The COMMAND DIRECTORY card contains additional links to cards that describe each of the debugger commands. By selecting the **`help`** link, you can access the card shown in Display 2.5 on page 18.

Display 2.4 Help Window, COMMAND DIRECTORY Card



```
Help
NEXT   PREV   BTRK   INDEX   HELP
                        COMMAND DIRECTORY
    %          ?          abort      assign     attn       auto
    break      config     continue   copy       dbinput    dblog
    define     disable    drop       dump       enable     escape
    exec       exit       go         goto       help       ignore
    install    keys       list       monitor    on         print
    query      resume     return     rsystem    runto      scope
    step       storage    system     trace      transfer   undef

keys list 1
PF01 help <>
keys list 4
PF04 window close <>
keys list 5
PF05 window resize <>
Cdebug:
```

Display 2.5 Help Window, Help Command Card

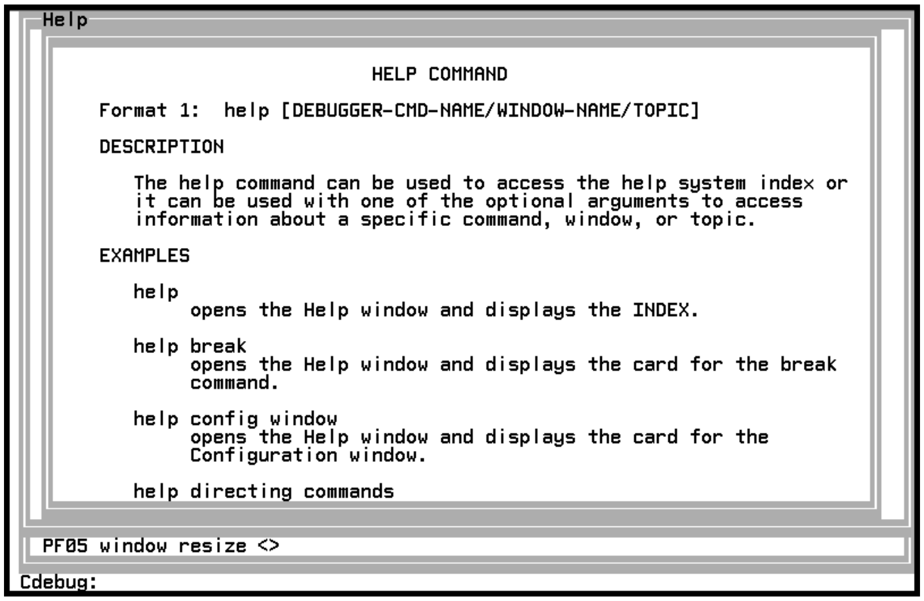


```
Help
NEXT   PREV   BTRK   INDEX   HELP
   HELP - Access debugger Online Help

   ABBREVIATION
      h{elp}

   FORMAT
      help [ DEBUGGER-CMD-NAME / WINDOW-NAME / TOPIC ]

      ( examples )

   DESCRIPTION
      The help command invokes the help system. The optional
      arguments, DEBUGGER-CMD-NAME, WINDOW-NAME, and TOPIC, are
      used to access specific cards in the help system. Context
      sensitive help is provided by the help <> command, which is
      assigned to the PF1 key by default.

   DEBUGGER USER´S GUIDE
      Chapter 1, "Introduction to the SAS/C Debugger"
      Chapter 2, "The Windowing Interface"
      Chapter 12, "Command Directory"

   SEE ALSO
      Windows:  Help

PF05 window resize <>
Cdebug:
```

Links to Pop-Up Windows

Some links are associated to pop-up windows. Selecting such a link causes a pop-up window to be displayed inside the Help window. For example, selecting the link named **`examples`** on the help card shown in Display 2.5 on page 18 causes the pop-up window shown in Display 2.6 on page 19 to be displayed. After you read a pop-up window, press the ENTER key to close the pop-up window and return to the card from which it was selected.

Display 2.6 Help Window, Pop-Up Window



```
Help
                         HELP COMMAND

    Format 1:  help [DEBUGGER-CMD-NAME/WINDOW-NAME/TOPIC]

    DESCRIPTION

       The help command can be used to access the help system index or
       it can be used with one of the optional arguments to access
       information about a specific command, window, or topic.

    EXAMPLES

       help
            opens the Help window and displays the INDEX.

       help break
            opens the Help window and displays the card for the break
            command.

       help config window
            opens the Help window and displays the card for the
            Configuration window.

       help directing commands

 PF05 window resize <>
Cdebug:
```

Using the Primary Windows

The SAS/C Debugger gives you maximum control over your debugging session. It also provides you with a wealth of information about the program you are debugging. The key to harnessing this capability is the windowing interface.

The Command, Log, Status, and Source windows are your primary interface to the SAS/C Debugger when it is running in full-screen mode.

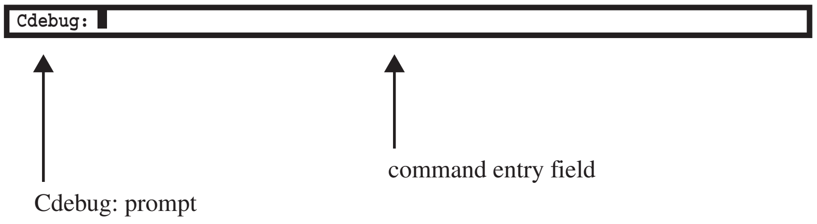
Using the Command Window

The SAC/C Debugger offers a powerful set of commands that are used to control your debugging session. When the debugger is running in full-screen mode, you submit these commands either from the command window or by pressing a PF key.

Note: The Keys window is used to assign debugger commands to PF keys. For a quick look at the default command assignments, see “Using PF Keys” on page 27. △

The Command window, shown in Display 2.7 on page 20, is used to submit debugger commands. The method is similar to the way that you submit commands in a line-mode session. You can issue any line-mode commands after the `Cdebug:` prompt and then submit the command to the debugger by pressing the ENTER key. As the debugger executes the command, it may display output or messages about the execution in the Log window or in one of the other windows that are described in “Types of Windows” on page 14.

Display 2.7 The Command Window



```
Cdebug: █
```

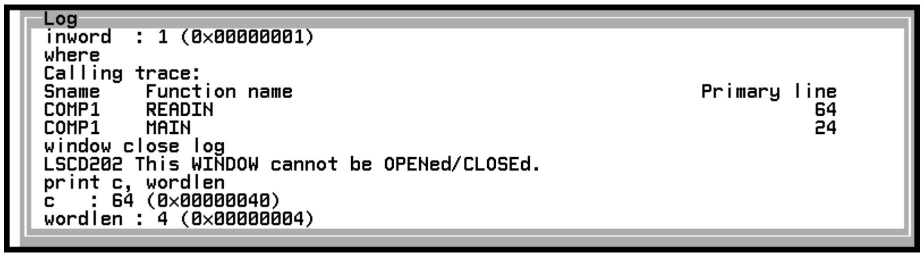
If a command is too long for the Command window to display, you can type the command into the Log window, using a backslash for continuation.

The debugger maintains a circular list of the commands that you issue.. However, issuing a command more than once in succession results in only one copy of the command being maintained in the list. You can cycle through the list by using the **`window scroll up`** and **`window scroll down`** commands, which are assigned to the PF19 and PF20 keys by default. To issue a previously issued command, press the ENTER key when the command is displayed. The previously entered commands are accessed in first-in, first-out order. As the list fills up, the oldest commands are deleted. Use the **`window clear`** command to clear the list.

Using the Log Window

As shown in Display 2.8 on page 20, the Log window contains a log of commands that are issued during the current session. It also displays output from certain commands and some error messages. Commands that are echoed in the Log window are those commands that are issued either in the Command window or in the Log window. The **`window scroll up`** and **`window scroll down`** commands provide the capability to view previous commands and their output. The PF19 and PF20 keys are dedicated by default to vertically scrolling the window that contains the cursor.

Display 2.8 The Log Window



```
Log
inword  : 1 (0x00000001)
where
Calling trace:
Sname    Function name                                    Primary line
COMP1    READIN                                                     64
COMP1    MAIN                                                       24
window close log
LSCD202 This WINDOW cannot be OPENed/CLOSEd.
print c, wordlen
c   : 64 (0x00000040)
wordlen : 4 (0x00000004)
```

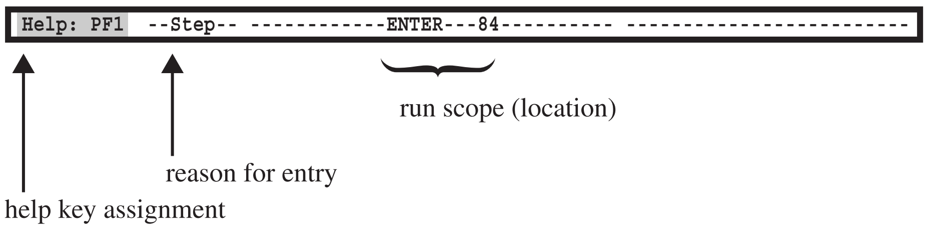
Using the Status Window

During a debugging session, control passes back and forth between your program and the debugger. See Display 2.9 on page 21. The Status window provides information about the current status of your debugging session, including your current location in the code. This information consists of the following:

- □ the number of the PF key that is assigned to function as the help key.
- □ the reason for transferring control to the debugger (whether it was the result of a break, step, continue, monitor, attention, or a similar event).

□ the location in your code at which the debugger was entered: the function name, and either the line number stopped at or an identification of the side of a function call or return that the execution stopped on. This location is also called the *run scope*.

Display 2.9 The Status Window



```
Help: PF1  --Step-- ------------ENTER---84---------- ------------------------
```

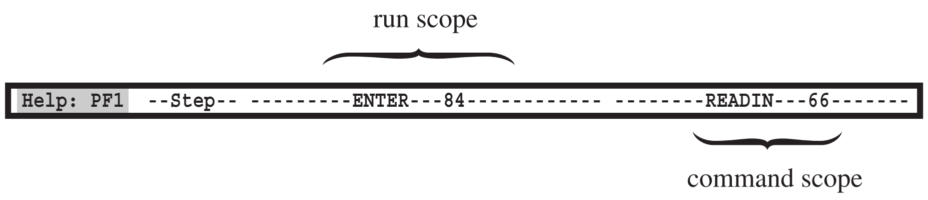
Transferring Control to the Debugger

The point in your code at which control can be passed from your program to the debugger is called a *hook*. Whenever control is passed to the debugger, the Status window displays the location of the hook and the reason that control was transferred. For example, in the Status window shown in Display 2.10 on page 21, a **`step`** command that was issued at the entry hook of the **`getname`** function caused control to be transferred to the debugger.

Run Scope and Command Scope

The location in your program at which execution stops determines the run scope. The debugger usually uses this location to resolve any references to variables. The debugger recognizes expressions that are visible to your programs at the point where execution stops without any additional actions on your part. However, if you want to view the value of expressions that are not visible at the point in your code indicated by your run scope, you must change the scope to a new location. This new scope, which you control, is called the command scope. Display 2.10 on page 21 shows a Status window that displays both a run scope and a command scope.

Display 2.10 Status Window, Showing Run Scope and Command Scope



```
Help: PF1  --Step-- ---------ENTER---84------------ --------READIN---66-------
```

There are two ways in which you can change the command scope. The **`scope`** command, which is described in Chapter 14, “Command Directory,” on page 187, provides the greatest control. However, if you have more than one function in your calling sequence, you can move the cursor into the Status window and use the PF19 or PF20 keys to change command scope. The **`window scroll <> up`** or **`window scroll <> down`** commands are assigned to these keys by default. The PF19 key causes the command scope to move up in the calling sequence, and the PF20 key causes the run scope to move down in the calling sequence.

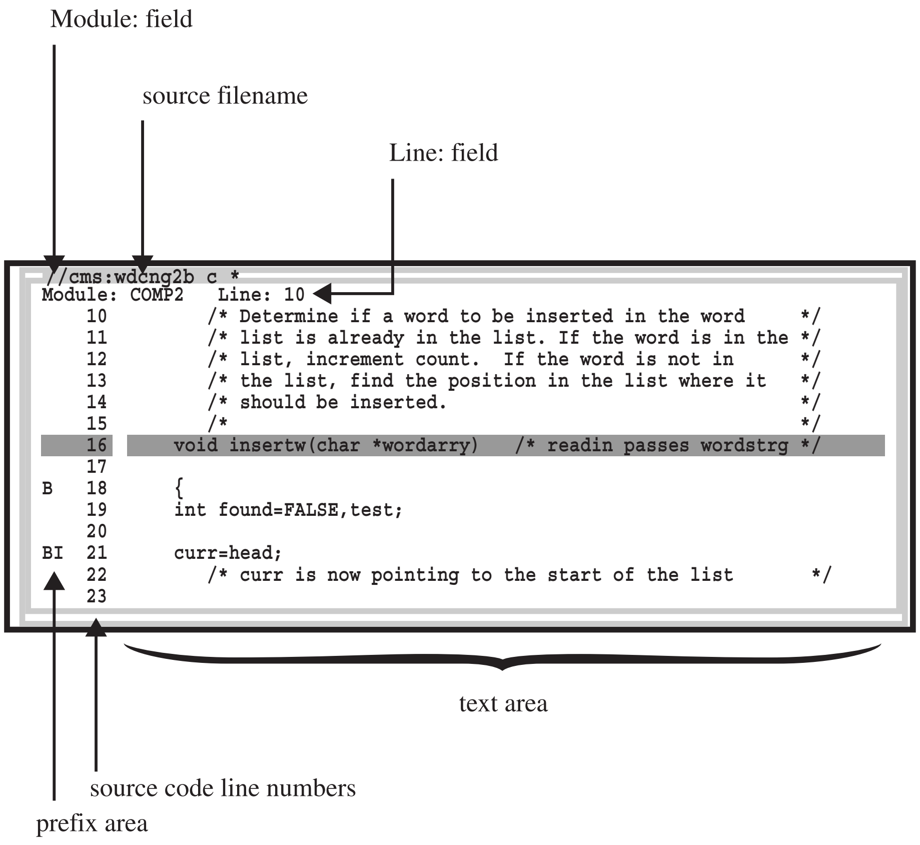
Using the Source Window

The Source window displays your source code and highlights the line on which the debugger has stopped, as shown in Display 2.11 on page 22. The top border, which is always present, contains the name of the source file.* The first line of the window provides information that always displays the following:

Module
: identifies the compilation that is currently displayed in the source window

Line
: is the line number of the first line of source code that is displayed in the visible portion of the Source window text area.

Display 2.11 The Source Window



```
 //cms:wdcng2b c *
Module: COMP2   Line: 10
    10        /* Determine if a word to be inserted in the word     */
    11        /* list is already in the list. If the word is in the */
    12        /* list, increment count.  If the word is not in      */
    13        /* the list, find the position in the list where it   */
    14        /* should be inserted.                                */
    15        /*                                                    */
    16     void insertw(char *wordarry)   /* readin passes wordstrg */
    17
B   18     {
    19     int found=FALSE,test;
    20
BI  21     curr=head;
    22        /* curr is now pointing to the start of the list       */
    23
```

The remaining portion of the Source window contains text and line number areas that are used to view source code. The left side of each line number field is a prefix area. The prefix area displays the location of requests that have been assigned to a specific line in your code.

The SAS/C Debugger uses a request system that keeps track of the breakpoint and *action* requests that you specify with debugger commands. These requests, which tell the debugger to interrupt program execution at a hook, are assigned to one line or a range of lines in your source code. As shown in Display 2.11 on page 22, the prefix area

* The form of the source filename displayed in the Source window is operating-environment specific. Display 2.11 on page 22 illustrates a CMS source filename.

includes an indication of the commands that have been requested, such as **`break`**, **`ignore`**, **`on`**, and **`trace`**. The prefix area can also be used to issue prefix-area commands that are used to either make or control debugger requests. Prefix-area commands are explained under "Source Window" on page 173.

The code that is displayed in the Source window changes as debugger commands are executed. When you issue **`step`**, **`go`**, **`continue`**, or any other command that causes additional lines of your program to run, the highlighted line advances through the text field of the Source window. If the cursor is in the text field, it stays in step with the highlighted line as your program runs. Leaving the cursor on any of the other areas disables this tracking behavior.

Note: The **`print`** command is assigned to the PF16 key by default. If you move the cursor to a variable that is located within the run scope and you issue the **`print`** command with a PF key, then the log window displays the value of the variable. The tracking behavior of the cursor within the Source window is particularly useful with this technique. △

Moving Around in the Source Window

To view your source code, you can either scroll through the window or jump directly to a specific line. After viewing a portion of your source code, you can use the **`list`** command to return to your current line.

Scrolling By scrolling through the text field of the Source window, you can view any line of your source code. The following **`window scroll`** commands are assigned to PF keys by default:

PF7	**`window scroll source up`**
PF8	**`window scroll source down`**
PF 19	**`window scroll <> up`**
PF 20	**`window scroll <> down`**
PF 22	**`window scroll <> left`**
PF 23	**`window scroll <> right`**

The **<>** symbol is used as a placeholder to specify the window in which the cursor is currently located. When you press the PF key, the name of your current window is substituted for **<>**, and the command is executed. See "Placeholders in Commands" on page 150 and "window" on page 280 for additional information about the **<>** placeholder.

You must position the cursor in the Source window before you use the PF19, PF20, PF22, or PF23 keys. However, the PF7 and PF8 keys have been assigned **`window scroll`** commands that cause the Source window to be scrolled regardless of the position of the physical cursor. These two keys are useful when your cursor is located in the Command window and you want to scroll the source code without moving the cursor into the Source window. See Chapter 14, "Command Directory," on page 187 for additional information about the **`window scroll`** command.

As you scroll through your source code, the line number and text areas scroll together. Although the text field can be scrolled either horizontally or vertically, the line number area can be scrolled only vertically. Thus, the line number is always visible, even if you scroll to the far right side of the text area.

Scroll amount By default, the position of the cursor in a window controls the amount that is scrolled up and down. You can override the default scroll value during a session by issuing a **`window scroll`** command or specifying a new value in the Config window. Valid values are **`cursor`**, **`half`**, **`page`**, and **`max`**.

The initial scroll amount, which has a default value of cursor, can also be set by a **`window scroll`** command in the configuration file. A scroll amount of **`max`** cannot be specified in the configuration file. The configuration file is used to customize the attributes of the windowing interface. See "Setting Up a Configuration File" on page 46 for more information.

Jumping to a line Scrolling is a simple and fast way to move through displayed lines. However, it is not an effective way to reach parts of your code that are located far from your current position. Another way to view a part of the displayed module is to type the line number at the **`Line:`** prompt of the Source window. Sources from a different module in the calling sequence can be viewed by typing the module name after the **`Module:`** prompt. If either the module name or line number is invalid, a window pops up so that you can correct the invalid input.

Note: You can use the **`list`** command to produce the same effect during a full-screen session. See Chapter 14, "Command Directory," on page 187 for information about the **`list`** command. △

Returning to the highlighted line After viewing source code from different modules, or viewing a different area of the current source file, you can quickly return to the highlighted line by issuing the **`list`** command with no parameters in the Command window.

Controlling the Windowing Interface

During a debugging session, you can control the windowing interface by

- □ opening, closing, moving, resizing, and zooming windows
- □ directing output to dedicated windows
- □ changing PF key definitions.

Opening and Closing Windows

Most windows can be opened with the **`window open`** command and closed with the **`window close`** command. You cannot use these commands to open and close the four primary windows, Message windows, or Pop-up windows.

To be more precise, **`open`** and **`close`** are actually subcommands of the **`window`** command. The **`window`** command is used to issue a number of subcommands, all of which control the windowing interface. Most of these subcommands can be issued from either the Command window or a configuration file. Refer to "Setting Up a Configuration File" on page 46 for more information about using a configuration file to customize the windowing interface. Also see Chapter 14, "Command Directory," on page 187 for more information about the **`window`** command and its subcommands.

The general form for using the **`window`** command to open a window is as follows:

`window open` *WINDOW-NAME*

`window close` *WINDOW-NAME*

In either case, *WINDOW-NAME* identifies the window to be opened and can be any of the following:

Browse	Config	Dump
Help	Keys	Print
Register	Termin	Termout
Watch		

Moving, Resizing, and Zooming Windows

With the exception of Message and Pop-up windows, all windows can be moved with the **`window move`** command, resized with the **`window resize`** command, and zoomed with the **`window zoom`** command. These commands can be executed from the Command window; however, the easiest way to accomplish these tasks is to use PF keys. By default, the following commands are assigned to PF keys:

PF2	**`window move`** <>
PF14	**`window resize`** <>
PF13	**`window zoom`** <>

Moving a Window

To move a window, perform the following steps:

1. Place the cursor in the window you want to move. The cursor can be placed anywhere inside the window or on the border.
2. Press the PF2 key. **`MOVE`** is displayed in the lower-right border of the window to indicate that a move is pending.
3. Use the arrow keys to move the cursor to the desired location for the window. The window does not follow the cursor until the next step.
4. Press the ENTER key. The window moves to the desired location.

Resizing a Window

To resize a window, perform the following steps:

1. Place the cursor on the border of the window you want to resize. The cursor can be placed anywhere inside the window, but resizing is much easier if you place it on one of the borders.
2. Press the PF14 key. **`RESIZE`** is displayed in the lower-right border of the window to indicate that a resize is pending.
3. Use the arrow keys to move the cursor to the desired location for the window border. The border does not follow the cursor until the next step.
4. Press the ENTER key. The border moves to the desired location.

Zooming a Window

When you zoom a window, it fills the entire display area. Other windows are hidden behind the zoomed window. To zoom a window, perform the following steps:

1. Place the cursor in the window you want to zoom. The cursor can be placed anywhere inside the window or on the border.
2. Press the PF13 key. The window zooms out to fill the entire display area or to its maximum size, whichever is smaller.

3 Press the PF13 key a second time to restore the original display.

Directing Commands to a Window

Usually any output that is generated by debugger commands is displayed in the Log window. For example, you can use the **print** command to display the value of a variable, and, unless you specify otherwise, the value of that variable is displayed in the Log window. However, certain commands, such as the **print** command, can be directed to a dedicated window.

The **print** command can be directed to the Print window, the **dump** command to the Dump window, and the **keys** command to the Keys window by using the following command prefixes:

redirect command (>)	opens a new Print, Dump, or Keys window and directs output to it. Only the Keys window can be opened.
redirect command (>>)	directs output to a previously opened Print, Dump, or Keys window. If the window has not been opened, this prefix command opens a new window exactly as the > prefix does.

Note: The > and >> symbols are prefixed to either the **print**, **dump**, or **keys** command. The syntax for the command does not change. △

For example, either of the following commands directs a variable named *my_variable* to the Print window:

> print *my_variable*

>> print *my_variable*

The first command opens a new Print window, and the second command reuses an open Print window.

You can open several Print or Dump windows with the > command prefix; however, only one Keys window can be opened. Both the > and >> command prefixes have the same effect on the **keys** command if a Keys window is already opened: the existing window is reused.

Anytime a dedicated window is either opened or reused, it automatically becomes the top window. That is, the physical cursor is placed inside the window, which is then placed on top of the stack of windows that are currently open.

Changing the Window Configuration

The size, position, and display attributes of debugger windows can be controlled with the Config window. You can open the Config window with the **window open config** command and change the display characteristics for any window at any point in your debug session. These changes can be saved to your configuration file if you want to make them permanent. The Config window allows you to customize the following parameters that affect the configuration of the debugger:

AUTOPOP
: can be set for each window. If a window is set to autopop, it automatically becomes the top window whenever output is sent to it.

BORDER
: specifies whether border characters are in hexadecimal or character format. The same characters are used for all windows with borders.

COLORING
: specifies the color, attributes, and intensity for each area in each window. There is a field in the Config window that contains a description of the target area. The

field is colored in the same way as the target area. Changes in color, attributes, or intensity are immediately reflected in this field and in the target area.

CONFIGURATION
specifies the configuration (size, position, and presence of borders) of each window.

CONTEXT
contains parameters that control the number of context lines in the Source window. These parameters take effect the next time the Source window is updated.

MEMORY
specifies the memory that is allocated to the buffers of the Command, Log and Source windows. However, the memory that is used by this debugging session is not dynamically reallocated: if you save the configuration with the changed setting, the changed memory values are used the next time you run the debugger.

SCROLL AMOUNT
specifies the default scroll amount.

TRACE LOG
turns on and off the trace status of the Log window.

Certain types of windows, such as the Print window, allow you to open several windows of that type simultaneously. However, other types of windows, such as the Log window, allow only one instance at a time. You cannot have two Log windows open simultaneously.

For windows that allow only one instance, changing configuration parameters results in that window being closed and reopened. If the change results in borders being added, the reopen fails if the resources required to display the window exceed the capabilities of the debugger. This can happen when a large number of windows are open. See "Number of Open Windows" on page 37 for a discussion of this limitation. However, failure to reopen one of the four basic windows is severe enough for the debugger to reopen the window without a border.

For windows that allow several instances to be displayed simultaneously, such as the Print window, changing configuration parameters has no effect on windows of that type that are already open. New instances of that type of window are displayed using the new parameters.

You can use the config window to change the size or position of a window. However, information in the window may move, or the moved or resized window may appear on top of the Config window. Therefore, using PF keys is the preferred way to move or resize windows.

To save your current configuration to a file, specify the file name and type in Y after the **`Save`**: prompt on the first line of the Config window. "window" on page 280 has complete details on the various customization parameters that are saved when the current configuration is written to a file. Those that may be set in the Config window are only a subset of those that are saved.

Using PF Keys

PF keys offer the fastest and easiest way to issue some of the debugger commands in a full-screen session.* The default PF key command assignments are adequate for most debugging tasks; however, you can reassign debugger commands to PF keys by using either the **`keys`** command or the Keys window.

* Debugger PF key command assignments are used only in full-screen mode. If you switch to a line-mode session, the PF keys are not used.

Default PF Key Commands

The debugger maintains two sets of tables for key assignments, the current set and the default set. On start-up, both sets of tables are identical to the command assignments that are shown in Table 2.1 on page 28.

Table 2.1 Default PF Key Commands

Key	Command	Action
PF1	**help**	open Help Window
PF2	**window move < >**	move current window
PF3	**exit**	exit debugger
PF4	**> dump < > str**	dump memory contents pointed to by expression under cursor
PF5	**/**/**	none (may be used in a future release)
PF6	**/**/**	none (may be used in a future release)
PF7	**window scroll source up**	scroll Source window up
PF8	**window scroll source down**	scroll Source window down
PF9	**window next**	jump to next window in stack
PF10	**continue**	resume execution and break at next line-number hook without stepping into functions
PF11	**step**	resume execution and break at next hook
PF12	**go**	resume execution
PF13	**window zoom < >**	zoom current window
PF14	**window resize < >**	resize current window
PF15	**window close < >**	close current window
PF16	**print < >**	display value of variable under cursor
PF17	**window find <>**	find in the current window
PF18	**/**/**	none (may be used in a future release)
PF19	**window scroll < > up**	scroll current window up
PF20	**window scroll < > down**	scroll current window down
PF21	**window previous**	jump to previous window
PF22	**window scroll < > left**	scroll current window left
PF23	**window scroll < > right**	scroll current window right
PF24	**window top command**	jump to Command window

You can use a configuration file to modify the command assignments to both sets of tables. After the configuration file is executed, both sets of tables are identical and contain the default assignments you have specified. See “Setting Up a Configuration File” on page 46 for more information.

You can also change the command assignment for a PF key outside the configuration file (sometime after starting your debugging session). These changes affect only your current session unless you choose to save the new configuration with the **`config save`** command.

Using the Keys Command

Any of the following **`keys`** commands can be used to list or modify PF key assignments. You can issue these commands either in the configuration file or from the Command window.

`keys`
is an alias for **`keys list *`**.

`keys list` *n*
lists the key definition for *PFN* from the current set of tables.

`keys list *`
lists all key definitions from the current set of tables.

`keys default` *n*
gives *PFN* the default definition. The debugger copies the definition from the default set of tables to the current set of tables.

`keys default *`
gives all keys their default definitions. The debugger copies the definitions from the default set of tables to the current set of tables.

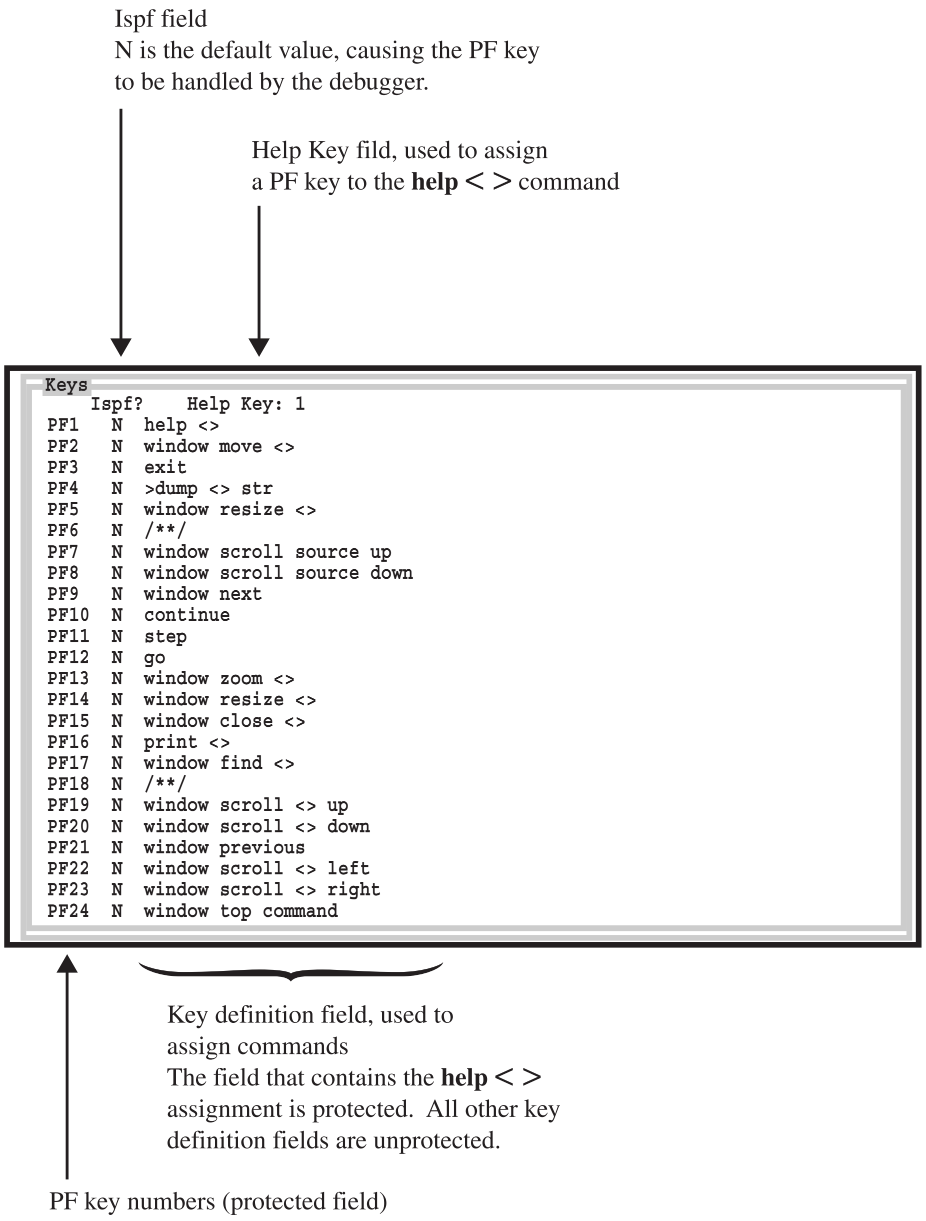
`keys define` *n "text"*
in the configuration file, changes the definition for *PFN*, both in the default set of tables and in the current set of tables. Outside of the configuration file, this changes the definition only in the current set of tables. The changed definition is specified by *text*, which may not exceed 80 characters.

Using the Keys Window

You can also use the Keys window to display and change current PF key assignments. Open the Keys window either by issuing a **`window open keys`** command or by redirecting any of the **`keys`** commands to a window. This is done by prefixing the command with a **>** or a **>>**. (See "Directing Commands to a Window" on page 26.)

The Keys window, shown in Display 2.12 on page 30, comprises four fields: a protected field that identifies the keys, and three unprotected fields. The **`Help Key`** field is used to assign a key to the **`help <>`** command, and the key definition field is used to assign debugger commands to the other keys. The **`ISPF`** field is used only when running the debugger under ISPF, as described in Appendix 4, "Debugger ISPF Interface."

Display 2.12 The Keys Window


Ispf field
N is the default value, causing the PF key to be handled by the debugger.
Help Key fild, used to assign a PF key to the **help** < > command
Key definition field, used to assign commands
The field that contains the **help** < > assignment is protected. All other key definition fields are unprotected.
PF key numbers (protected field)

```
Keys
     Ispf?    Help Key: 1
PF1   N  help <>
PF2   N  window move <>
PF3   N  exit
PF4   N  >dump <> str
PF5   N  window resize <>
PF6   N  /**/
PF7   N  window scroll source up
PF8   N  window scroll source down
PF9   N  window next
PF10  N  continue
PF11  N  step
PF12  N  go
PF13  N  window zoom <>
PF14  N  window resize <>
PF15  N  window close <>
PF16  N  print <>
PF17  N  window find <>
PF18  N  /**/
PF19  N  window scroll <> up
PF20  N  window scroll <> down
PF21  N  window previous
PF22  N  window scroll <> left
PF23  N  window scroll <> right
PF24  N  window top command
```

If the height of the window is smaller than that needed to display all of the PF key definitions, you can view information that is not visible by scrolling vertically using the the **`window scroll up`** and **`window scroll down`** commands. The PF19 and PF20 keys are assigned to these commands by default. You can also use the **`window zoom`** command, the effect of which is shown in Display 2.12 on page 30.

PF key definitions can be changed by typing over the current definition. Blanking out (erasing) the field gives the default definition. (When erasing a field, make sure that the field has been scrolled all the way to the left.) Each key definition can be up to 80 characters long. The definition field may be scrolled horizontally, using the **`window scroll left`** and **`window scroll right`** commands.

Customizing the Keyboard

You can easily customize a keyboard by creating a configuration file that contains **`keys define`** *n “text”* commands. However, note that customized PF key definitions work only in full-screen mode. Refer to “Setting Up a Configuration File” on page 46 for more information about using a configuration file.

Switching Between Full-Screen Mode and Line Mode

At any time during a debugging session, you can switch between full-screen mode and line mode using the **`window off`** and **`window on`** commands.

`window off`
terminates the windowing interface and continues the debugging session in line mode, preserving the states of the Log and Command window buffers.

`window on`
starts up the windowing interface. If full-screen mode was used earlier in this invocation of the debugger, the configuration last used determines the setup, and the contents of the Log and Command window buffers reappear unchanged. However, if this is the first time that full-screen mode is used, your default initial configuration is used. See “Setting Up a Configuration File” on page 46 for more information.

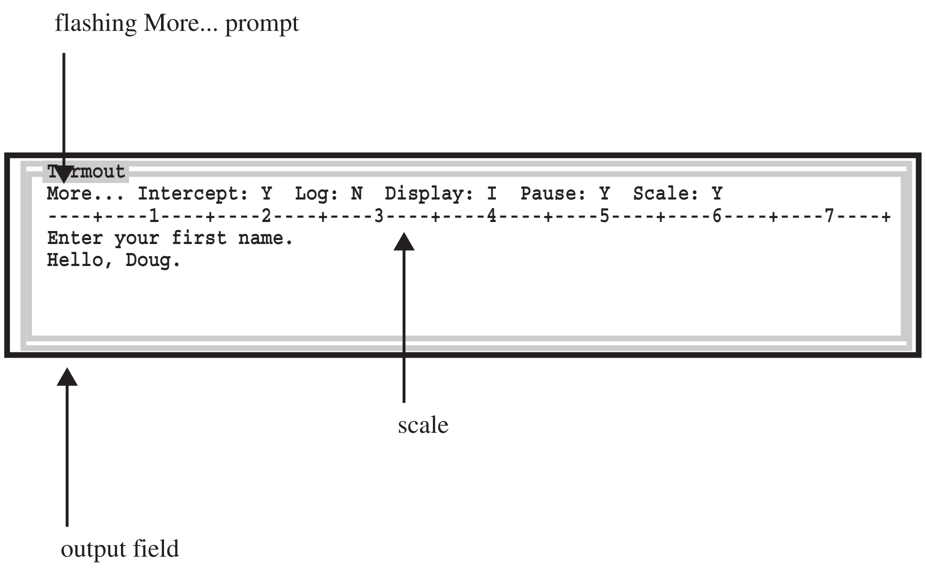
Looking at Terminal I/O

Most programs involve some input or output to the terminal. The SAS/C Debugger provides two windows that are specifically designed to enable efficient terminal I/O debugging. Terminal output from functions such as **`printf`** can be displayed in the Termout window, and terminal input from functions such as **`gets`** can be entered from the Termin window.

Using the Termout Window

As shown in Display 2.13 on page 32, there are three areas in the Termout window: a status and prompt line, an optional scale or ruler line, and the output field.

Display 2.13 The Termout Window



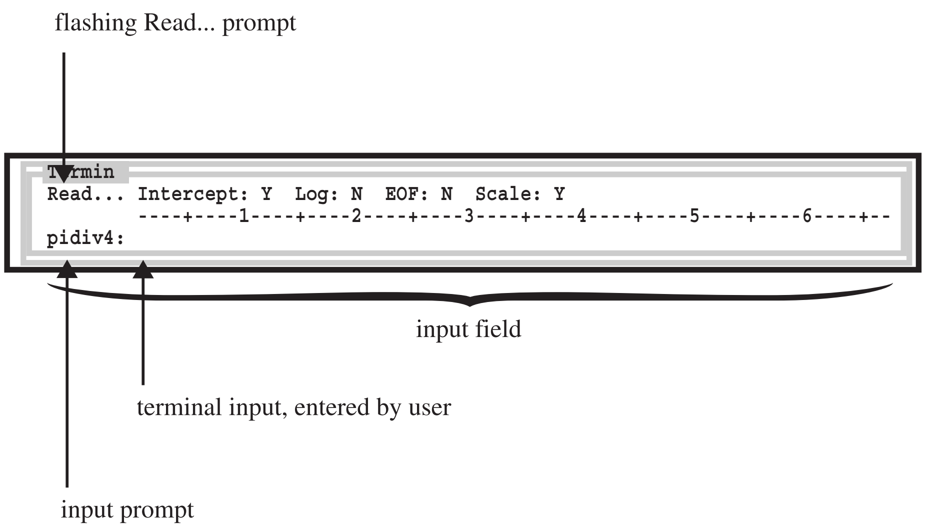
```
Termout
More... Intercept: Y  Log: N  Display: I  Pause: Y  Scale: Y
----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+
Enter your first name.
Hello, Doug.
```

Use the **`window open`** command to open the Termout window, as previously described in "Opening and Closing Windows" on page 24. However, in full-screen mode the debugger usually intercepts program output and displays it in the Termout window automatically. Switching to line mode turns the intercept off, and switching back to full-screen mode restores the previous status of the intercept.

Using the Termin Window

As shown in Display 2.14 on page 32, there are three areas in the Termin window: a status and prompt line, an optional scale or ruler line, and the input field, which includes the input prompt.

Display 2.14 The Termin Window



```
Termin
Read... Intercept: Y  Log: N  EOF: N  Scale: Y
        ----+----1----+----2----+----3----+----4----+----5----+----6----+--
pidiv4:
```

By default, the debugger intercepts program input requests, opens the Termin window, and prompts you for input. Switching to line mode turns the intercept off, and returning to full-screen mode restores the previous status of the intercept.

Looking at Variables, Memory, and Registers

Much of the power of a debugger comes from its enabling you to examine the values assigned to variables, dump the contents of memory, and look at registers. For example, as you step through a program you can examine the contents of an array to see whether it is initialized as you intended.

The SAS/C Debugger includes four windows that are especially useful when you need to take a closer look at your program and the values that it is manipulating. This section explains how to use the Print, Watch, Dump, and Register windows.

Using the Print Window

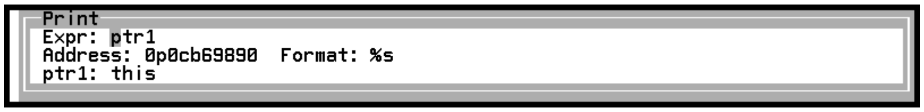
The Print window can be used to display the value of an expression. To use the Print window, you direct the output of the **`print`** command to the Print window as previously described in “Directing Commands to a Window” on page 26. For example, in Display 2.15 on page 33 the string pointed to by an expression named **`ptr1`** was displayed by issuing the following command:

```
> print ptr1 %s
```

Use the command prefix **`>`** to redirect output from the **`print`** command to a new Print window. You can use **`>>`** to redirect output to an existing Print window.

Also notice that you can specify format when directing the **`print`** command to the Print window. In the previous command, **`%s`** specifies that **`ptr1`** should be formatted as a string. You can use any of the format specifiers that are valid with the **`sprintf`** function. See “print” on page 241 for more information about using format specifiers with the **`print`** command.

Display 2.15 The Print Window

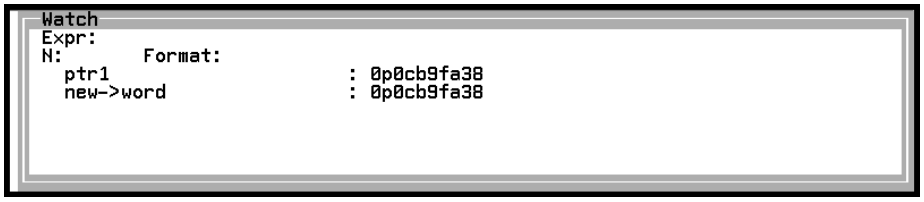


```
Print
Expr: ptr1
Address: 0p0cb69890  Format: %s
ptr1: this
```

Using the Watch Window

The Watch window is used to track the value of an expression or an area of memory during your debugging session. It acts like an automatic **`print`** or **`dump`** command, displaying the expression or area of memory each time control is transferred to the debugger. As shown in Display 2.16 on page 34, you can specify several watches, each of which is displayed in the Watch window.

Display 2.16 The Watch Window



```
Watch
Expr:
N:       Format:
  ptr1                    : 0p0cb9fa38
  new->word               : 0p0cb9fa38
```

The **watch** command can be used to specify a watch; however, the easiest way to specify a watch is to open the Watch window and then set your watches by using the **Expr:**, **N:**, and **Format:** fields. You can use a prefix field in order to drop watches. These fields are described in “Watch Window” on page 184.

Using the Dump Window

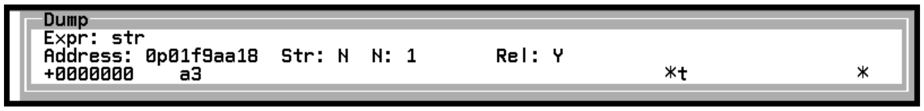
The Dump window is used inorder to display a dump of memory in both character and hexadecimal format. This window is useful when you need to examine a region of memory for possible address space conflicts. For example, it can help you determine why a portion of an array is being overwritten. It can also help isolate the cause of “garbage” information in your data structures.

Output from the **dump** command is directed to the Dump window in much the same way as output from the **print** command is directed to the Print window. The **>** and **>>** command prefixes are used with the **dump** command to direct the output from a memory dump to either a new or existing Dump window, as described earlier in “Directing Commands to a Window” on page 26. For example, the following command dumps 80 bytes of memory that is pointed to by an expression named **str**:

```
> dump str 80
```

The output from this **dump** command is directed to a Dump window as illustrated by Display 2.17 on page 34.

Display 2.17 The Dump Window



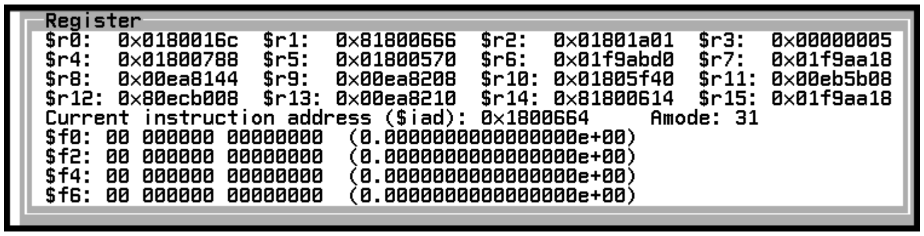
```
Dump
Expr: str
Address: 0p01f9aa18  Str: N  N: 1       Rel: Y
+0000000    a3                                   *t              *
```

As shown in Display 2.17 on page 34, the relative address is displayed on the left side of the Dump window, a hexadecimal representation of the contents of memory is displayed in the middle, and an EBCDIC character representation is displayed on the right side.

Using the Register Window

The Register window, shown in Display 2.18 on page 35, enables you to view the contents of the 16 general-purpose registers and the 4 floating-point registers. It also displays the current instruction address and the address mode.

Display 2.18 The Register Window



```
Register
$r0:  0x0180016c  $r1:  0x81800666  $r2:  0x01801a01  $r3:  0x00000005
$r4:  0x01800788  $r5:  0x01800570  $r6:  0x01f9abd0  $r7:  0x01f9aa18
$r8:  0x00ea8144  $r9:  0x00ea8208  $r10: 0x01805f40  $r11: 0x00eb5b08
$r12: 0x80ecb008  $r13: 0x00ea8210  $r14: 0x81800614  $r15: 0x01f9aa18
Current instruction address ($iad): 0x1800664     Amode: 31
$f0: 00 000000 00000000  (0.0000000000000000e+00)
$f2: 00 000000 00000000  (0.0000000000000000e+00)
$f4: 00 000000 00000000  (0.0000000000000000e+00)
$f6: 00 000000 00000000  (0.0000000000000000e+00)
```

The **`window open register`** command must be used to open the Register window. Issuing this command when the Register window is open updates the window. Pressing ENTER when the cursor is in the Register window also updates it.

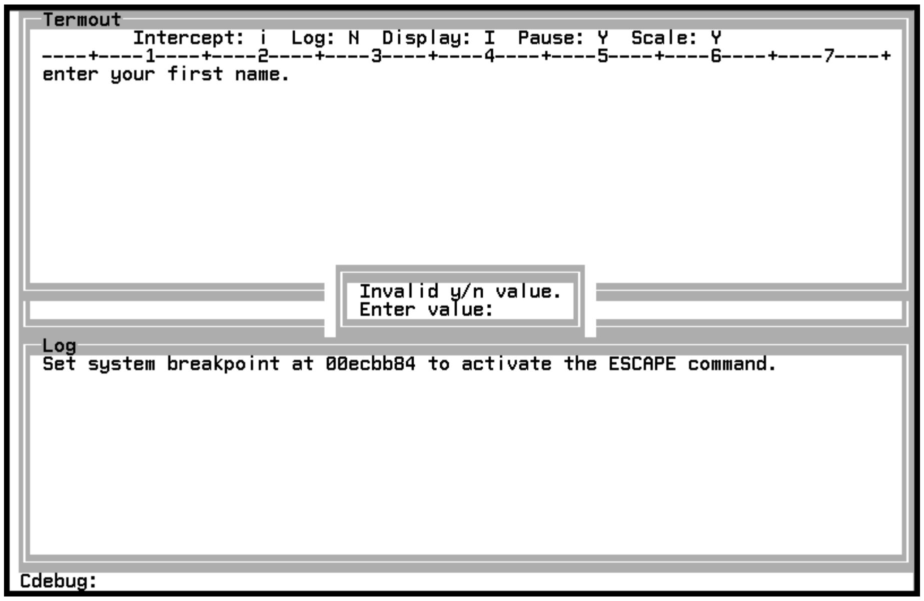
Pop-Up and Message Windows: Error Processing

In general, Pop-up windows are used to correct invalid window input, and Message windows are used to display error messages that are associated with invalid commands. This section describes how the Pop-up and Message windows are used in error processing. See Appendix 1, “Error Handling,” on page 297 for general information about error conditions.

Pop-Up Windows

If invalid input is specified in certain areas of certain windows, an alarm sounds, and a Pop-up window opens automatically. For example, Display 2.19 on page 35 shows a Pop-up window that is displayed when an invalid input of **`Z`** is typed in the **`Intercept`** field of the Termout window. The message in the Pop-up window describes the invalid input. To close the window, you must type either a valid value or a blank. See “Popup Window” on page 170 for more information.

Display 2.19 A Pop-Up Window

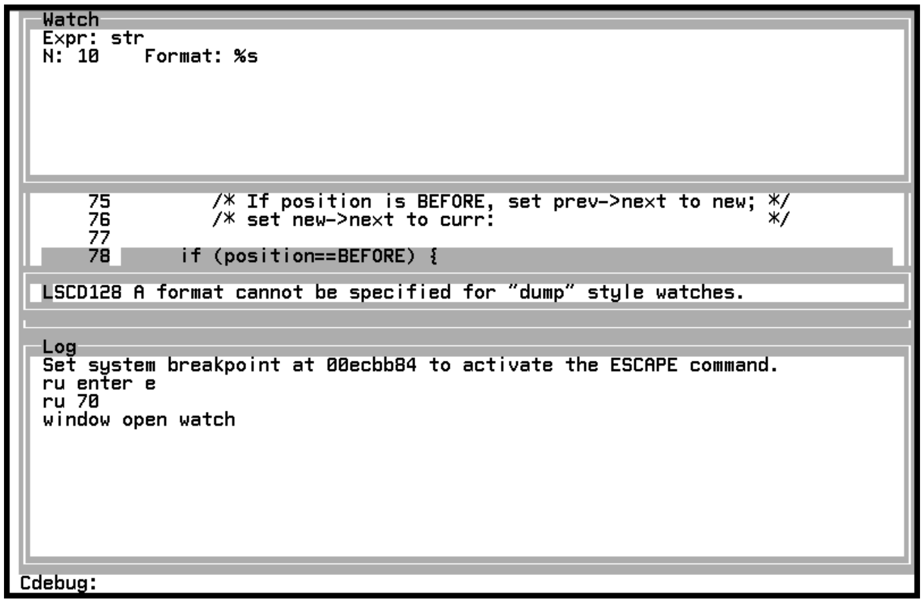


```
Termout
       Intercept: i  Log: N  Display: I  Pause: Y  Scale: Y
----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+
enter your first name.

                        Invalid y/n value.
                        Enter value:
Log
Set system breakpoint at 00ecbb84 to activate the ESCAPE command.

Cdebug:
```

Using Message Windows

If invalid input is specified in the input area of any window other than the Command or Log windows, the debugger opens a Message window automatically to display an error message. For an example see Display 2.20 on page 36. After viewing the message, you can press ENTER or any PF key to close the window. Pressing a PF key that contains an invalid command also causes the debugger to open a Message window that contains an error message. See "Message Window" on page 169 for more information.

Display 2.20 A Message Window



```
Watch
Expr: str
N: 10    Format: %s

   75        /* If position is BEFORE, set prev->next to new; */
   76        /* set new->next to curr:                        */
   77
   78     if (position==BEFORE) {

LSCD128 A format cannot be specified for "dump" style watches.

Log
Set system breakpoint at 00ecbb84 to activate the ESCAPE command.
ru enter e
ru 70
window open watch

Cdebug:
```

Factors Affecting Your Full-Screen Session

You should be aware of the following two factors that affect your full-screen session:

- □ The debugger uses a priority sequence to process simultaneous input from windows and PF keys.
- □ There is a limit to the number of windows that you can display at one time. This number varies, depending on the type of windows that you display.

Window and PF Key Priorities

When input is specified in more than one window, the debugger processes windows according to a sequence determined by window priorities. Generally, a window of a higher priority is processed before a window of a lower priority. The debugger sets priorities that cannot be changed. The following list of windows is in order from highest to lowest priority:

1. Keys
2. Watch
3. Termout
4. Termin

5 Status
6 Source
7 Log
8 Command
9 Config
10 Register
11 Dump
12 Print
13 Watch

PF keys have a lower priority than any of the windows. Consequently, if input is specified in a window and a PF key pressed, the window input is processed first and then the PF key command is processed.

The only exception to these priorities occurs when there is text after the **`Cdebug:`** prompt in the Command window and a PF key is pressed while the cursor is inside the Command window. In this case, the text is completely ignored and the PF key is processed. This enables command scrolling in the Command window: you can use the PF19 and PF20 keys to scroll through or recall previously issued commands. (The PF19 and PF20 keys are assigned the **`window scroll < > up`** and **`window scroll < > down`** commands by default.)

Number of Open Windows

There is a limit to the number of windows that the debugger can display at one time. The limit depends on the type of windows that are being displayed and the display attributes of those windows. To avoid reaching the debugger's display limit, close windows that are no longer necessary. By default, the PF15 key is assigned to the **`window close < >`** command.

You must be especially careful with the Print and Dump windows. Directing output to these windows with the > command prefix causes a new window to be opened. It is possible to open a large number of these windows at one time, each of which overlays the previous window. When either the **`print`** or **`dump`** command is used in conjunction with the **`on`** command, it is easy to open a large number of windows.

Restoring Control and Exiting the Debugger

This section explains how to restore control to the debugger when, for example, a program that you are running is not responding, or is "hung" in an infinite loop. It also explains how to exit the debugger.

Attention Key

The attention key can be used to restore control to the debugger when a problem is encountered during a debugging session. Pressing the attention key has various consequences,which depend on

- which debugger command you have issued
- the type of problem that you are debugging (an infinite loop, for example)
- whether you are running the SAS/C Debugger with your operating environment debugger.

Under TSO, for example, the most general case is that user control is returned when you press the attention key. Under CMS, pressing the attention key toggles CP and CMS.

Exiting the Debugger

The **`exit`** command is used to exit the debugger. It is assigned to the PF3 key by default, or it can be issued from the Command window. See Chapter 14, “Command Directory,” on page 187 for complete details of the **`exit`** command.

P A R T 2

Configuring and Using the Debugger

CHAPTER

3

Debugger PROFILEs, Configuration Files, and EXECs

Introduction

The SAS/C Debugger uses your PROFILE and a configuration file to execute debugger commands and determine your initial configuration upon entry to the debugger. This chapter explains how to do the following:

- ☐ set up a PROFILE containing debugger commands that are issued at initial entry to your program
- ☐ set up and specify a configuration file that controls your initial debugger configuration
- ☐ issue the **exec** command from your PROFILE to execute a CLIST or an EXEC when the debugger is invoked

- use the **exececho** keyword with the **auto** command in order to echo lines from an EXEC or CLIST
- use the **dbinput** command in order to input data from the terminal to an EXEC or CLIST
- use the **dblog** command in order to output information from an EXEC or CLIST to the Log window or session log.

Setting Up a Debugger PROFILE

The debugger PROFILE is a debugger feature that enables you to create an EXEC or a CLIST that contains a set of debugger commands. This PROFILE executes at initial entry to the program that is being run under the debugger. If you usually issue a certain set of commands when you first begin to debug a program, then using a debugger PROFILE saves time.

Note: You cannot use a PROFILE with debugger sessions in OS/390 batch. △

A PROFILE can contain any of the debugger commands in the following list (in addition to standard CLIST or EXEC commands that you need):

abort	**install**
auto	**on**
break	**query**
config file	**set**
define	**system**
disable	**trace**
drop	**undef**
enable	**window off**
exec	% (TSO only)
help	user-installed commands
ignore	

Note: The **config file** command is used to specify your configuration file as described in “Setting Up a Configuration File” on page 46. Additional details about using the **exec** command in a PROFILE are provided in “Executing EXECs or CLISTs from the Debugger” on page 49. △

When you start your program under the debugger (by specifying **'=d'**), your PROFILE executes automatically. Next, the debugger passes control to the terminal on entry to **main** (or, if the **indep** option is used, to the first function that is called in your program) and sends the **Cdebug**: prompt.

When your PROFILE executes, no function is active (including **main**). For this reason, commands such as **list** and **continue**, which require a function to be active, do not work in a PROFILE. Similarly, the **assign**, **copy**, and **dump** commands, which require a variable or object to be visible, do not work because none have been defined on entry to the debugger. If the PROFILE contains a user-installed command or an **exec** (or `%`) command, the same commands can be used in the called EXEC or CLIST as in the PROFILE itself.

Setting Up a PROFILE under CMS

Write an EXEC named PROFILE CDEBUG that contains all the debugger commands to issue when the debugger starts to run. You pass the filename of the program as the only parameter.

PROFILE CDEBUG can be written in REXX or EXEC2. Example Code 3.1 on page 43 illustrates a debugger PROFILE CDEBUG under CMS.

Example Code 3.1 Debugger PROFILE CDEBUG (CMS)

```
/*  Sample DEBUGGER PROFILE CDEBUG   */
/*                                   */
config file myconfig ❶
arg filename
say '****** Start of Debugger PROFILE' ❷
say 'Running file' filename
   if filename = 'WORDCOUN' then do  ❸
      'exec wordcoun'
   else do ❹
      'on main return break'
      'exec gencmds'
   end
'break main *' ❺
'query' ❻
say '******' rc 'actions active' ❼
say '****** End of Debugger PROFILE'
exit
```

The numbers in the previous example correspond to the following items:

1 The configuration file named MYCONFIG is specified.

2 Two SAY statements output a banner and the source filename.

3 The EXEC checks that the filename is WORDCOUN. If it is, the debugger command **`exec wordcoun`** is executed. WORDCOUN is the name of an EXEC that contains CMS commands.

4 If the filename is not WORDCOUN, the debugger performs an **`on`** command to break on return to **`main`**; then, the debugger performs another **`exec`** command by using the CMS EXEC GENCMDS.

5 A breakpoint is requested at every hook in **`main`**.

6 The **`query`** command is issued.

7 The REXX **`rc`** variable displays the number of debugger actions in effect as returned by **`query`**.

Setting Up a PROFILE under TSO

To set up a PROFILE under TSO, do the following:

1 Write a CLIST or REXX EXEC that contains the debugger commands in the order that you want to issue them. If you are using a REXX EXEC, the following address command must be issued from the EXEC before any debugger commands are issued:

```
address 'CDEBUG'
```

2 Put the PROFILE in a sequential data set that is named

```
first_level_qualifier.CDEBUG.CLIST
```

If your TSO profile specifies NOPREFIX, then your userid is used as the *first_level_qualifier*; otherwise, if a PREFIX is specified, that prefix is used as the *first_level_qualifier.*

If you want to put your PROFILE in another data set, you can also use the DDname DBGPROF to specify a PROFILE. If DBGPROF is allocated to the CLIST or EXEC, the debugger uses DBGPROF to find the PROFILE.

The PROFILE is any valid CLIST data set (as described by IBM publication *OS/390 TSO/E CLISTs*, SC28-1973-02). Not the following two general restrictions for a CLIST that is used in your PROFILE:

- □ You cannot use the ATTN CLIST command in a PROFILE.
- □ You cannot call a program that uses the debugger from a CLIST and share a GLOBAL with the PROFILE or with any other CLIST that is called by the debugger via **exec**.

The first line of the CLIST should be the following:

```
PROC 1 pgmname
```

where you subsitute the program name for *pgmname*. This variable is determined by the following guidelines:

- □ If you are running your program by using a TSO command processor, the command name is the program name. For example, in the following cases, **myprog** is the command name and, therefore, it is also the program name:

  ```
  alloc f(cplib) da(library.name) shr
  myprog =d
  ```

  or

  ```
  alloc f(cplib) da(library.name) shr
  c myprog =d
  ```

  Note that the command name (from the command line for a program), called as a command processor, takes precedence over the compilation section name.
- □ If you are executing your program by using the TSO CALL command and if the **main** function has been compiled with the **sname** option, the section name is the program name, as in the following:

  ```
  call library.name(myprog) '=d'
  ```

  If no section name is defined for **main**, but you have a declaration in your program for **_pgmnm**, that is the program name.
- □ If none of these situations fit-that is, you are not executing your program with a command processor, no section name is assigned to **main**, and **_pgmnm** is not defined-then the program name is UNKNOWN. See the *SAS/C Compiler and Library User's Guide, Third Edition* for more information about how the **_pgmnm** variable can be used.

The following example is a debugger PROFILE for TSO.

Example Code 3.2 Debugger PROFILE CLIST (TSO)

```
PROC 1 PGMNAME
   /* Example DEBUGGER PROFILE CLIST (TSO) */
```

```
/*---------------------------------------------------------*/
CONTROL NOCAPS ❶
CONFIG FILE 'USERID.MY.CONFIGS(CONFIG1)' ❷
WRITE ****** START OF DEBUGGER PROFILE ****** ❸
WRITE running program &PGMNAME
IF &PGMNAME NE UNKNOWN THEN DO ❹
   IF &PGMNAME = WRDCNT THEN DO ❺
      %wrdcnt
      END
   ELSE IF &PGMNAME = LINCNT THEN DO ❻
      %lincnt
      END
   ELSE DO ❼
      on main return break
      %gencmds
      END
   break main * ' ❽
   query ❾
   WRITE ****** &LASTCC actions active ❿
   END
WRITE ****** END OF DEBUGGER PROFILE ******
```

The numbers in the previous example correspond to the following items:

1 Uppercase is turned off.

2 The configuration file that is named USERID.MY.CONFIG(CONFIG1) is specified.

3 Two WRITE statements output a banner and the program name.

4 The CLIST checks to determine whether the program name that is passed through PGMNAME is known. (If it is not, execution of the CLIST ends.)

5 If the program name that is passed through PGMNAME is **`wrdcnt`**, the debugger executes the **`wrdcnt`** CLIST.

6 If the program name is **`lincnt`**, the debugger executes the **`lincnt`** CLIST.

7 Otherwise, the debugger requests a breakpoint on return from **`main`**, and executes the **`gencmds`** CLIST.

8 A breakpoint is requested at every hook in **`main`**.

9 The **`query`** command is issued.

10 The number of breakpoints and actions in effect is output.

Using a PROFILE to Select Full-Screen or Line Mode

The debugger opens, by default, in full-screen mode. By specifying a **`window off`** command in the PROFILE, you can start the SAS/C Debugger in line mode. This is the only **`window`** subcommand that can be issued in the PROFILE; other **`window`** subcommands can be issued from a configuration file, as described in the next section.

Note: You can also alternate between full-screen mode and line mode as described in "Switching Between Full-Screen Mode and Line Mode" on page 31. △

Return Codes for Invalid Commands for the PROFILE

See "Setting Up a Debugger PROFILE" on page 42, for a list of commands that are valid for use in a debugger PROFILE. If you pass an invalid command to the debugger when you enter the debugger, you receive a return code of –3.

Note: Since the PROFILE is a special-purpose CLIST or EXEC, return codes for debugger commands that are valid in a PROFILE are the same as those that are discussed in “Return Codes” on page 50. △

Setting Up a Configuration File

The configuration file controls the initial configuration of the PF keys and windows when you invoke the debugger. It is processed after your PROFILE, and you can specify a user-defined configuration file in a number of ways as described in “Specifying a Configuration File” on page 46. If no configuration file is specified, the debugger uses a default initial configuration that is supplied with the SAS/C Debugger.

The commands that are valid in a configuration file and the methods of saving configurations are described in “Creating Configuration Files” on page 47.

Specifying a Configuration File

You can specify a user-defined configuration file in the following ways:

1 Issue a **`config file`** command in your PROFILE. See “Specifying a Configuration File from a PROFILE” on page 46.

2 Associate a configuration file with your program. See “Associating a Configuration File with a Program” on page 47.

3 Create a user-specific default configuration file. See “Creating Your Own Default Configuration File” on page 47.

If more than one user-defined configuration file exists, the order of precedence for determining which file to use is as listed above.

Specifying a Configuration File from a PROFILE

The format of the **`config`** command that is used in your PROFILE to specify a user-defined configuration file depends on your operating environment. See “config” on page 202 for a complete description of the **`config`** command.

Under CMS You use the following format of the **`config`** command to specify a configuration file in your PROFILE:

```
config file filename
```

See the code sample in “Setting Up a PROFILE under CMS” on page 43 for an example of this format. *filename* can be any valid CMS filename.

Under OS/390 You use the following format of the **`config`** command to specify a configuration file in your PROFILE:

```
config file filename (member)
```

See the code sample in “Setting Up a PROFILE under TSO” on page 43 for an example of the first form of this format. In the first form, *filename* can be any OS/390 data set name. The second form is used to specify the *member* name of a file that is stored in a partitioned data set named *userid*.CDEBUG.CONFIG.

Associating a Configuration File with a Program

You can create configuration files that are selected for processing by the debugger according to a name that is generated during compilation of your program. The method of creating and specifying these configuration files depends on your operating environment.

Under CMS Create a configuration file that contains the desired configuration information. The file must have a valid CMS filename, a filetype of DBCONFIG, and a filemode of *, as in the following example:

```
program-name DBCONFIG *
```

When you compile the program, specify *program-name* with the **sname** option. Refer to the SAS/C Compiler and Library User's Guide, Third Edition for information about using the **sname** compiler option to specify a program name under CMS.

Under OS/390 Create a partitioned data set that is named *userid*.CDEBUG.CONFIG. You can add members to this PDS that are selected by member name when you compile your program. For example, you can create a configuration file that is named *userid*.CDEBUG.CONFIG(*myconfig*), which is associated with your compilation by the **sname** option when you compile a program in *userid.source*.C(*mycode*). See the SAS/C Compiler and Library User's Guide, Fourth Edition for information about using the **sname** compiler option to specify a program name under OS/390.

Creating Your Own Default Configuration File

You can create a user-defined configuration file that sets your default initial configuration. This configuration is used only if you do not specify a configuration file in your PROFILE or as described in "Associating a Configuration File with a Program" on page 47.

User-specific configuration files are created in the same manner as described for program-specific configuration files. The only difference is the name of the file. You must name your user-specific configuration file UNKNOWN. Under CMS it is created with the following filename, filetype, and filemode:

```
UNKNOWN DBCONFIG *
```

Under OS/390, it is created as the following PDS member:

```
'first_level_qualifier.CDEBUG.CONFIG(UNKNOWN)'
```

If your TSO profile specifies NOPREFIX, then your userid is the *first_level_qualifier*, otherwise, if a PREFIX is specified, then that prefix is the *first_level_qualifier.*

Creating Configuration Files

A configuration file can be created in the following ways:

- Use a text editor to type commands into the file
- Modify your configuration with the Keys and Configuration windows and then save the configuration with the **config save** command.

The second method offers the easier way of creating a new configuration file.

Note: When you use the **config save** command to save a configuration, the existing configuration file is replaced by the new one. △

Valid Commands in the Configuration File

The configuration file may contain only **keys define** commands and a certain subset of **window** commands. Invalid commands or errors in your configuration file usually cause the debugger to display messages. Under CMS, errors in your configuration file cause the debugger to pause with a **MORE** prompt.

PF key customization Any of the PF keys may be customized by issuing appropriate **keys define** commands, as described in "Using PF Keys" on page 27. You can type these commands directly into a configuration file by using a text editor, or you can use the **config save** command after modifying PF key assignments with the Keys window. See "Saving Your Configuration" on page 48.

Window customization In the configuration file, **window** commands have several purposes, some of which are the following:

- to specify the size, location, presence of borders, and color of windows
- to specify characters that are used for window borders
- to determine which windows are present.

The following subcommands of the **window** command are valid in the configuration file:

autopop
automatically pops up (makes unobscured) any window of the type that is specified when the file is updated.

border
specifies the characters to be used to form the borders of windows with borders.

color
customizes the color, attributes, and intensity of the border and the different areas in the named window.

config
customizes the configuration (size, position, and presence of borders) of the named window. The subcommand does not open a window.

context
controls the amount of context information around the highlighted line in the Source window.

intercepts
specifies the status of the input and the output intercepts and the processing of intercepted I/O.

memory
specifies the amount of memory to be allocated for various window buffers.

open
causes the named window to be present in the initial configuration.

scroll
sets the scroll amount displayed in the Status window.

trace
controls the production of trace lines in the Log window.

Note: The **autopop**, **color**, **config**, **context**, **open**, and **trace** commands take a window name as a parameter. △

Saving Your Configuration

The effect of specifying **window** commands outside the configuration file is discussed in "Windowing Interface and Command Execution" on page 148. Certain commands have the effect of changing the configuration (window colors, window position and size, number of windows, scroll amount, and so on) of the debugger. Similarly, **`keys define`** commands that are issued during execution (see “Using PF Keys” on page 27) may also change the configuration. The debugger enables you to save this configuration by issuing the **`config file`** and **`config save`** commands. For example, the following command displays the name of the current configuration file in the Log window:

```
config file
```

Then, when you issue the following command, your configuration is saved to that file:

```
config save
```

You can save your configuration to a file other than the current configuration file by issuing the **`config save`** command with a FILENAME argument. For example, under CMS, the following command saves the configuration to a configuration file that is named **`config1`**:

```
config save config1
```

This command also changes the current configuration file to **`config1`**.

Both the **`config file`** and **`config save`** commands are described in Chapter 14, “Command Directory,” on page 187.

Executing EXECs or CLISTs from the Debugger

See “Setting Up a Debugger PROFILE” on page 42 for information about the use of the debugger PROFILE, a feature of the debugger in which a CLIST or an EXEC is passed automatically to the debugger for execution when the debugger is invoked.

The **`exec`** command enables you to pass any valid EXEC or CLIST to the debugger at any point during your debugging session.

Under TSO, you can use both CLISTs and REXX EXECs. In a REXX EXEC, you must issue the following command so that subcommands are directed to the debugger rather than to TSO:

```
address 'CDEBUG'
```

There is no restriction against mixing CLISTs and REXX EXECs. A program of either type can call one of the other type. While a REXX EXEC is active, attention interrupts are trapped by the EXEC and not by the debugger. See “exec (TSO)” on page 219 for additional information about executing CLISTs and EXECs under TSO.

Under CMS, the EXEC must be written in REXX or EXEC2 and have filetype CDEBUG. See “exec (TSO)” on page 219 for additional information about executing EXECs under CMS.

Command Considerations

For both TSO and CMS, no restrictions exist about which commands can be used in an EXEC or CLIST. However, there are consequences to the use of certain commands. In addition, there are some rules about **`exec`**, used alone or used as part of an **`on`** command.

Commands That Continue Execution

If a CLIST or EXEC contains a command that requires the debugger to continue execution (**`abort`**, **`continue`**, **`exit`**, **`go`**, **`goto`**, **`resume`**, **`runto`**, and **`step`**), the command is executed and the EXEC or CLIST is ended. Other commands in the CLIST/EXEC that follow the command are not executed.

Echoing EXEC or CLIST Lines

The **`exececho`** keyword can be used with the **`auto`** command to echo each line of a CLIST or EXEC before the line is parsed and executed by the debugger. In a full-screen session the line is echoed to the Log window, and in line mode it is echoed to the session log.

The default setting for this command is **`noexececho`**. You can verify the setting with the **`query`** command.

The auto keyword of the **`transfer`** command also supports **`exececho`**. See "transfer" on page 271 for additional information.

exec Command

You cannot use another debugger command on the same line following **`exec`** or `%` commands. (The `%` command is used only under OS/390.)

dbinput Command

The **`dbinput`** command can be called from a CLIST or EXEC in order to input information from the terminal. In a full-screen session, input is made with the Termin window, and in a line-mode session it is made from the line prompt.

dblog Command

The **`dblog`** command can be called from a CLIST or EXEC in order to output information to the terminal. In a full-screen session output is sent to the Log window, and in line mode it is displayed in the session log.

Return Codes

You can design a CLIST or EXEC to test the value of a global variable that contains the previous condition code that is passed back from the debugger. Table 3.1 on page 50 lists global variables that can be checked for condition code values.

Table 3.1 Global Variables That Contain Condition Code Values

Command Language	Global Variable
CLIST	&LASTCC
EXEC	rc
EXEC2	&RC

SAS/C Debugger commands set the return codes that are listed in Table 3.2 on page 51. The column labeled "Successful" shows the return code passed back to a CLIST if the debugger command is executed. The column labeled "Unsuccessful" shows the code that is passed if the debugger command is not executed.

Table 3.2 Return Codes Set by Debugger Commands

Command	**Successful**	**Unsuccessful**
abort	not applicable	not applicable
assign	**0**	**1**
attn	**0**	**1**
auto	**0**	**1**
break	number of the action from the query list	**0***
catch	**0**	**1**
config	**0**	**1**
continue	not applicable	not applicable
copy	**0**	**1**
dbinput	**0**	**1**
dblog	**0**	**1**
define	**0**	**1**
disable	a nonzero number	**0**
drop	a nonzero number	**0**
dump	**0**	**1**
enable	a nonzero number	**0**
escape	none	none
exec (TSO)	code set by the CLIST called	code from **exec** command
exec (CMS)	code returned from the EXEC	**-3**
exit	not applicable	not applicable
go	not applicable	not applicable
goto	not applicable	not applicable
help	**0**	code from system help
ignore	request number from query	**0***
install	**0**	**1**
keys	**0**	**1**
list	number of last line listed	**0**
monitor	**0**	**1**
on	request number	**0**
print	**0**	**1**
query	number of last request satisfying the arguments of the **query** command	**0**
resume	not applicable	not applicable
return	**0**	**1**
runto	not applicable	not applicable

Command	Successful	Unsuccessful
scope	**0**	**1**
set	**0**	**1**
step	not applicable	not applicable
storage	**0**	**1**
system	**0**	return code from system
trace	number of action from query list	**0***
transfer	**0**	**1**
undef	**0**	**1**
watch	**0**	**1**
whatis	**0**	**1**
where	**0**	**1**
window	**0**	**1**
%	**0**	code from CLIST
?	not applicable	not applicable

In Table 3.2 on page 51 * indicates the request number is returned, even if there is unrecognized text after a valid command.

Example REXX EXEC Application

The following source listings demonstrate how to use a REXX EXEC when a program is running under control of the SAS/C Debugger.

The first listing, “BTREE” on page 52, is a program that generates a binary tree. When it is run under the control of the debugger, it opens up a Termin window in which you can type several data lines. The program inserts the lines in a binary tree and then performs an in-order traversal, printing the data lines in sorted order.

Note: Since the input is **stdin**, you could type some lines in a file and redirect **stdin** to the file. However, typing the lines from the Termin window is simpler. △

The second listing, “DUMPTREE” on page 54, is a REXX EXEC that can be used to display the nodes of the binary tree that is created by BTREE.

BTREE

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

typedef struct TREENODE {
   size_t length;
   char *value;
   struct TREENODE *left, *right;
   } TreeNode;

static TreeNode *alloc_TreeNode(size_t, const char *);
static void insert_TreeNode(TreeNode *, size_t, const char *);
static void print_Tree(TreeNode *);
```

```
void main(void)
   {
   TreeNode *tree = NULL;
   char io_buffer[258];

   fgets(io_buffer, 256, stdin);
   while (!feof(stdin) && !ferror(stdin)) {
      size_t length = strlen(io_buffer) - 1;
      if (io_buffer[length] != '\ n') {
         printf("String \ "%.40s \ " is too long.\ n", io_buffer);
         exit(8);
         }
      if (tree == NULL)
         tree = alloc_TreeNode(length, io_buffer);
      else
         insert_TreeNode(tree, length, io_buffer);
      fgets(io_buffer, 256, stdin);
      }

   if (ferror(stdin)) {
      puts("Error reading input file.");
      exit(8);
      }

   print_Tree(tree);
   exit(0);
   }

static void insert_TreeNode(TreeNode *root, size_t length,
                            const char *string)

   {
   int cmp;

   cmp = memcmp(string, root->value, min(length, root->length));
   if (cmp == 0)
      cmp = length - root->length;

   if (cmp > 0) {
      if (root->left != NULL)
         insert_TreeNode(root->left, length, string);
      else
         root->left = alloc_TreeNode(length, string);
      }
   else if (cmp < 0) {
      if (root->right != NULL)
         insert_TreeNode(root->right, length, string);
      else
         root->right = alloc_TreeNode(length, string);
      }

   else
      return;
   }

static void print_Tree(TreeNode *root)
   {
   if (root->right != NULL)
      print_Tree(root->right);

   printf("%.*s\ n", root->length, root->value);

   if (root->left != NULL)
      print_Tree(root->left);

   }
static TreeNode *alloc_TreeNode(size_t length, const char *string)
   {
   TreeNode *new;
   char *val;

   new = malloc(sizeof(TreeNode));

   if (new == NULL || (val = malloc(length)) == NULL) {
      puts("Can't allocate a new node");
      exit(8);
      }

   memcpy(val, string, length);
   new->length = length;
   new->value = val;
   new->left = new->right = NULL;

   return new;
   }
```

DUMPTREE

```
'/* REXX */'
trace ?r /* Use "trace o to turn off tracing */
address 'CDEBUG'
parse arg root

if root = '' then do
   'DBLOG DUMPTREE Error: root name not specified.'
   exit 4
   end

signal on error

'TRANSFER ROOT_TYPE TYPEOF' root
'TRANSFER ROOT_PTR VALUE' root
call dump root_ptr, root_type
exit 0

error:
'DBLOG DUMPTREE Error: debugger command at line' sigl 'failed.'
exit 4
```

```
dump: procedure
parse arg root_ptr, root_type

'TRANSFER RIGHT_PTR VALUE ((’root_type’)’root_ptr’)->right’
if right_ptr ¬ = ’0p00000000’ then
   call dump right_ptr, root_type

'TRANSFER LENGTH VALUE ((’root_type’)’root_ptr’)->length’
'TRANSFER STRING STR ((’root_type’)’root_ptr’)->value,’ length
'DBLOG DUMPTREE:’ string

'TRANSFER LEFT_PTR VALUE ((’root_type’)’root_ptr’)->left’
if left_ptr ¬ = ’0p00000000’ then
   call dump left_ptr, root_type
return
```

Running the Example

You can run this example REXX EXEC application under either CMS or TSO by performing the following steps:

1. Invoke BTREE under control of the debugger.
2. Use the **break** command to set a breakpoint at the entry to the **print_Tree** function.
3. Issue a **go** command in order to start program execution.
4. Type several data lines in the Termin window.
5. Type a **Y** in the **EOF:** field of the Termin window in order to signal the end of the file.
6. When execution stops at the **print_Tree** breakpoint, use the **exec** or **%** command to execute the DUMPTREE EXEC.
7. Single-step through the EXEC, examining the nodes of your binary tree.
8. When you finish examining your binary tree, type EXIT to return to the debugger.

Concepts Demonstrated

The example REXX EXEC application demonstrates how to use a recursive debugging EXEC in order to debug a recursive program. It shows that you can use an EXEC to create specialized debugger commands. In addition, it provides some good examples of the **transfer** command.

CHAPTER

4

Compiler Options

Introduction

Table 4.1 on page 57 summarizes compiler options that you use when you run programs under the SAS/C Debugger. All of the listed options are implemented in the OS/390 batch, TSO, and CMS environments.

Table 4.1 Compiler Options Useful for Debugging

Option	Default	Negation[1]
debug	**nodebug**	**!**
dbgmacro	**nodbgmacro**	**!**
dbhook	**nodbhook**	**!**
japan	**nojapan**	**!**
sname	**see description**	**+**

1 **!** means that the option can be negated. To negate an option, precede it with **no**. **+** means that the option cannot be negated.

debug Option

To use the full functionality of the debugger with your program, you must specify a compiler option that generates hooks in the object code. Hooks are used in order to transfer control to the debugger. Normally, you generate hooks by specifying the **debug** option at compilation. You can use the **dbhook** option to generate hooks when you are compiling with the **optimize** option. Without these hooks, there are only three times in which the debugger can gain control in a function that is compiled without **debug**:

- □ when a signal is raised
- □ at calls to or returns from the function
- □ when the function calls another function or returns from another function.

Library functions are not compiled with **debug**. Calls to some library functions cannot be trapped.

When you use the **debug** option, a debugger symbol table file is produced during compilation. See "Invoking the Debugger under TSO" on page 62 for information about debugging under TSO, "Invoking the Debugger under OS/390 Batch" on page 66 for information about debugging under OS/390 batch, and "Invoking the Debugger under CMS" on page 70 for information about debugging under CMS.

debug under TSO and OS/390 batch

The following describes what happens at compile time and debug time when you use the **debug** option under TSO and OS/390 batch.

Compile time   In the compile step, the debugger symbol table file is output to a member of a partitioned data set with the DDname SYSDBLIB, where the member name is the same as the section name that is used by the compiler. See the discussion of **sname** in "sname Option" on page 60. The data set that is allocated to SYSDBLIB should have a record format U and a block size of 4,080 bytes.

Debug time   When you run a program with the debugger under TSO or OS/390 batch, allocate the debugger symbol table file to the DDname DBGLIB. If DBGLIB is not allocated, then you cannot access source code or variable names. If DBGLIB is not allocated when the debugger begins execution, the debugger prompts you for it. Then you type the partitioned data set name of the debugger symbol table file, using standard TSO naming conventions.

If a symbol file table does not exist for your compilation, you receive the following message:

```
LSCD224 no debug file - your-section-name name not compiled with debug (-d)
```

Note that use of the **debug** option suppresses code optimizations and increases the size of the object code for your program. Also, you cannot use the **optimize** option when you use **debug**. After you have debugged your program, recompile it (without specifying **debug**) for the following reasons:

- □ Hooks that are inserted in your program by the **debug** option take up space.
- □ Loss of register variables and the need for extra stores make your program slower.
- □ Suppression of optimizations makes the object code longer and slower.

debug under CMS

The symbol table file is output to a file with the same filename as the source file and a filetype of DB.

Note: If no file is available with filetype DB when the program executes under the debugger, you receive the following message:

```
LSCD224 no debug file -
your-section-name name not compiled with debug (-d)
```

△

dbhook Option

The **dbhook** option generates hooks in the object code that is generated by the compiler. When you compile a module with the **debug** option, the **dbhook** option is implied. **dbhook** can be used with the **optimize** option in order to enable debugging of optimized object code. The default is **nodbhook**.

When you are using the debugger with optimized object code that has been compiled with the **dbhook** option, the source code is not displayed in the Source window and you cannot access variables. Therefore, the **print** command, and other commands that are normally used with variables, are not used when you are debugging optimized code. However, you can issue commands such as **step**, **goto**, and **runto** in order to control the execution of your program. Also, source code line numbers are displayed in the Source window, providing an indication of your location in the code. Also, you have the capability of viewing register values in the Register window.

The debugging of optimized code is most effective when it is used in conjunction with the Object Module Disassembler (OMD) or your system's debugger. The OMD is described in the SAS/C Compiler and Library User's Guide.

dbgmacro Option

This option causes the definitions of macros in the source file to be saved in the debugger symbol table. Only definitions of macros that contain the NO parameter are saved. You cannot take advantage of the debugger's handling of program macros unless you use this option.

japan Option

The following sections describe how the **japan** option affects both the symbol table that is produced at compile time and the debugger operation at run time.

Compile Time

If **japan** is used, the compiler stores all identifiers in lowercase characters in the symbol tables.

Debug Time

When you debug a module that has been compiled with the **japan** option, debugger input and output is affected as follows.

Input to the debugger   Commands can be issued in upper- or lowercase characters, as can command keywords such as **calls**, **entry**, **str**, and so on. When a section that is compiled with **japan** is debugged, the debugger converts all identifiers to lowercase automatically before looking them up in the symbol table. Types such as **struct**, **union**, **enum**, **unsigned**, **long**, **int**, **short**, **char**, and so on, also can be specified in uppercase.

Output from the debugger   Modules that have been compiled with the **japan** option can be displayed on either standard terminals or on KANJI terminals. KANJI terminals support only uppercase English alphabet characters. In order to display output on KANJI terminals, you must set the _UPPER environment variable to YES as described in “KANJI Terminal Support under CMS” on page 71 or “KANJI Terminal Support under TSO” on page 63.

sname Option

sname *name* defines the section name as *name*, where *name* can be up to seven characters in length. **sname** has system-dependent aspects, as described below. The compiler assigns the section name as follows:

- The section name is the name that you specify with the **sname** option.
- In the absence of a specific compile-time **sname** option, the section name is the name of the first external function in the module, truncated to seven characters.
- If no name is provided through the **sname** option and there is no external function in the module, the section name is the name of the first external variable in the function.
- If no name is provided through the **sname** option, if no external function is in the module, and if no external variable is in the module (that is, the module contains only static data or functions), then the section name is @ISOL@.

sname under TSO or OS/390 Batch

The specification is

```
sname (name)
```

where *name* defines the section name.

The debugger uses the section name in order to locate the member in DBGLIB that contains debugging information for that section. DBGLIB is discussed in Chapter 5, “Running the Debugger under TSO,” on page 61, and Chapter 6, “Running the Debugger under OS/390 Batch,” on page 65, along with the instructions for running the compiler under each operating environment.

sname under CMS

The specification is

```
sname name
```

CMS uses the source filename as the debugger filename, but the section name is distinct and independent of the source filename. The **sname** option enables you to use a nondefault SECTION-NAME argument in debugger commands.

CHAPTER
5

Running the Debugger under TSO

Data Sets Needed by the Debugger

When you run the debugger under TSO, you must allocate a data set for the debugger symbol table file. This partitioned data set, which is created when you compile with the **`debug`** option, contains a member for each compilation. The data set member contains debugging information for that compilation, including the name of the source for the compilation.

For this file, create a partitioned data set, unformatted (U), with a block size of 4080. This data set is then associated with the DDname SYSDBLIB at compile time and the DDname DBGLIB at run time. See "Quick Start to Using the Debugger under TSO" on page 7 for more details on these DDnames.

When you run the debugger, it looks for the source at the same location as at compile time (the compile-time data set). If you move your source between compile time and run time, you must indicate the new location. You can allocate the DDname DBGSRC in order to define a partitioned data set (PDS) as the location of the source file or files.

The debugger follows these steps in order to find the source file:

1 If the DDname DBGSRC is not defined, the debugger assumes that the source file is in the compile-time data set.

2 If DBGSRC is defined, the debugger checks whether the source file is allocated to a member of a partitioned data set (PDS). If it is, the debugger ignores the member name but uses the PDS as the location of the source file.

3 The debugger checks whether the source file was a PDS member at compile time. If it was, the debugger looks for a member with the **`same`** name. Otherwise, the debugger looks for a member with the **`sname`** name.

4 If the source file is not found by this search, the debugger looks for it in the compile-time data set.

The debugger also uses a data set that contains information that is displayed by the help. If the debugger is properly installed, you do not have to allocate this data set in order to to access the debugger help system.

In addition, the debugger uses a temporary data set in order to maintain symbol tables. Do not be concerned about this data set unless the debugger runs out of space in it. If this happens, you receive an abend (B37, D37, or E37) system completion code at run time. Then you must allocate more space to the file.

One way to allocate more disk space to the debugger work file in TSO or OS/390 batch is to allocate a temporary file of sufficient size to the DDname SYSTMPDB. If this DD statement is defined, the debugger uses this file as its work file rather than allocating its own. The normal space allocation for the debugger work file is 50 tracks, unless this was changed at your site when the SAS/C Debugger was installed. Contact your SAS Installation Representative for more information.

General Instructions

This section provides specific instructions for running the SAS/C Debugger under TSO. See"Quick Start to Using the Debugger under TSO" on page 7 for more general information.

Compiling Your Source Files

Compile each source file that you want to debug with the **debug** compiler option. Use the LC370 CLIST in order to compile your program. If you use the recommended naming conventions for your source data set and debugger data set (*userid.pdsname*.C and *userid.pdsname*.DBGLIB) and you do not fully qualify your source data set when you invoke the CLIST, the CLIST automatically associates your debugger file with the DDname SYSDBLIB. Otherwise, you are first prompted for the name of your debugger data set, and then the CLIST associates it with the DDname SYSDBLIB.

.

Linking Your Programs

Use the CLK370 CLISTin order to link your programs. For detailed instructions, see the SAS/C Compiler and Library User's Guide.

Linking C++ template programs requires the SYSDBLIB DD statement as well. The default option format for SYSDBLIB is **dbglib(ddn:sysdblib)**, and the default filename format is **ddn:sysdblib(sname)**.

Invoking the Debugger under TSO

Allocate data sets that are needed at run time, and call the debugger.

1 Issue a TSO ALLOCATE command for the debugger symbol file table. This is the same data set name that is defined as DDname SYSDBLIB in the compile step. When you invoke the debugger, this data set should have a DDname of DBGLIB. If you do not allocate DBGLIB before you invoke the debugger, the debugger prompts you for it. Note that if you compile with the **sname** option, the member name in DBGLIB is the section name that you specify. Otherwise, the member name is the default section name.

2 If you have moved your source in the time since you compiled the program, associate the DDname DBGSRC with the data set that contains the source.

3 Invoke the debugger by using one of the following routines:

   □ Issue the TSO CALL command and pass the parameter **=d** with the following command:

```
call dsname '=d'
```

The *dsname* names the load module and must follow standard TSO naming conventions.

If you need to examine portions of your program by using TSO TEST, invoke the SAS/C Debugger under TSO TEST. Here is a sample command that uses the same data set as the previous example:

```
test dsname '=d'
```

Note that you need to pass the option **=d** to TEST. Once the debugger has been invoked, you can escape to TSO TEST by pressing the attention key (PA1).

- If your site has installed the SAS/C TSO command processor support, you can call your program under TSO as a command processor with the following commands:

  ```
  program-name =d <arguments>
  ```

  or

  ```
  c program-name =d <arguments>
  ```

  You can also use TEST with a command processor by issuing the following command:

  ```
  test dsname cp
  ```

  Then TEST prompts you with **ENTER COMMAND FOR CP**, and you type

  ```
  program-name  =d <arguments>
  ```

  In these examples of calling a program as a command processor, *arguments* are run-time options, redirections, or values to be passed to the program.

  As previously described, once the debugger has been invoked, you can escape to TSO TEST by pressing the attention key (PA1).

4 In a line-mode session, you can save TSO output. The TSO Session Manager enables you to copy output from your screen (TSOOUT) to a sequential data set or a partitioned data set member using the TSO SMCOPY (Session Manager Copy) command. For example, to copy debugger output to a member OUT5 of a partitioned data set, type the following:

```
smc fs(tsoout) tds('userid.group.type (out5)') asis
```

where TSOOUT is the TSO output stream (that is, your terminal screen), and *userid.group.type* identifies a partitioned data set. OUT5 is the data set member.

For details about the SMCOPY command, see the IBM publication *OS/390 V2R9.0 SC28-1969 TSO Command Reference*.

KANJI Terminal Support under TSO

The SAS/C Debugger provides support for KANJI terminals or terminals that support only uppercase English alphabet characters. If you are using the debugger with a KANJI terminal, your program most likely was compiled with the **japan** option. The **japan** option is described in Chapter 4, “Compiler Options,” on page 57.

At debug time you must also set the _UPPER environment variable to YES or Y before debugging a module on a KANJI terminal. Under TSO this is accomplished with the **putenv** command, which is supplied with the SAS/C Library. Issue either of the following commands in order to set the _UPPER environment variable:

```
PUTENV _UPPER=YES
```

or

```
PUTENV _UPPER=YES PERM
```

The first **putenv** command sets the environment variable value for the remainder of the current session. The second one sets it permanently so that it is in effect for the remainder of the current session and for all future sessions.

It is possible that your site may not have installed the **putenv** command in a standard system library. For a full description of the **putenv** command and information about environment variables under TSO, see theSAS/C Compiler and Library User's Guide and the SAS/C Library Reference, Volume 1.

CHAPTER

6

Running the Debugger under OS/390 Batch

Data Sets Needed by the Debugger

When you run the debugger under OS/390 batch, you must allocate a data set for the debugger symbol table file. This partitioned data set, which is created as a result of compiling with the **`debug`** option, contains a member for each compilation. The data set member contains debugging information for that compilation, including the name of the source for the compilation.

For this file, create a partitioned data set, unformatted (U), with a block size of 4080. This data set is then associated with the DDname SYSDBLIB at compile time and the DDname DBGLIB at run time. See “Quick Start to Using the Debugger under TSO” on page 7 for more details on these DDnames.

When you run the debugger, it looks for the source at the same location as at compile time (the compile-time data set). If you move your source between compile time and run time, you must indicate the new location. You can allocate the DDname DBGSRC in order to define a partitioned data set (PDS) as the location of the source file or files.

The debugger goes through the following steps in order to find the source file:

1 If the DDname DBGSRC is not defined, the debugger assumes that the source file is in the compile-time data set.
2 If DBGSRC is defined, the debugger checks whether the source file is allocated to a member of a PDS. If it is, the debugger ignores the member name but uses the PDS as the location of the source file.
3 The debugger checks whether the source file was a PDS member at compile time. If it was, the debugger looks for a member with the same name. Otherwise, the debugger looks for a member with the **`sname`** name.
4 If the source file is not found by this search, the debugger looks for it in the compile-time data set.

The debugger also uses a data set that contains information that is displayed by the hypertext help system. If the debugger is properly installed, you do not have to allocate this data set in order to access the debugger help system.

In addition the debugger uses a temporary data set in order to maintain symbol tables. Do not be concerned about this data set unless the debugger runs out of space in it. If this happens, you receive an abend (B37, D37, or E37) at run time. Then you must allocate more space to the file.

One way to allocate more disk space to the debugger work file in TSO or OS/390 batch is to allocate a temporary file of sufficient size to the DDname SYSTMPDB. If this DD statement is defined, the debugger uses this file as its work file rather than allocating its own. The normal space allocation for the debugger work file is 50 tracks, unless this was changed at your site when the SAS/C Debugger was installed. Contact your SAS Software Representative for C Compiler products for more information.

General Instructions

This section provides specific instructions for running the SAS/C Debugger under OS/390 batch. See “Quick Start to Using the Debugger” on page 6 for more general information.

Compiling Your Source Files

Use one of the JCL cataloged procedures that are provided with the SAS/C Compiler, or write your own JCL to compile your program. If you use one of the cataloged procedures, override the SYSDBLIB card with one of your own that specifies your debugger data set name. If you write your own JCL, include a SYSDBLIB card for your debugger data set name.

If you compile with the **sname** option, the member name in the symbol table file is the section name that you specify. Otherwise, the member name is the default section name. See the SAS/C Compiler and Library User’s Guide for more information about the compiler.

Linking Your Programs

Use one of the JCL cataloged procedures that are provided with the SAS/C Compiler, or write your own JCL in order to link your program. See the SAS/C Compiler and Library User’s Guide

Invoking the Debugger under OS/390 Batch

Allocate the data sets that are needed at run time, and call the debugger. When you invoke the debugger, you need DD statements for DBGIN (the debugger input file), DBGLIB (the debugger symbol table file), and DBGLOG (the debugger output file). DBGLOG is provided automatically if you use one of the LC370 cataloged procedures. Also, if you moved your source between compile time and run time, then allocate DDname DBGSRC to the partitioned data set (or data set concatenation) that contains your source code library. Use JCL such as that shown in Example Code 6.1 on page 66 for running the SAS/C Debugger.

Example Code 6.1 Sample JCL for Running the Debugger under OS/390 Batch

```
//JOBNAME   JOB    jobcard information
//*-----------------------------------------------------------
//* RUN A PROGRAM, USING THE DEBUGGER
//*-----------------------------------------------------------
//          EXEC PGM=membermono ,PARM='=D'
//STEPLIB   DD DISP=SHR,DSN=your.load.library
//          DD DISP=SHR,DSN=your.site.LINKLIB
```

```
//SYSTERM  DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//DBGLOG   DD SYSOUT=*
//DBGSRC   DD DISP=SHR,DSN=your.source.library
//DBGLIB   DD DISP=SHR,DSN=your.debugger.library
//DBGIN    DD *
break main *
on main e print argc
go;go;go
print argc
exit
/*
```

In Example Code 6.1 on page 66, DBGSRC and DBGLIB are defined. Debugger commands are provided in the JCL. Also, you can put debugger commands in a data set, and you can allocate the data set to DBGIN, as in the following:

```
//DBGIN    DD DISP=SHR,DSN=your.debugger.commands
```

The debugger writes OS/390 batch output to DBGLOG.

Example Code 6.2 on page 67 shows JCL for compiling, linking, and running a program under the debugger. The example uses the LC370CLG procedure.

Example Code 6.2 Sample JCL for Compiling, Linking, and Running the Debugger by Using LC370CLG

```
//JOBNAME JOB jobcard information
//*--------------------------------------------------------
//* COMPILE, LINK, AND RUN A PROGRAM WITH DEBUGGER OPTION
//*--------------------------------------------------------
//STEP1       EXEC LC370CLG,PARM.C='DEBUG',PARM.GO='=D'
//C.SYSIN     DD DISP=SHR,DSN=your.source.library(member)
//GO.DBGIN    DD DISP=SHR,DSN=your.debugger.input
/*
```

CHAPTER 7

Running the Debugger under CMS

General Instructions

This chapter provides specific instructions for running the SAS/C Debugger under CMS. See also "Quick Start to Using the Debugger" on page 6 for more general information.

Compiling Your Source Files

Compile each program module that you want to debug with the **`debug`** compiler option.

Use the LC370 EXEC to compile your program under CMS. The LC370 EXEC is invoked with one of the following commands:

`LC370 filename <.filetype<.filemode>> <(options<)> >`
This format uses a CMS fileid to specify the input source file. You must specify *filename*; however, *filetype* and *filemode* are optional. The default value for the *filetype* argument is C. If you specify *filetype*, you must join it to *filename* with a period (.). If you do not specify *filemode*, all accessed disks are searched. If you use *filemode*, you must join it to *filetype* with a period.

`LC370 ddn:ddname <(member)> <(options<)> >`
With the ddn: format, the source file is specified by a DDname that has been defined with a CMS FILEDEF command. If *member* is used, it refers to a member of an OS/390 partitioned data set (PDS).

LC370 sf:**filename**<**filetype** <**dirname**> <(**options**<)> >
The **`sf:`** format specifies a Shared File System (SFS) fileid. Like the first format, *filename* is used to specify the source file, and *filetype* defaults to C. The SFS directory name is specified by *dirname*, which defaults to a period and selects your home directory. If you choose to specify a dirname, it can be associated with a filemode, in which case the **`sf:`** format functions exactly like the first format shown above. It is also possible to assign a logical name, NAMEDEF, to a dirname. NAMEDEFs can be used interchangeably with dirnames.

When you compile a program with **debug**, a debugger symbol table file is produced. This file outputs to a file with the same filename as the source file and a filetype of DB. The compiler writes its output files (LISTING, TEXT, and debugger symbol table) to different places depending on the command that you issue in order to invoke the LC370 EXEC.

- If a CMS file id or a DDname is used to specify the input source file, the compiler writes the output files to the input file's minidisk if possible. If you have read-only access to the input file's minidisk, the output files are written to your A disk.
- If an SFS file id is used to specify the input source file, the compiler writes the output files to the input file's directory, if possible. If you have read-only access to the input file's directory, the output files are written to your home directory.

  When using an SFS fileid, you must set the _DB environment variable before you invoke the debugger. The _DB environment variable contains a directory list, and the debugger searches it for the debugger table file. Instructions for setting the _DB variable are provided in "Invoking the Debugger under CMS" on page 70.

You can compile your program with the **sname** option as well as the **debug** option. The **sname** option is useful if you plan to specify the SECTION-NAME argument in debugger commands and do not want to determine the default section name. "debug Option" on page 57 and "sname Option" on page 60 describe the **debug** and **sname** options. See the SAS/C Compiler and Library User's Guide for additional information.

Linking Your Programs

If it is necessary, link your programs and run COOL by using the COOL EXEC. See the SAS/C Compiler and Library User's Guide for more information.

Invoking the Debugger under CMS

Invoke the debugger by using the following command:

```
program-name =d <parameters>
```

program-name is the filename of the MODULE that you want to debug, and *parameters* are any optional values, such as redirections, run-time options, or values that you want to pass to the program (for example, **argv**, **argc**[ ]).

If you are running the debugger with a VM release that supports the CMS Shared File System (VM/SP Release 6 and later), you can use an environment variable in order to specify a list of directories that you want to search for the debugger symbol table file. Specify the _DB environment variable with the GLOBALV command as follows:

```
GLOBALV SELECT LC370 SETL _DB directory-list
```

directory-list is the list of directories that you want to search. You may specify either a dirname or a NAMEDEF when listing a directory that you want to search. The debugger searches the directories that are specified by _DB in the order in which you list them, until it finds the debugger symbol table file. For example, the following command instructs the debugger to search the .C.PROJ1 directory first and then the .C.PROJ2 directory:

```
GLOBALV SELECT LC370 SETL _DB .C.PROJ1 .C.PROJ2
```

Keep in mind that the time and date stamps for the object file and the symbol table file must match; otherwise, the debugger does not accept the symbol table. Thus, old symbol tables for newly compiled source files (even if the source file has not changed) are not accepted by the debugger. If you recompile a source file that you plan to debug, you must recompile the file with **debug** in order to create a new symbol table file that corresponds to the new object file.

KANJI Terminal Support under CMS

The SAS/C Debugger supports KANJI terminals or terminals that support only uppercase English alphabet characters. When you are using the debugger with a KANJI terminal, your program most likely was compiled with the **japan** option. The **japan** option is described in Chapter 4, “Compiler Options,” on page 57.

At debug time you must also set the _UPPER environment variable to YES or Y before debugging a module on a KANJI terminal. Under CMS this is accomplished with one of the following GLOBALV commands:

```
GLOBALV SELECT CENV SET _UPPER YES
```

or

```
GLOBALV SELECT CENV SETP _UPPER YES
```

The first GLOBALV command sets the environment variable value for the remainder of the current session, and the second one sets it permanently so that it is in effect for the remainder of the current session and all future sessions.

Both commands set the environment variable _UPPER in group CENV to a value of YES. In the first GLOBALV command, _UPPER is a STORAGE class variable, and in the second command it is a LASTING class variable. See the SAS/C Library Reference, Volume 1 for more information about the **putenv** function as well as STORAGE and LASTING class variables.

Saving Line-Mode Output

In a line-mode session, you can save CMS debugger output (written to a line-mode terminal) by issuing CP commands as follows:

1 Before you invoke the debugger, issue the CP SPOOL command.

```
CP SPOOL CONSOLE START * CLASS A
```

2 Invoke the debugger as described in “Invoking the Debugger under CMS” on page 70.

3 When you have finished debugging your program, issue two more CP commands.

```
CP SPOOL CONSOLE STOP
CP SPOOL CONSOLE CLOSE
```

The debugger output is sent to your virtual reader. You can access it when you want it.

Note: You can use the debugger **system** command in order to issue CP commands that save your debugger session. △

CHAPTER
8

Using the Debugger from a Remote System

General Instructions

The remote debugger allows you to run the debugger display in one process and the program that is being debugged in another. The processes can be run on the same system or on different systems.

Under TSO and CMS, the remote debugger takes the form of a REXX EXEC that is named SASCDBG. Under the OS/390 UNIX System Services shell, it takes the form of an executable that is named SASCDBG.

The remote debugger is intended for use in the following situations:

- when you want to debug a Customer Information Control System (CICS) application
- when you want to debug a UNIX System Services application
- when you want to run the debugger and application on different terminals or systems
- when you need to debug a full-screen program
- when you need to debug a multitasking application.

We recommend using the local debugger for most other debugging tasks. For example, when debugging an application under TSO that is not a UNIX System Services application, the local debugger is considerably more efficient, since it is not subject to context-switching or communication delays.

How the Remote Debugger Works

The remote debugger is similar to the local debugger in its appearance and operation. The main difference is in the program architecture and start-up procedure.

When you are running the remote debugger under TSO or CMS, the remote debugger provides the same full-screen debugging capabilities as the local debugger. When you are running the remote debugger in the UNIX System Services shell or under OS/390 batch, the remote debugger is limited to line-mode operation only.

If you are familiar with the local debugger, you should have no problem using the remote debugger, once you understand the environment and start-up procedures that are described here.

Architecture

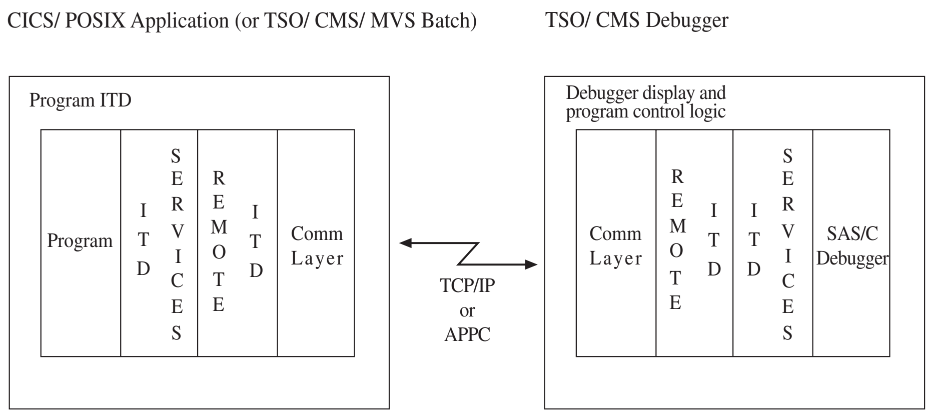
The SAS/C Debugger client/server architecture allows debugging of remote applications. There are two components in a remote debugging session:

- the debugger display and program control logic
- the program being debugged and an interface to the debugger (ITD).

The debugger display, along with the program control logic, act as the server. It provides the debugger services, under user control, to the client program. Each debugger component runs in a separate process. Depending on your operating environment, this may mean a different address space, task, virtual machine, or UNIX System Services process. In addition, the client program may be running on a physically different host.

As shown in Figure 8.1 on page 75, the debugger processes communicate with each other through a communications layer, which can use either the TCP/IP or APPC communications access method.

Figure 8.1 Remote Debugger Architecture



For maximum productivity, typically the debugger display component is run in a full-screen session under TSO or CMS. You can run the client program in any environment that supports the SAS/C library. CICS and UNIX System Services processes are the most typical environments. You can also run the client program in environments such as TSO, CMS, OS/390 APPC address spaces, or OS/390 batch.

Start-Up Methods

From the user's point of view, the most significant difference between the local debugger and remote debugger is the start-up procedure. To use the local debugger, you compile your program in debug mode and include the **`=debug`** run-time option when you run the executable. The remote debugger has two start-up methods:

- independent start-up
- automatic start-up.

With the independent start-up method, you start the debugger display process first, by using the SASCDBG debugger interface. When the process starts, the terminal displays the communications access method and other information that is needed by the client program in order to connect to the debugger display. For example, under TSO, you might see the following message after starting the remote debugger display:

```
SASCDBG DBCOMM(TCPIP)

SAS/C Remote Debugger can be reached via:
_DB_COMM=TCPIP connect to _DB_HOST=10.1.1.1
  (OS/390), at _DB_PORT=13227
```

Then, in a separate step, you start the program to be debugged in the normal way for your environment, and you specify the **`=debug`** run-time option on the command line. You can specify the connection information with environment variables before starting the program, or you can issue the equivalent command-line options. For example, you might call a TSO load module in the following manner:

```
CALL ABC.LOAD(MYPGM) '=D =_DB_COMM=TCPIP
  =_DB_HOST=OS/390 =_DB_PORT=13227'
```

With the automatic start-up method, you start both processes in a single step by including the program name in the SASCDBG command line. For example, in the UNIX System Services shell, you might start the remote debugger with the following command:

```
sascdbg -tcpip mypgm
```

For more information about environment variables, See “Debugger Environment Variables” on page 76. For a complete description of SASCDBG syntax, see “Using the SASCDBG Debugger Interface” on page 78.

CICS applications are a special case. Since CICS does not support a command line, you must use the remote debugger’s CICS front-end transaction in order to set the environment variables and to launch the program that is being debugged. “Debugging CICS Applications” on page 83 describes this procedure.

Other Start-Up Methods

For applications that have unique start-up requirements or where automatic start-up is unsupported (for example, OS/390 batch), you can supply your own *program invocation exit* for starting the program that is being debugged. A program invocation exit can be written in C or assembly language and must conform to the specifications that are described in “Using Remote Debugger User Exits” on page 86.

Debugger Environment Variables

The SAS/C Debugger inspects one or more environment variables in order to determine the debugger operating mode: local or remote. In a remote debugging session, both debugger processes inspect these variables in order to determine the communications access method and the information that is needed to in order to establish communication between the debugger processes. The following sections describe these environment variables and how to set them.

Environment Variable Descriptions

`_DB_COMM`=TCPIP | TCPIP_*xxx* | APPC | NONE | LOCAL
: determines the debugger operating mode—local or remote—and the communications access method for remote operation. Specifying TCPIP, TCPIP_*xxx*, or APPC selects remote operation and specifies whether to use TCP/IP or APPC as the communication method between the debugger processes. When TCPIP_*xxx* is specified, it selects the TCP/IP implementation that is specified by the **`setsockimp(’’xxx’’)`** function call. The default value is NONE, which starts a local debugging session. LOCAL and NONE are synonyms.

`_DB_HOST=`*ip_addr* | *hostname*
: specifies the dotted decimal IP address or host name of the machine where the remote debugger display process will be run or is running (TCP/IP only). During independent start-up, this value is displayed at the terminal and is used by the client program in order to to connect to the debugger display.

Note: The host name is translated to an IP address by a **`gethostbyname`** function call. △

`_DB_PORT=`*port_number*
: specifies the TCP/IP port number of the debugger display and control process (TCP/IP only). During independent start-up, this value is displayed at the terminal and is used by the client program in order to connect to the debugger display.

`_DB_LU=`*lu_name*
: specifies the name of the SNA Network Logical Unit (LU) where the debugger display process will run or is running (APPC only). The LU name consists of 1–8

nonblank uppercase letters or numbers (A–Z, 0–9). During independent start-up, the LU name is displayed at the terminal and is used by the client program in order to request an APPC connection to the debugger display. The default value is 8 blanks, which indicates the system base LU.

Note: The default value may or may not work, depending on the APPC definitions for your system. △

`_DB_TP=`*tp_name*
specifies the Transaction Program (TP) name of the debugger display process (APPC only). This name is registered with the APPC LU and consists of 1–64 uppercase or lowercase alphabetic characters, numbers, or special characters, except $, #, and @. During independent start-up, the TP name is displayed at the terminal and is used by the client program in order to request an APPC connection to the debugger display. The default value is SASCDBG.

`_DB_MODE=`*mode_table* | ISTINCLM
specifies the APPC logon mode table (APPC only). This table establishes the VTAM session parameters for the LU partners. This variable is specified only when you are debugging a CICS application or when you are running the client program on a physically different system. You can obtain the mode table specification from the VTAM APPL definition for the LU. You can also use the mode table ISTINCLM, which is a default system-supplied mode table for VTAM. You must specify the ISTINCLM table explicitly; **`_DB_MODE`** does not default to this value.

`_DB_TIMEOUT=`*n*
specifies the timeout value, in seconds, for interprocess communications. A process is considered nonresponsive if it fails to acknowledge a request within the specified period. When this occures, a message is displayed at the terminal.

Setting Environment Variables

You can set the debugger environment variables at three different levels, depending on the system:

- CMS and TSO support environment variables with permanent, external, and program scopes.
- CICS supports external and program scopes.
- OS/390 batch supports program scope only.

There is no equivalent to variable scopes in the UNIX System Services shell. However, UNIX System Services environment variables are similar to external scope variables, because they remain defined for the duration of the shell and can be inherited from the shell. For details about environment variable scopes, See the SAS/C Library Reference, Volume 1.

Under TSO, you can use the SAS/C PUTENV command in order to set an environment variable with the specified scope. Under CMS, you must use the CMS GLOBALV command. Under the UNIX System Services shell, use the **`export`** command.

For example, the following commands set the **`_DB_COMM`** environment variable to TCPIP. For TSO and CMS, the variable is defined with an external scope and remains defined for the life of the session.

TSO environment:

```
PUTENV _DB_COMM=TCPIP EXTERNAL
```

CMS environment:

```
GLOBALV SELECT CENVSETS _DB_COMM TCPIP
```

UNIX System Services Shell

```
export _DB_COMM=TCPIP
```

For details about the PUTENV command, see the SAS/C Compiler and Library User's Guide. For details about the GLOBALV command, see the IBM publication SC24-5776 VM/ESA V2R4.0 CMS Command Reference.

Optionally, you can set the environment variables on the command line during independent start-up of the program that is being debugged. This is equivalent to setting the environment variables with a program scope. For example, under CMS, the following command starts MYPGM in debug mode and sets the **`_DB_COMM`** environment variable to TCPIP:

```
MYPGM =d =_DB_COMM=TCPIP
```

Under OS/390 batch, you must set the environment variables in an EXEC PARM string. There is a 100-character limit on EXEC PARM strings. If this limit presents a problem, you can use another method for setting the environment variables. For example, you could use *argument redirection* in order to obtain the environment variables from a file. For details about argument redirection, see the SAS/C Compiler and Library User's Guide.

Because CICS does not enable you to specify program-scope variables on the command line, you should use the remote debugger's CICS front-end transaction in order to set the environment variables for the remote debugging session. See "Debugging CICS Applications" on page 83 for details.

Using the SASCDBG Debugger Interface

The SASCDBG debugger interface has three forms: TSO, CMS , and UNIX System Services shell. In addition, for TSO and UNIX System Services, there is a form for the independent start-up method and a form for the automatic start-up method. The start-up method depends on the presence or absence of a program name. CMS supports the independent start-up method only. The independent start-up method does not require the program name when you run the SASCDBG debugger interface.

Process Invocation

When you start the remote debugger under TSO or the UNIX System Services shell, you can specify the method that invokes the program that is being debugged. Your choices are

- FORK
- OEATTACH
- ATTACH

The FORK method is similar to a UNIX fork. It is the standard method in the UNIX System Services environment for creating a child process with its own address space. Typically, you will use the FORK method when debugging a UNIX System Services application under TSO.

The OEATTACH method provides an alternative for debugging UNIX System Services applications under TSO or the UNIX System Services shell. File handling, signal handling, and so forth, is like UNIX System Services but the address space is the same as the debugger. Typically, you will use the OEATTACH method for improved performance over FORK.

The ATTACH method starts the debugger processes by using the assembler ATTACH macro. You can use the ATTACH method for debugging applications under TSO that are not UNIX System Services applications. However, we do not recommend using this method, because the local debugger is considerably more efficient than the remote debugger in this environment.

Program I/O Handling

When debugging a UNIX System Services application under TSO by using the FORK or OEATTACH method, a UNIX System Services terminal (`tty`) is not defined for the program. Instead, the debugger intercepts program I/O to the standard UNIX System Services file descriptors 0, 1, and 2, and processes it according to the specifications in the Termin and Termout windows (if the debugger is running in full-screen mode). Any attempt by the program to open /dev/tty fails. If you invoke the program with OEATTACH, the program can open the file //ddn:* in order to access the TSO terminal.

You can disable program I/O intercepts with the SASCDBG DBTERM parameter. This causes the program output to the UNIX System Services file descriptors to be discarded. In addition, file 0 input requests receive an immediate end-of-file indicator. For more information about DBTERM, see "sascdbg Arguments" on page 80.

sascdbg Syntax

This section describes the syntax for the SASCDBG debugger interface. The syntax is different in each environment, but the arguments have the same meaning.

Note: You can abbreviate command keywords to four or fewer characters in most cases: PARMS and RESTART can be abbreviated to a single character; DBCOMM, DBPORT, and DBINVK can be abbreviated to three characters; DEBUG can be abbreviated to two characters; and DBTERM can be abbreviated to four characters. YES and NO values can be abbreviated to one character. All intermediate abbreviations are accepted. DBLU and DBTP cannot be abbreviated △

TSO forms:

SASCDBG *pgm_name*
  [PARMS(*pgm_args*...)]
    [DBCOMM(TCPIP | TCPIP_ *xxx*)]
    [DBPORT(*port_num*)]
    [RESTART(YES | NO)]
    [DBTERM(YES | NO)]
    [DBINVK(FORK | OEATTACH | ATTACH)]
    [DEBUG(YES | NO)]

SASCDBG
  [DBCOMM(TCPIP | TCPIP_ *xxx* | APPC)]
    [DBPORT(*port_num*)]
    [RESTART(YES | NO)]
    [DBLU(*lu_name*)]
    [DBTP(*tp_name*)]

CMS form:

SASCDBG (TCPIP | TCPIP_ *xxx*
  [PORT=*port_num*]
    [RESTART])

UNIX System Services Shell forms:

```
sascdbg
   [ -tcpip | -tcpip_ xxx ]
      [ -port=port_num ]
      [ -nod ]
      [ -fork | -oeattach ]
      pgm_name [ pgm_args... ]

sascdbg
   [ -tcpip | -tcpip_ xxx ]
      [ -port= port_num ]
```

sascdbg Arguments

No arguments except *pgm_name* and *pgm_args* are case sensitive.

FORK
: requests program invocation with a FORK function call and then an EXEC of the program to be debugged in the child process of the FORK. This is the default program invocation method when you debug a UNIX System Services application under TSO or the UNIX System Services shell.

OEATTACH
: requests program invocation with an OEATTACH function call for improved performance over the FORK method when you debug a UNIX System Services application under TSO or the UNIX System Services shell.

ATTACH
: requests program invocation with the OS/390 ATTACH macro. You can use this option to debug applications under TSO that are not UNIX System Services applications. However, because the local debugger does not have to communicate over a network to debug and is therefore more efficient, using the local debugger instead of the **ATTACH** option is recommended.

DEBUG
: determines whether the **=debug** run-time option is added during program invocation; the default is YES. Normally, the **=debug** option must be passed by SASCDBG in order to cause the client program to invoke the debugger interface. However, if the program uses the **_nlibopt** external variable in order to suppress run-time option handling, **=debug** is interpreted as a program argument. If your program sets **_nlibopt**, it must initialize the library external variable **_options** to specify **_DEBUG** in order to allow the program to be debugged. For more information about the **_nlibopt** and **_options** variables, see the SAS/C Compiler and Library User's Guide.

NOD
: suppresses insertion of the **=debug** run-time option during program invocation. **-nod** is the equivalent of DEBUG(NO) for the UNIX System Services.

DBTERM
: determines whether the debugger intercepts terminal I/O when you are debugging a UNIX System Services application under TSO by using the FORK or OEATTACH method; the default is YES.

Note: This option has no effect when you debug a program that is not a UNIX System Services program. Terminal I/O is controlled by the initial configuration file settings and the I/O window intercept settings of the debugger's terminal. △

TCPIP
TCPIP_*xxx*
APPC

determines the communication method between the debugger processes, either TCP/IP or APPC. For SAS/C Releases prior to Release 7.00, when TCPIP is specified, the debugger uses your site's default TCP/IP implementation of nonintegrated sockets. For SAS/C Release Release 7.00, when TCPIP is specified, the debugger uses your site's default TCP/IP implementation of integrated sockets. When TCPIP_*xxx* is specified, the debugger uses the TCP/IP implementation that is specified by the **setsockimp(''*xxx*'')** function call. For example, TCPIP_OE requests UNIX System Services integrated sockets. You can specify any TCP/IP implementation that is installed and available on your system. For more information, see the description of the **setsockimp** function in SAS/C Library Reference, Volume 2.

Specifying the communications access method on the command line sets the program scope **_DB_COMM** environment variable. If this variable is already defined with an external or permanent scope, the command line specification takes precedence. If you do not specify the communications access method, the **_DB_COMM** environment variable is used.

RESTART

determines whether the debugger display process is restarted after program termination or loss of communication with the client program (TSO and CMS only). The default is NO. Use this option with the independent start-up method in the event that you need to restart the debugger processes, for example, when you debug a pseudo-conversational CICS transaction. See "Debugging CICS Applications" on page 83 for details. Also see the *port_num* option.

If you specify the RESTART option, you must use attention (PA1) on TSO or the HX command on CMS in order to terminate SASCDBG. Otherwise, it continues to try to reconnect with the client program.

Note: This option does not preserve breakpoints or other debugger session parameters. It recreates the debugger's entire C environment and reloads the debugger load modules. △

port_num

specifies the TCP/IP port number that is used by the remote debugger (TCP/IP communications access method only). If the debugger cannot use the specified port, for example, because it is in use by another process, it displays a warning message and allows the system to assign a port number. Port numbers can be in the range of 0–65535. If *port_num* is 0, the system assigns a port number from 1 to 65535.

If the operating environment supports external scope environment variables, the debugger saves the port number from the current session in an external scope **_DB_PORT** variable. (See the note below for information about environments that do not support external scope variables.) By default, the debugger tries to reuse the port in the **_DB_PORT** variable if you do not set *port_num* explicitly. This behavior is especially useful when you specify the RESTART option, for example, when debugging a pseudo-conversational CICS transaction. This allows the client program to automatically reestablish communication with the debugger on subsequent invocations.

The system always assigns the port number if this is the first debugging session since logging in, if the **_DB_PORT** environment variable is cleared, or if the debugger is unable to reuse the previous port number.

Note: Environments such as OS/390 batch and the UNIX System Services shell do not support external class environment variables. In these environments, the debugger uses the specified *port_num* for the current debugging session, but does not retain it for future sessions. △

lu_name
: specifies the name of the APPC Logical Unit (LU) where the debugger display is run (APPC communications access method only). The LU name consists of 1–8 nonblank uppercase letters or numbers (A–Z, 0–9). During independent start-up, the LU name is displayed at the terminal and is used by the client program in order to request an APPC connection to the debugger display. The default value is 8 blanks, which indicates the system base LU.

  Note: The default value may or may not work, depending on the APPC definitions for your system. △

tp_name
: specifies the name of the APPC Transaction Program (TP) that is registered with the APPC LU (APPC communications access method only). The TP name consists of 1–64 upper- or lowercase alphabetic characters, numbers, or special characters, except $, #, and @. During independent start-up, the TP name is displayed at the terminal and is used by the client program in order to request an APPC connection to the debugger display. The default value is SASCDBG.

pgm_name
: specifies the name of the program that is debugged. This program automatically connects with the remote debugger display. Under TSO, specify the program name as the first argument. Under the UNIX System Services shell, specify the program name last. For details about how the debugger locates programs in the UNIX System Services hierarchical file system (HFS), see “Pathname Resolution under UNIX System Services” on page 82.

pgm_args
: specifies one or more run-time arguments that are passed to the program that is debugged. The remote debugger adds the **`=debug`** run-time option automatically unless you specify **`-nod`** (UNIX System Services) or **`DEBUG(NO)`** (TSO/CMS).

Pathname Resolution under UNIX System Services

When you are debugging a UNIX System Services program, *pgm_name* takes the form of a pathname in the UNIX System Services hierarchical file system. If *pgm_name* does not begin with a / (slash), the debugger searches for the program in the following way:

1. If you run the debugger display in the UNIX System Services shell, it first checks your current working directory.
2. Otherwise, it checks your home directory.
3. Finally, it checks the directories on your search path, as defined in the **`PATH`** environment variable.

If *pgm_name* begins with a / (an absolute pathname), the debugger looks for the program at the specified pathname.

Restrictions

The remote debugger has the following restrictions on its use and operation:

- When debugging a UNIX System Services program under TSO, you must use the TCP/IP communications access method if you launch the program with ATTACH or OEATTACH.
- Under CMS and UNIX System Services, the debugger display component supports only the TCP/IP communications access method.

- When using the ATTACH method of program invocation, the debugger environment variables are appended to the program run-time arguments in the form of **`=name=value`**. This may cause problems for programs that are compiled with the **`_nlibopts`** variable, because the environment variables are interpreted as program arguments and are not set as environment variables.

Debugging CICS Applications

CICS does not support a command line. When you start a CICS program, you specify only the program's transaction name; you cannot specify run-time options, environment variables, or command parameters. Since the remote debugger uses environment variables to exchange information with the client program, you must use the remote debugger's CICS front-end transaction to set these variables and launch the program that is debugged.

To use the remote debugger with a CICS application, follow these steps:

1 Start the remote debugger on the system where you do your development, for example:

```
SASCDBG DBCOMM(TCPIP)

SAS/C Remote Debugger can be reached via:
_DB_COMM=TCPIP connect to _DB_HOST=10.1.1.1
  (OS/390), at _DB_PORT=13227
```

2 On your CICS system, start the remote debugger's front-end transaction. The transaction name, as distributed with the SAS/C product, is **`DBUG`**. Its format is

```
DBUG trans_name
```

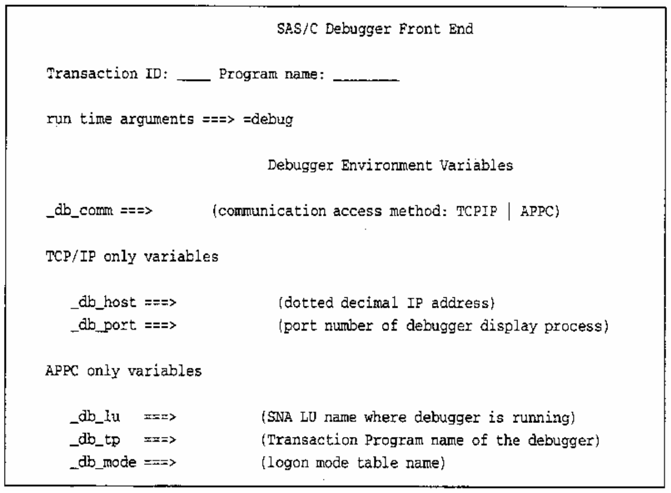
Where *trans_name* is the 4-character transaction name that is associated with the program to be debugged. For example:

```
DBUG ctim
```

Note: Your CICS systems administrator can change the name of the *DBUG* transaction, if desired. Check with your administrator if you have any questions about the name of the transaction at your site. △

Figure 8.2 on page 84 shows the screen that is displayed by the **`DBUG`** transaction. You can tab from field to field and specify values for the debugger environment variables. You can also specify run-time options; the **`=debug`** option is specified by default.

Figure 8.2 DBUG Transaction Screen


```
                         SAS/C Debugger Front End

Transaction ID: ____ Program name: ________

run time arguments ===> =debug

                         Debugger Environment Variables

_db_comm ===>       (communication access method: TCPIP | APPC)

TCP/IP only variables

   _db_host ===>          (dotted decimal IP address)
   _db_port ===>          (port number of debugger display process)

APPC only variables

   _db_lu   ===>        (SNA LU name where debugger is running)
   _db_tp   ===>        (Transaction Program name of the debugger)
   _db_mode ===>        (logon mode table name)
```

Once you type the appropriate values, press the ENTER key to in order start your application. The **DBUG** transaction verifies that the named transaction exists, that the specified program name is associated with the transaction, and that the program can be loaded into memory. If any of these checks fail, the **DBUG** transaction displays an error message and exits. Otherwise, it starts the application, and it produces the remote debugger display with your application's source code in the Source window.

If you specify the wrong value for any debugger environment variable (for example, the wrong TCP/IP port number), the application exits with an ABEND 1219 error message. You can rerun the **DBUG** transaction and correct the value without restarting the debugger. The **DBUG** transaction saves the latest value of each field in an external scope environment variable. You see this value when the transaction screen reappears. Replace the value with a new one and press ENTER when you are finished.

If you debug a pseudo-conversational CICS application, you start the remote debugger with the RESTART parameter under TSO or CMS. The RESTART parameter causes the debugger to restart with the same connection parameters. If you use the TCP/IP communications access method, do not set the DBPORT parameter to 0, since this forces the system to select a new port number when the debugger is restarted. If this occurs, the CICS application terminates with an ABEND 1219 error message. Otherwise, your pseudo-conversational program reestablishes communication with the debugger automatically, and you can continue to debug your program without interruption.

If the debugger cannot reuse the previous TCP/IP port, for example, because it is in use by another process, it displays its normal connection message, and the CICS application terminates with an ABEND 1219 error message.

Note: CICS applications cannot communicate with the remote debugger over TCP/IP until you start TCP/IP for your CICS region. For IBM TCP/IP, if the start-up transaction has not been executed, your CICS application terminates with an ABEND AEY9 error message. The CICS administrator can start IBM TCP/IP with the IBM CSKE transaction. Other TCP/IP vendors may have different start-up requirements and procedures. △

Start-Up Scenarios

The following examples show how you can start the remote debugger in different environments.

TSO Independent Start-Up (TCP/IP)

TSO session 1:

```
SASCDBG DBCOMM(TCPIP)
```

TSO session 2:

```
CALL XYZ.LOAD(MYPGM) '=D =_DB_COMM=TCPIP
  =_DB_HOST=OS/390 =_DB_PORT=1234'
```

Start the remote debugger by using the TCP/IP communications access method. Then, by using the information that is displayed at debugger start-up, call the program load module MYPGM in debug mode. On the command line, specify the environment variables for the communications access method.

TSO Independent Start-Up (APPC)

TSO session 1:

```
SASCDBG DBCOMM(APPC) DBLU(C02SESS) DBTP(SASCDBG)
```

TSO session 2:

```
CALL XYZ.LOAD(MYPGM) '=D =_DB_COMM=APPC
  =_DB_LU=C02SESS =_DB_TP=SASCDBG'
```

Start the remote debugger by using the APPC communications access method. On the command line, specify the Logical Unit name and Transaction Program name (this is required if the corresponding environment variables are undefined). Then call the program load module MYPGM in debug mode. On the command line, specify the APPC communications access method environment variables.

Note: The **_DB_TP** environment variable defaults to SASCDBG and is shown in this example for illustration only. △

CMS Independent Start-Up

CMS session 1:

```
SASCDBG (TCPIP
```

CMS session 2:

```
MYPGM =D =DB_COMM=TCPIP =_DB_HOST=VM =_DB_PORT=1234
```

Start the remote debugger by using the TCP/IP communications access method. Then, by using the information displayed at debugger start-up, run MYPGM in debug mode. On the command line, specify the environment variables for the communications access method.

OS/390 Batch Independent Start-Up

TSO session:

```
SASCDBG DBCOMM(TCPIP)
```

OS/390 batch JCL:

```
//STEP1 EXEC PGM=MYPGM,
//  PARM='=D =_DB_COMM=TCPIP =_DB_HOST=OS/390=_DB_PORT=1234'
//STEPLIB  DD DSN=pgm.load,DISP=SHR
//CTRANS   DD DSN=SASC.LOAD,DISP=SHR
//SYSPRINT DD SYSOUT=A
```

Start the remote debugger by using the TCP/IP communications access method. Then submit a batch job that runs MYPGM in debug mode. In a PARM string, specify the values of the environment variables for the communications access method.

TSO automatic start-up

```
SASCDBG mypgm DBCOMM(TCPIP) DBINVK(FORK)
```

Start the remote debugger by using the TCP/IP communications access method. Debug the UNIX System Services program MYPGM by using the FORK method.

Note: The program name is case sensitive and is the first argument in the command line. △

UNIX System Services Shell Automatic Start-Up

```
sascdbg -tcpip -fork /devel/r6/mypgm
```

Start the remote debugger by using the TCP/IP communications access method. Debug the program /dev/r6/mypgm by using the FORK method.

Note: The FORK method is the default for invoking a program under the UNIX System Services shell and is included in this example for illustration only. △

Using Remote Debugger User Exits

The remote debugger supports the use of program invocation exits for starting applications with unique start-up requirements or where automatic start-up is unsupported, such as CMS, CICS, or OS/390 batch. A program invocation exit can be written in C or assembly language. It can perform any processing that is necessary in order to start the program that is being debugged. For example, you might use a program invocation exit to

- build and submit JCL
- initiate a CICS transaction
- AUTOLOG a CMS session.

The program invocation exit can use the full SAS/C library and any other operating environment services that are available to normal (unauthorized) SAS/C environments.

Calling Sequence

The remote debugger calls the program invocation exit just before it would normally wait for a TCP/IP or APPC connection request from the program to be debugged. When the exit finishes its processing and returns, the debugger waits for the connection request from the remote program until it continues to initialize.

The program invocation exit is called once per remote debugger session. It is not called at all when the debugger is running in local mode (for example, when the **_DB_COMM** environment variable equals LOCAL or NONE).

The remote debugger supports the input and output exits L$UDBIN and L$UDBOUT that are described in Appendix 3, "Debugger I/O Exit Routines," on page 301. The program invocation exit is independent of the I/O exits. The L$UDBOUT output exit is called for initialization before the program invocation exit is called. The program invocation exit and debugger output exit can exchange information by using the CRAB user words. This allows the output exit to clean up resources that are allocated by the invocation exit during the output exit termination call.

Installation Requirements

The program invocation exit takes the form of a separate load module that is named L$UDBNVK. Under CMS, this load module must be placed in the L$CUSER LOADLIB, which is created by the user. Under OS/390, it may be placed in the transient library data set sasc.LINKLIB, or in a separate library that is concatenated ahead of the transient library to an appropriate DDname (STEPLIB or CTRANS).

Dummy Exit Routines

As installed under OS/390, the SAS/C run-time library contains a dummy L$UDBNVK exit routine in sasc.LINKLIB. This routine does nothing; it only returns control to the caller. You can replace this load module with your own exit or delete it. If you delete the dummy exit, your operating environment might produce error messages when the remote debugger is invoked (for example, OS/390 message CSV003I), but the debugger starts up normally.

Note: There is no dummy exit for CMS. △

Assembly Language Implementation

The program invocation exit is invoked under standard IBM linkage conventions. Any registers that are used by the routine should be saved (the standard save area that is addressed by register 13 can be used for this) and restored on exit. Write the exit so that it can be issued in AMODE 31; however, it always starts in the same addressing mode as the remote debugger's first load module.

When the program invocation exit receives control, register 1 points to the following parameter block.

Table 8.1 Program Exit Parameter Block

Offset	**Description**
0	Address of the null-terminated string for the **`_DB_COMM`** environment variable value. This value indicates the debugger communications access method and is ("APPC", "TCPIP",) or the "TCPIP_xxx" variant.
4	Address of the null-terminated string for the **`_DB_HOST`** (TCP/IP) or **`_DB_LU`** (APPC) environment variable value.
8	Address of the null-terminated string for the **`_DB_PORT`** (TCP/IP) or **`_DB_TP`** (APPC) environment variable value.
12	Address of the area that is mapped by C **`struct RDBG_RUNOPTS`** or assembler **`RDBGOPTS DSECT`**.
16	Address of the value from CRABUSR1 (CRAB user word 1).
20	Address of the value from CRABUSR2 (CRAB user word 2).
24	Address of the value from CRABUSR3 (CRAB user word 3).
28	Address of the value from CRABTUSR (CRAB user word 4).

Offset 12 addresses the area that is mapped by the **`RDBGOPTS DSECT`**, as shown in Example Code 8.1 on page 88. This DSECT is created from the run-time options that are specified when SASCDBG is invoked. This DSECT provides access to the name of the program to be debugged, its run-time arguments, and the program invocation method (FORK, OEATTACH, ATTACH, manual).

Example Code 8.1 RDBGOPTS DSECT Mapping

```
         SPACE 3
RDBGOPTS DSECT
RDBBPNAM DS    A        Address of program name to be invoked
RDBGARGS DS    A        Address of runtime arguments passed to program
*
         DS    XL2      --Reserved
RDBGOPTF DS    XL1      Option flag
RDBGSUPD EQU   X'01'    Suppress automatic =D runtime arg insertion
RDBGDBTN EQU   X'02'    Intercept UNIX System Services Terminal I/O
*
RDBGINVM DS    XL1      Request invocation method for program
RDBGMANU EQU   X'00'    Manual invocation
RDBGFORK EQU   X'01     Via UNIX System Services fork and exec services
RDBGOEAT EQU   X'02'    Via UNIX System Services/TSO OEATTACH service
RDBGATTA EQU   X'03'    Via OS/390 ATTACH
*
         DS    0D
RDBGOLEN EQU   *-RDBGOPTS Length of DSECT
```

Offsets 16, 20, 24, and 28 address CRAB user words 1–4, respectively. The CRAB user words are the remote debugger's CRAB, since the program that is being debugged has not yet received control. You can modify the CRAB user words by replacing the values in the parameter block. When the program invocation exit returns, the new values are copied into the debugger's CRAB.

You can also modify other information in the **`RDBGOPTS DSECT`** to affect later debugger invocation processing.

The program invocation exit must return one of the values that are described in "Return Codes" on page 89. These return codes are used by exit routines that are written in both assembly language and C.

C Implementation

You can implement the program invocation exit as a C function by using the following syntax:

```
#include <rdbgnvk.h>
int _dynamn (char *rdbg_comm_method,
             char *rdbg_host_lu,
             char *rdbg_port_tp,
             struct RDBG_RUNOPTS *rdbg_runopts,
             void *crabusr1_copy,
             void *crabusr2_copy,
             void *crabusr3_copy,
             void *crabtusr_copy)
        {
          /* exit code */
```

All arguments are input arguments and are passed by the remote debugger when it calls the exit. The arguments correspond to the information in Table 8.1 on page 87.

The **RDBGOPTS DSECT** maps the C **struct RDBG_RUNOPTS**. This structure is created from the run-time options that are specified when SASCDBG is invoked. This structure provides access to the name of the program that is debugged, its run-time arguments, and the program invocation method (FORK, OEATTACH, ATTACH, manual).

You can modify the information in the **struct RDBG_RUNOPTS** to affect later debugger invocation processing. You can also modify the copies of the CRAB user words. When the program invocation exit returns, the new values are copied into the debugger's CRAB.

The header file **{<rdbgnvk.h>}**, shown in Example Code 8.2 on page 89, is stored in sasc.MACLIBC. The header file defines the prototype for the program invocation exit and the **RDBG_RUNOPTS** structure. It also defines the return code constants **RDBG_CONTINUE_RC**, **RDBG_SUPRESS_MSG_RC**, **RDBG_SUPRESS_DBGINVOKE_RC**, and **RDBG_FAIL_RC**. The program invocation exit must return one of these values, as described in "Return Codes" on page 89.

Example Code 8.2 Remote Debugger Header File

```
#ifndef _RDBGNVK
#define _RDBGNVK
/*
 * This header file defines the prototypes and options struct passed
 * to the L$UDBNVK remote debugger user exit which can be used to
 * start the application to be debugged by the remote debugger.
 */
/* Prototype for exit function */
int l$udbnvk(char *rdbg_comm_method, char *rdbg_host_lu,
             char *rdbg_port_tp, struct RDBG_RUNOPTS *rdbg_runopts,
             void *crabusr1, void *crabusr2, void *crabusr3,
             void *crabtusr);
/* Define return codes for l$udbnvk */
#define RDBG_CONTINUE_RC 0
#define RDBG_SUPRESS_MSG_RC 4
#define RDBG_SUPRESS_DBGINVOKE_RC 8
#define RDBG_FAIL_RC 12
/* Define options struct passed to l$udbnvk exit */
struct RDBG_RUNOPTS
{
 char *pgm_name;    /* Name of program to be invoked               */
 char **pgm_args;   /* Runtime arguments to be passed to program   */
 char optresv[2];
 char optflag;
#define RDBG_SUPPRESS_D 1 /* Suppress auto. =D option insertion     */
#define RDBG_DBTERM_NO  2 /* Turn off Posix Terminal I/O Intercepts */
 char invoke_method; /* Requested invocation method for program     */
#define RDBG_MANUAL   0
#define RDBG_FORK     1
#define RDBG_OEATTACH 2
#define RDBG_ATTACH   3
;
#endif
```

Return Codes

The program invocation exit must return one of the following values:

Table 8.2 Return Codes

Value	**Meaning**
0	Continue with debugger normal processing and program invocation, as defined by **`RDBGOPTS DSECT`** or C **`struct RDBG_RUNOPTS`**. If the program invocation method is manual, issue a message that contains process connection information.
4	Continue with debugger normal processing and program invocation, as defined by **`RDBGOPTS DSECT`** or C **`struct RDBG_RUNOPTS`**, but suppress the process connection information message.
8	Continue with debugger normal processing, but suppress debugger program invocation processing and the connection information message.
12	Terminate debugger connection processing and remote debugger initialization, and issue a failure message.

Typically, return codes 0 and 12 are used as the standard for success and failure status. Return codes 4 and 8 are intended for special purposes. For example, the exit might return the value 4 after starting a program under OS/390 batch. It might return the value 8 if your site uses a custom program invocation method.

CHAPTER 9
Using the Debugger in a Cross-Development Environment

Introduction

The SAS/C Debugger enables you to debug programs in a cross-development environment. To debug a load module that was compiled with the SAS/C or C++ cross-platform compiler, you run the program with the `=debug` run-time option, just like any other SAS/C or C++ load module.

The debugger provides access to information from several different types of files that may be resident on either the UNIX or Windows host or the target mainframe, including

- system-include files
- user-include files
- source files
- alternate source files
- debugger files

In a cross-development environment, the files that are used by the debugger, with the exception of the load module file, may reside on the host workstation. In order for the debugger to access files that reside on the workstation, a distributed file system must establish a client/server relationship between the target mainframe and the host workstation. The Network File System (NFS) is the distributed file system that is used in the SAS/C cross-development environment. For more information, see Appendix 4, "Installing and Administering the NFS Client," on page 309. Using NFS and running on the mainframe under OS/390 or CMS, the debugger has direct access to the source, include, and debugger files that reside on the host workstation.

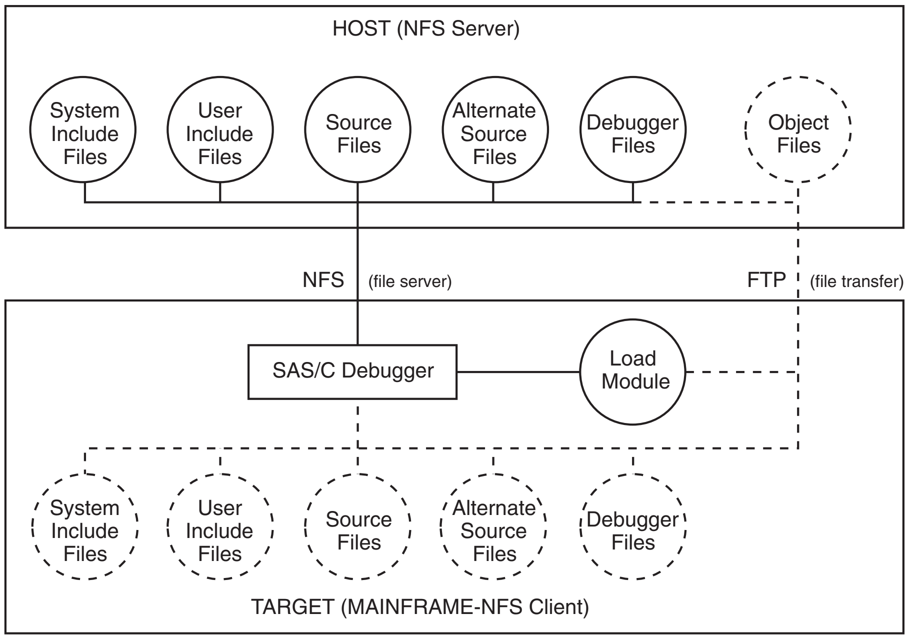
If the debugger's default file-searching mechanism does not meet your needs, you can change or augment the search mechanism with the debugger's **`set search`** command. The **`set search`** command is used to specify filename templates. Filename templates are used to specify the identity and location of the source, include, or debugger files that are associated with the load module that is being debugged. Multiple filename templates can be defined for each type of file. As a result, the debugger can search for a file by more than one name or in multiple locations. Each template is saved in a search list, and each search list is associated with a specific type of file.

Filename templates are character strings that are similar to the patterns in a printf statement. Each filename template may contain *conversion specifiers* and characters. A conversion specifier is a character or a string that is preceded by a percent character. The conversion specifier is either replaced by its associated string or specifies the format of the conversion specifier that follows it. The resulting string is used as the name of the file to be opened. If a file with the resulting name cannot be opened, the next filename template in the search list is processed until either a file is opened or there are no more filename templates in the search list for that type of file.

This powerful technique enables you to direct the debugger to files that have moved or even changed names or file systems. This chapter explains how to use the **`set search`** and **`set cache`** commands in order to define filename templates and to establish search lists.

Figure 9.1 on page 92 illustrates the relationship between the files that are used by the SAS/C Debugger in a cross-development environment.

Figure 9.1 Debugging in a Cross-Development Environment



Using the SAS/C Debugger in a Cross-Development Environment

To debug a program in the cross-development environment, perform the following steps:

1 Compile the program on the host workstation. Specify the **`-Kdebug`** option to specify generation of a debugger file.

2 Create a load module for your program that resides on the target mainframe.

3 Use the NFSLOGIN command in order to access the NFS server network from the mainframe. See “Logging on to the NFS Network” on page 321 for more information.

4 Mount the workstation’s file system from your mainframe client by using one of the methods described in “Accessing Remote File Systems” on page 322.

5 Invoke the debugger. Specify **`set search`** commands in the debugger PROFILE in order to specify search lists for the source, include, and debugger files.

Note: The debugger uses standard **`fopen`** calls in order to access these files. If you encounter difficulty when accessing files, the problem may be caused by your remote file mount and an improper match between the mount point and the templates in the debugger's search lists. For assistance in solving such a problem, enable special tracing by specifying the _SASC_NFS_VERBOSE environment variable. For more information see Appendix 4, "Installing and Administering the NFS Client," on page 309 △

If you do not use the **`set search`** command in order to specify search lists, the debugger resorts to its default search mechanism and uses the filenames that are contained in the object and debugger files in order to locate files. By default, the debugger uses the **`path`**: filename style prefix with workstation filenames. The **`path`**: prefix is described in Appendix 4, "Installing and Administering the NFS Client," on page 309.

Using the Debugger's set Command

The SAS/C Debugger's **`set`** command provides two subcommands: **`set search`** and **`set cache`**. The **`set search`** command specifies a search list that consists of one or more filename templates. Each filename template specifies a location that is used by the debugger to search for source, include, or debugger files that are associated with the load module that is being debugged. The debugger traverses the search list, looking for the file that is specified by each filename template.

The **`set cache`** command is used in cross-development environments that support a distributed file system, primarily to improve the debugger's performance when accessing debugger files. The benefit is especially noticeable when debugger files are large. This command uses a filename template that specifies the primary location to save files and search for files. In a typical cross-debugging session, this location would be on the mainframe.

Note: Frequently, file access problems are caused by an improper mount to the remote file system. If you encounter difficulty with either the **`set search`** or the **`set cache`** subcommands, see "Accessing Remote File Systems" on page 322. △

Locating the Debugger File

When load modules are generated from objects that are compiled by the SAS/C compiler, the load modules contain filename information for the debugger file. The format of this filename information depends on the host that performs the compilation and the file system in which the debugger file is created. When debugging a program that was compiled by the SAS/C Cross-Platform Compiler, the debugger looks for the debugger file in the following locations, in the order listed:

1. any cache location, as specified by the **`set cache`** command
2. any location in the debug search list, as specified by the **`set search debug`** command
3. the original filename that was used by the compiler to open the file when it was created
4. the filename that was used by the compiler to open the file when it was created with the SAS/C filename style prefix **`path:`**.

The debugger first checks to see if a cache location has been specified. The **set cache** command uses a filename template in order to specify a location for the debugger file. For example, the following form of the **set cache** command could be used to specify a cache location in the CMS file system:

```
'SET CACHE DEBUG = "%sname dbg370"'
```

If the debugger file is found in the cache location, that file is opened. If the debugger file is not found in the cache location or the module has been recompiled since the debugger file in the cache location was last copied, the debugger continues to search for the file by performing the remaining steps in the search order. If the debugger file is found, it is then copied to the specified cache location and the new cache file is used.

If no cache location was specified or a debugger file is not found in the cache location, the debugger attempts to find the debugger file by using any filename templates that are defined in the debug search list. Under OS/390, the debugger's default search list for debugger files is equivalent to the following command:

```
set search debug = "//ddn:DBGLIB(%sname)"
```

Note: You can create an empty debug search list with a **set search debug** command of the form: **set search debug = ""**. △

Under CMS, no default templates are defined for the debug search list, so you can define one or more templates. The following form of the **set search** command can be used in order to specify a new search list for the debugger file:

```
'SET SEARCH DEBUG = "cms: %sname db *"'
```

If the debugger file is not found by using the debug search list, then the debugger attempts to open a file with the name that the compiler used when it created the file.

Finally, the debugger attempts to open a file with the name that the compiler used when it created the file, together with the SAS/C filename style prefix **path:**.

Locating Source Files

The debugger file contains filename information for the source and alternate source files that are used to compile your program. The debugger looks for the source file in the following locations in the order listed:

1. any location in the source search list, as specified by the **set search source** command
2. the original filename that the compiler used to open the file when it was created
3. the filename that the compiler used to open the file when it was created with the SAS/C filename style prefix **path:**

Under OS/390, the debugger's default search list for source files is equivalent to the following command:

```
set search source = "//ddn:DBGSRC(%sname)"
```

If a file is not found by using one of the templates in the source search list, the debugger attempts to open a file with the name that the compiler used for the file. Finally, the debugger attempts to open a file with the name that the compiler used when it created the file, together with the SAS/C filename style prefix **path:**.

The source search list is not checked for source files that have been altered by a **#line** preprocessor statement that specify a filename. Instead, the separate **altsource** search list is used. See Table 14.7 on page 255 for more information on **altsource**.

You can use the following forms of the **`set search`** command in order to specify a new source search list:

```
set search source = "template1" "template2"...

set search altsource = "template1" "template2"...
```

Locating Include Files

The debugger file also contains filename information for the system-include and user-include files that are used to compile your program. The different types of include files each have a separate search list. The debugger looks for an include file in the following locations, in the order listed:

1. Any location in the associated search list, as specified by the **`set search systeminclude`** command or the **`set search userinclude`** command.
2. The original filename that the compiler used to open the file when it was created.
3. The filename that the compiler used to open the file when it was created, together with the SAS/C filename style prefix **`path:`**.

You can use the following forms of the **`set search`** command in order to specify a new search list:

```
set search systeminclude =
   "template1" "template2"...

set search userinclude =
   "template1" "template2"...
```

Debugger Performance Considerations

A distributed file system makes it possible to develop your applications in a cross-development environment. In a distributed file system, programs can read or write files directly in a file system on a remote machine. The Network File System (NFS) client support that is provided by the SAS/C Connectivity Support Library enables the SAS/C Debugger to access files that do not reside on the mainframe at all. Additional information can be found in Appendix 5, "Using the NFS Client," on page 321.

The main performance issue to consider when you debug in a cross-development environment is time. A debugger that runs on the mainframe can use a considerable amount of time to access files that reside on the host workstation. In general, you can improve performance by reducing the number of workstation files that are accessed by the debugger.

One method of improving debugger performance is to use the **`set search`** command in order to direct the debugger to mainframe files. For example, when you develop in a cross-development environment, it is likely that identical copies of the system-include files reside on both the host workstation and the target mainframe. Use the **`set search systeminclude`** command in order to direct the debugger to the system-include files that are located on the target mainframe.

Another way to improve performance is to specify a debugger Source Window buffer that is large enough to hold the entire source file. This allows the debugger to keep the entire source file in mainframe memory for the time that the compilation is being debugged. Switching compilations causes the file to be flushed. As a guideline, the amount of memory that is needed to hold one source line is equal to the length of the line, after stripping trailing blanks, plus three bytes. For more information about debugger window buffers, see “Config Window” on page 157. For information about the **`window memory`** command, see “window” on page 280.

It may be advantageous to use a file transfer mechanism, such as FTP, to copy some of the source, include, and debugger files to the target mainframe. For example, your system may not have a distributed file system, or your situation may require you to minimize network traffic. In addition, if you are debugging an application that is composed of many source files and you are only actively developing the code in one or two of those files, the performance of the debugger improves if the inactive source files reside on the target mainframe as well as the host workstation.

Similarly, you may use the **`set cache`** command in order to establish a cache location for your debugger file.

PART 3

Using the Debugger in Your Environment

CHAPTER

10

Using the Debugger

Performing Basic Debugger Actions

Running Your Program

The **go** command starts (or resumes) program execution under the debugger. The program then executes until the next requested breakpoint or action. PF12 is the default PF key assignment for the **go** command. See "go" on page 222 for details.

Stepping Through Your Program

Stepping into Functions, Calls, and Returns

The **step** command resumes execution and breaks execution again when a hook is reached. **step** stops for all hooks, including source lines, function calls, and returns, in the context of both the called and the calling function. The format of the **step** command is

```
step  [INTEGER]
```

The optional INTEGER is a nonnegative decimal integer. Use the INTEGER argument to specify the number of times that you want the **step** command to be performed. The **step** command is usually used without an optional INTEGER argument. This is the most common form of the step command and it causes the debugger to continue execution to the next hook. In other words, it is used to single step through your program. PF11 is the default PF key assignment for the **step** command. See "step" on page 260 for more details.

Stepping over Functions

When you issue a **continue** command, it steps over function calls. The format of the **continue** command is

```
continue [INTEGER]
```

The optional INTEGER is a nonnegative decimal integer. Use the INTEGER argument to specify the number of times that you want the **continue** command to be performed. When a **continue** command is issued at a function call, the function is not stepped into. Execution proceeds to the next line hook in the current function, provided that there are no other breakpoints requested before the next hook. PF10 is the default PF key assignment for the **continue** command. See "continue" on page 204 for more details.

Stopping in Your Program

Setting Breakpoints

The **break** command requests breakpoints at hooks in your program. A common format of the **break** command is

```
break HOOK-TYPE
```

Values for the HOOK-TYPE argument are explained in Chapter 12, "Using Debugger Commands," on page 129. The HOOK-TYPE argument can be one of several possible formats. Another format of the **break** command is

```
break FUNCTION-NAME entry
```

This format sets a breakpoint at entry to the function that is specified by the FUNCTION-NAME argument. See “break” on page 199 for a complete discussion of all the formats for the **break** command.

break *
breaks at every line-number hook in a source file that is compiled with debug.

break entry
breaks on entry to all functions.

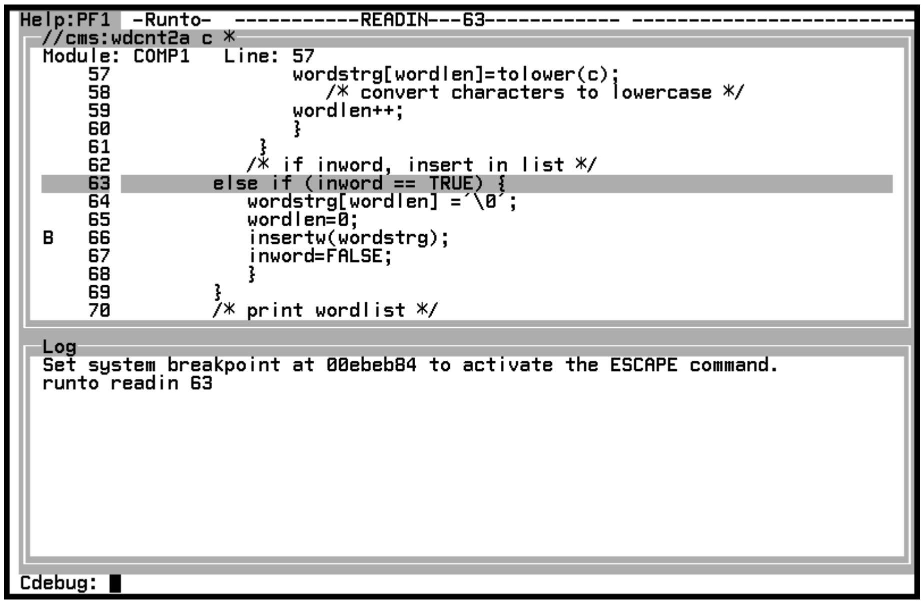
break calls
breaks at each call to a function and at each return from a function.

break func1 entry when (parm ==5)
breaks on entry to the **func1** function when the value of **parm1** is 5.

break 66
breaks at line 66 of the current function.

The prefix-area command **B** breaks on line 66 in Display 10.1 on page 101 and is entered from the prefix area of the line number field. This command can be issued only between the lines that contain the opening and closing braces of a function.
See Table 13.1 on page 175 for a comprehensive listing of the prefix-area commands.

Display 10.1 Prefix-Area Command B Example



```
Help:PF1  -Runto-  -----------READIN---63------------ -----------------------
  //cms:wdcnt2a c *
  Module: COMP1   Line: 57
      57                  wordstrg[wordlen]=tolower(c);
      58                     /* convert characters to lowercase */
      59                  wordlen++;
      60                  }
      61               }
      62              /* if inword, insert in list */
      63           else if (inword == TRUE) {
      64              wordstrg[wordlen] ='\0';
      65              wordlen=0;
  B   66              insertw(wordstrg);
      67              inword=FALSE;
      68              }
      69           }
      70           /* print wordlist */
  Log
  Set system breakpoint at 00ebeb84 to activate the ESCAPE command.
  runto readin 63
Cdebug:
```

Setting Temporary Breakpoints

The **runto** command places a temporary breakpoint at a hook in your program. One format of the **runto** command is

```
runto HOOK-TYPE [when (EXPRESSION)]
```

The hook is identified by the HOOK-TYPE argument. One commonly used HOOK-TYPE is a line number. The **when**(EXPRESSION) argument causes a break at the specified breakpoint whenever the expression is true.
The **runto** command can be thought of as a shorthand method of issuing a **break** command followed by a **go** command. Use the **break** and **go** commands if you want to

stop at a hook several times. In contrast, if you want to stop a hook only once, use **runto**. The **runto** command works particularly well if you know that the hook is hit several times. The reason is that you do not have to drop the temporary breakpoint that is set by the **runto** command. However, if your program might be interrupted before it stops for the temporary breakpoint, do not use the **runto** command. The reason is that the temporary breakpoint is dropped at the interruption.

Breakpoints that are set by the **runto** command are temporary and are removed the first time the program execution stops. This occurs even if execution stops at another breakpoint before reaching the breakpoint that is specified in the **runto** command. Issuing a **runto** command causes the program to run until it hits the breakpoint that is specified, unless execution stops before reaching the breakpoint. See "runto" on page 251 for more details. The following are **examples** of the **runto** command:

runto main 52
sets a temporary breakpoint at line 52 of the main function and then resumes execution.

runto 75 count 5
sets a temporary breakpoint at line 75 of the current function the fifth time the line-number hook at that line is reached and then resumes execution.

Changing Program Execution

The **resume** command is an alias for the **goto** command. It enables you to resume program execution after you correct the cause of an error condition. Also, it can be used to go to any accessible line number or return hook and resume execution from there. The **resume** command can be used for the following types of error conditions:

- an OC4 or OC5 abend, which are identified by the signal **SIGSEGV**. **SIGSEGV** is the illegal memory access signal.
- computational or floating-point errors, which are identified by the **SIGFPE** class of signals: **SIGIDIV**, **SIGFPDIV**, **SIGFPOFL**, and **SIGFPUFL**.
- a call to the function **abort** which is identified by the signal **SIGABRT**.
- a user or system ABEND, identified by the signal **SIGABRT** or **SIGABND**.
- a catch of a thrown C++ exception.

See the SAS/C Library Reference, Volume 1 for a discussion of the characteristics of signals.

When you receive a signal or exception of one of these types, you can examine the value of variables. If you issue a **resume** command, it causes the debugger to discard the signal or exception and resume execution. The location where execution resumes depends on the arguments that you use with the **resume** command, when you issue the command, and where execution in the program stopped. You cannot resume a library function or a function that is compiled with **nodebug**. One format of the resume command is

```
resume [FUNCTION-NAME][LINENO]
```

Arguments

FUNCTION-NAME is the name of a function. LINENO is a source line number. To illustrate what happens when you issue a **resume** command with different arguments, consider the following hypothetical case:

```
  math_a()
   {
   stmt1;                   /* lines of code */
   stmt2;

     .
     .
     .

   p = 0;

27 if (a == b) *p = d;  /* This line causes a SIGSEGV signal */
     .
     .
     .

59 strncpy(p, "XYZ",4);  /* Call strcmp, passing p--which is   */
                         /* bad, and a string "XYZ". This line */
                         /* would cause a SIGSEGV signal to    */
                         /* occur in the library strncpy       */
                         /* function                           */
```

Issuing resume with No Arguments

Suppose a **SIGSEGV** occurs at line 27. Depending on how **math_a** is supposed to work, several different errors could be the reason for the **SIGSEGV**. Perhaps **p** should not have been set to 0, or perhaps **a** and **b** should have been equal. If the problem is that **p** should not have been 0, you can use **assign** to give **p** a correct value and then use **resume** to try the assignment again. Execution starts at the last hook that was encountered before the code that caused the error condition. So, when the first **SIGSEGV** is encountered at line 27, **resume** moves back to the closing parenthesis of **if (a == b)** and retries the rest of the line.

Issuing resume Followed by a Line Number

Execution is resumed at the first hook in the source line that is identified by LINENO in the abending function. The command **resume** 27 retries line 27 from the beginning of the **if** statement.

Suppose **p** is 0, and **a** and **b** are not supposed to be equal. You could use **assign** to change the value of **a** or **b**, and use **resume** 27 to reexecute the **if** statement, thereby avoiding the bad assignment to ***p**.

Issuing resume with a Function Name

Suppose that a **SIGSEGV** occurs at line 59 in **strncpy**. **resume** FUNCTION-NAME resumes execution at the last hook that was executed before the error in the active function (specified by FUNCTION-NAME). In this example, you cannot resume **strncpy** because **strncpy** is a library routine. Therefore, **resume math_a** restarts execution at the last hook in **math_a** before the call to **strncpy**.

Issuing resume with a Function Name and a Line Number

When used with a function name and a line number (for example, **resume math_a 27**) the **resume** command restarts execution at the first hook in the specified line of the named function.

Use **resume** with a LINENO argument for an error that occurs in a function call that takes several lines. **resume** with no arguments only reexecutes the last line of the call; you must issue a **resume** command with the first line in order to reexecute the entire statement. This is important if you correct the problem by changing a parameter in a line of the call other than the last: for example, if **math_a** contains the lines

```
15 longsub(a,b[i],c,
16        d,e,f);
```

If you have a failure in **longsub** that is due to a problem with **b**, and if you change **[i]** and then issue **resume math_a**, the same old **b[i]** values are still passed to **longsub** because **b[i]** is not reexecuted. You have to issue **resume math_a 15** to correct it.

If you issue a **go**, **step**, or **continue** command after the debugger gets control due to a signal, the signal is handled normally, as follows:

- □ If the program has a handler, the handler is called.
- □ If the program does not have a handler, the program abends. If you cannot fix a problem that caused an abend, you may want to type **go**. This lets you get a dump, for instance.

Also, if you issue the **go**, **step**, or $\boxed{\text{continue}}$ command after the debugger has caught a C++ exception (issued a catch command), the C++ exception is handled as follows:

- □ The call chain for the program is searched for an exception handler which is specified by a catch clause that handles the exception.
- □ If an exception is thrown for which there is no corresponding handler, the program will be terminated via the **terminate** function, which calls the **abort** function.

See "resume" on page 246 for more details on the **resume** command.

Displaying, Disabling, Dropping, and Ignoring Breakpoints Requests

The **query** command generates a query list, which is a numbered list of **break**, **catch**, **trace**, **ignore**, **on**, and **monitor** command requests that are currently in effect. When issued with no arguments, the **query** command produces the numbered list of all requests. See "query" on page 244 for more details on the **query** command.

The **disable** command disables requests. Requests are identified by the request number as displayed by the **query** command. One format for the **disable** command is

```
disable ACTION-RANGE
```

ACTION-RANGE is either a single request or a range of request numbers from the **query** list. To disable a single request, supply the number of the request as the argument ACTION-RANGE. To disable a range of requests, give the range, separated by a colon, as the argument ACTION-RANGE. For example, **disable** 2:5 disables request numbers 2, 3, 4, and 5.

Once you have disabled a request, the request is not honored until you enable it again. See "enable" on page 216 for more information. Disabled requests are marked with an asterisk next to the request number in the query list. See "disable" on page 212 for more details on the **disable** command.

The **drop** command drops one or more requests. Requests are identified by request number as displayed by the **query** command. The number that is associated with a dropped command is not reused. Example formats of the drop command are

```
drop ACTION-RANGE
```

```
drop all
```

The first format of the **drop** command allows you to drop requests from the **query** list by number. The ACTION-RANGE argument specifies either a single request number or a range of request numbers. The second form of the **drop** command allows you to drop all request for the entire program. See "drop" on page 213 for details about the **drop** command.

The **ignore** command instructs the debugger to ignore requests that are currently in effect. One format of the **ignore** command is

```
ignore HOOK-TYPE
```

Values for the HOOK-TYPE argument are explained in "HOOK-TYPE Argument" on page 135. As discussed in that section, this can be one of several possible formats. One commonly used format is

```
FUNCTION-NAME line-num1:line-num2
```

This means to ignore requests at source line range in the specified function (FUNCTION-NAME).

The **ignore** command is helpful if you want to ignore requests only at a specific line number or line number range. In other words, you can leave a request in effect except for a line range. You can drop the **ignore** command using **drop**. See "ignore" on page 226 for a description of all the formats for the **ignore** command.

Displaying Variables and Data

The **print** command prints the value of various program elements. The **print** command, when used with the single argument EXPRESSION, displays the expression that is specified in the argument and the value of the expression. The value is displayed according to its type as declared in the source code. This format of the command is

```
print EXPRESSION
```

"EXPRESSION Argument" on page 138 discusses the types of expressions that can be used with the print command and option formats.

The **print** command enables you to determine the contents of a scalar or aggregate, inspect function return values, print function parameters, and look at other program elements and objects. PF16 is the default PF key assignment for the **print** command. See "print" on page 241 for details on all the formats that you can use. The **whatis** command displays the type information that is associated with its argument. One format of the **whatis** command is

```
whatis EXPRESSION
```

EXPRESSION is a valid expression or macro. See "EXPRESSION Argument" on page 138 for more information. Used with this argument, **whatis** displays the type and length of the expression. To display information about macro names, you must compile with **dbgmacro**. The **auto** command keyword **cmacros** does not need to be set.

For example, assume the following declaration for the **whatis** EXPRESSION:

```
struct ABC {
   int x;
   double d;
   } new_struct;

 whatis new_struct::x
  displays:
  int x
```

See "whatis" on page 277 for more details.

Modifying Variables and Data

The **assign** command is similar to the assignment statement that operates on scalars. One format of the assign command is

```
assign SCALAR-TYPE-EXPRESSION = VALUE
```

The **assign** command assigns the value that is specified by the VALUE argument to the object identified by SCALAR-TYPE-EXPRESSION. As described in "SCALAR-TYPE-EXPRESSION Argument" on page 143, SCALAR-TYPE-EXPRESSION is an expression whose type is arithmetic, pointer, or bit-field. VALUE is an expression whose type is one of the following:

- constant
- address
- expression of scalar type
- enumeration constant
- array name.

You can assign a value to any scalar expression that is visible at the point where you issue the **assign** command. For example, after compiling with the **dbgmacro** option, the following declaration and **#define** statement assigns 18 to **avalue**

```
#define A_MIN 9
int avalue;
assign avalue = 9 + A_MIN
```

See "assign" on page 192 for a description of all formats for the **assign** command.

Exiting Your Program

The **exit** command immediately terminates program execution under the debugger, after closing all program and debugger files. PF3 is the default PF key assignment for the **exit** command. See "exit" on page 222 for details.

Performing Advanced Debugger Actions

Displaying a Traceback

The **where** command produces a calling trace or traceback that shows (among other information) the functions that are active at the point in the program from which the **where** command was issued. One format of the **where** command is

```
where
```

The **where** command enables you to see the calling sequence for active functions. Use the **where** command in order to determine the sequence of functions that are active or whether you are executing in the appropriate section.

Because source lines are accessed only for active compilation names and because variables are accessed only for active functions, use the **where** command to display the list of active functions. This allows you to determine what variables can be accessed. For example, suppose that you want to view lines of code (via the **list** command) for a function called **readin**. You could issue the following command:

```
list readin*
```

However, the command does not display the **readin** function in the Source window if **readin** is not currently in the calling sequence identified by your command scope. You can only access variables in functions and sections that are listed in the traceback (that is, active functions). An external variable is accessed from any function whose context contains a declaration for it. See "where" on page 279 for more details on the **where** command.

Viewing and Searching Source Code

Using the list Command

The **list** command lists source lines in a program that is executing under the debugger. In a full-screen session, the **list** command is used to move around in the Source window. For example, to move to a specific line, use the format

```
list LINENO
```

Then, to move back to your current position in your source code use the format

```
list
```

See "list" on page 231 for more details.

Using the window find Subcommand

The **window find** subcommand is used to search for strings and is supported in the following windows:

- Browse
- Log
- Source.

The following format is used with the **window find** subcommand:

```
window find WINDOW-NAME
```

The WINDOW-NAME argument can be any of the following:

- <>
- browse
- log
- source.

If the <> WINDOW-NAME argument is used, the position of the cursor determines the window to which the command is applied. See "Find Window" on page 164 for more details on the **window find** subcommand.

Changing Scope

The **scope** command is used to change the command scope. In order to understand the concept of command scope, you must first understand the concept of run scope. *Run scope* is the term used to describe the location in your program at which the debugger has stopped. Normally, the debugger uses the context of this location to resolve variable references. In other words, the debugger uses the value of a variable as it appears at the location in your code where you stopped.

Command scope identifies a second location in your source code that can be used to resolve variable references. Normally, command scope is the same as run scope; however, it can be changed with the **scope** command. Certain commands use command scope in order to resolve variable references and to supply default function and section names. As described in "Run Scope and Command Scope" on page 21, both command and run scope are displayed in the Status window. Some formats of the **scope** command are as follows:

scope FUNCTION-NAME
changes the command scope to the function that is named by the FUNCTION-NAME argument. This argument must name a function in the calling sequence. If multiple instances of the function are identified by FUNCTION-NAME, the command scope is set to the most recent.

scope + INTEGER
changes the command scope to a function that is farther up in the calling sequence. The INTEGER argument specifies the number of functions up in the calling sequence to change the command scope.

scope - INTEGER
changes the command scope to a function that is farther down in the calling sequence. The INTEGER argument specifies the number of functions down in the calling sequence to change the command scope.

scope
sets the command scope back to run scope.

See "scope" on page 252 for details on the **scope** command.

Stopping Execution When a Value Changes

The **monitor** command causes the debugger to test for changes in the object that is addressed by the expression at each hook. If the value changes, the program is interrupted. This request is called a **monitor**. When the **monitor** command is used to test for changes in a value, the debugger is monitoring the object. A change in value is said to trigger the monitor. One format of the command is

```
monitor EXPRESSION [LENGTH] [print]
```

Arguments and Option Used with the monitor Command

The arguments and option for this form of the monitor command are as follows:

EXPRESSION
identifies the object (such as a variable) to be monitored.

LENGTH
shows the number of bytes to be monitored. The number of bytes can be larger or smaller than the actual length of the object in bytes.

print
prints the value of the object when the monitor is triggered. (The abbreviation is **p**.) With the **print** option, the debugger displays the new value of the object. If the object is no more than 256 bytes in length, the debugger also displays the old value. The values are formatted appropriately for the type of the object. If the object is an aggregate and less than 256 bytes, then only those fields that change are displayed.

Arguments can follow the EXPRESSION in any order. See "monitor" on page 235 for more details on the **monitor** command.

Performing One or More Debugger Commands at Various Locations

The **on** command enables you to perform one or more debugger commands at a specific location in your program. You can issue a list of commands {CMD-LIST} to be performed at locations that are specified with the HOOK-TYPE argument when a certain condition is met (EXPRESSION). Values for the HOOK-TYPE argument are described in "HOOK-TYPE Argument" on page 135. Use the following format for the **on** command.

```
on HOOK-TYPE [when (EXPRESSION)] {CMD-LIST}
```

The CMD-LIST Argument

The CMD-LIST argument contains one or more debugger commands and command arguments, separated by semicolons, as in the following example:

```
on stats c {break; print lengthsum, totcnt}
```

Issuing this command interrupts program execution at calls from **stats** and at returns to **stats**, and prints the value of **lengthsum** and **totcnt**.

The CMD-LIST is enclosed by braces. The CMD-LIST argument may contain nested **on** commands, as in this example:

```
on lookup e when (strcmp(arg, "testid") == 0) {
on lookup * {dump var str; break;}print}
```

In this example, on entry to **lookup**, the following occurs: If the argument to lookup is TESTID, the debugger defines an action for each line of **lookup** and prints the value of arguments to **lookup**. The action, for each line at **lookup**, is requested to dump the value of **var** and interrupt execution. You can use this action in order to track down an error that occurs only on calls to **lookup** with a particular argument, without having to look at calls with other arguments.

This example also illustrates how you can issue commands that are part of the CMD-LIST argument on different lines. Type an open brace at the end of the first line, followed by the list of commands on subsequent lines, and finally the close brace.

Note: If you put commands on separate lines, the end of a line can be the command separator. You do not need a semicolon. You can also enter commands with long CMD-LIST arguments in the Log window by using the \ continuation character. You cannot use the continuation character in the Command window. △

How the on Command Works

When you enter text for an **on** command, the command text goes into the command buffer for only a moment. Then the debugger processes the CMD-LIST argument and stores it as text in an internal debugger data structure. No parsing is done to the **on** command's CMD-LIST at the time it is entered. When the hook that you specify is reached, the CMD-LIST is reinserted into the command buffer, this time parsed and ready for execution. If, at the time you enter an **on** command, there is already another debugger command in the buffer, the **on** CMD-LIST arguments are placed before the other command (or commands). You can think of an implied **go** command as being placed after the **on** command.

For example, suppose you want to break at line 15 in your program and dump two variables, **p** and **g**. Issue the following **on** command:

```
on main 15 {dump &p; dump &g}
```

The command is put in the command buffer momentarily and then is put (as text only) into some internal data structure. Now you reach line 14 and type the following commands:

```
go; print z; print w;
```

The debugger executes the **go** command and reaches line 15. As this point, the command buffer is

```
go; print z; print w;
     ↑
```

Your **on** command and an implied **go** command are inserted into the buffer after **go**, where the arrow is pointing above, and execution continues. The buffer now looks like

```
go; dump &p; dump &q; implied go; print z; print w;
    ↑
```

The debugger executes the inserted commands and then executes the added **go**. This means that the two print commands are not executed at line 15. In fact, they are not executed until the next breakpoint. See “on” on page 239 for more details about the **on** command.

Setting Different Debugger Modes

The **auto** command lets you set keywords that define several characteristics of output produced by the debugger. The format of the **auto** command is

```
 auto KEYWORD KEYWORD . . .
```

For example, the **cmacro** keyword allows substitution of program macros in expressions. In order to use the **cmacro** keyword command you must compile your program with the **dbgmacro** option. Suppose you had the following line in your program:

```
#define MAXLEN 15
```

Then at the command line you would type:

```
Cdebug: auto cmacro
```

And to print the value of MAXLEN you would type:

```
Cdebug: print MAXLEN
```

which prints a value of 15 to the Log window.

See “auto” on page 196 for a detailed discussion of these keywords and other **auto** command keywords.

Viewing Memory

The **dump** command dumps the contents of storage that is pointed to by an EXPRESSION. Formats of the dump command include the following:

```
dump EXPRESSION
```

```
dump EXPRESSION str
```

The EXPRESSION argument is either a pointer, an address, or an array. An absolute address must be specified with a leading 0p, for example, 0p00001234. For the first format of the **dump** command, the number of bytes dumped is determined as follows:

Table 10.1 Dump Command: number of Bytes Dumped

Argument Type	Number of Bytes Dumped
Pointer	The size of the pointed to object
Address of a scalar or aggregate	The size of the scalar or aggregate
Array	The size of one item of the array
Absolute address	One (treated as a pointer to char)

The second form of the command dumps the contents of storage that is pointed to by the EXPRESSION until the null terminator, \0, is encountered.

For all formats, the output of the **dump** command shows the contents of the EXPRESSION argument in characters and in hexadecimal format, and shows the address of the argument as a hexadecimal number.

In the following example, because **str** is specified, the entire string **test** is dumped:

```
Cdebug: dump wordstrg str


wordstrg


0002b1d0 a385a2a3  *test            *
```

In the next example, **inword** is an **int**, so 4 bytes are dumped (**int** type has a size of 4 bytes):

```
Cdebug: dump &inword


&inword


0002b1cc 00000001  *....             *
```

See “dump” on page 215 for more details on the **dump** command.

Defining Macros

The **define** command enables you to use the debugger commands as macros. Once you have defined a macro, you can invoke it by using the macro name, prefixed with #. One format for the **define** command is

```
define DMACRO "REPLACEMENT TEXT"
```

The DMACRO argument is any valid identifier. The REPLACEMENT TEXT argument is the debugger command that is substituted for the macro when you invoke the macro. The REPLACEMENT TEXT argument can contain double quotation marks, but you must escape the quotation marks with a backslash. Drop debugger macro definitions with the **undef** command when you no longer need them. See “undef” on page 275 for a discussion of the **undef** command.

Do not confuse C macros with debugger macros. C macros are defined in your program via the C preprocessor **#define** statement. Debugger macros define a shorthand version for commands that you plan to use often in a debugger session. In the following example, the **define** command defines a macro **dmp** as **dump wordarry str**:

```
Cdebug: define dmp "dump wordarry str"
```

See “define” on page 210 for more details on the **define** command.

Executing Operating Environment Commands

The **system** command sends an operating environment command to the operating environment. The **system** command is equivalent to calling the system function from within your C program by using the TSO: or CMS: prefix. The format of the **system** command is

```
system OPERATING-SYSTEM-COMMAND
```

Under CMS

The **system** command uses full command resolution as if the command were entered in response to the CMS prompt. See Chapter 14, “Command Directory,” on page 187, for more information on the system under CMS.

Under TSO

The OPERATING-SYSTEM-COMMAND is either a native TSO command or a CLIST or REXX EXEC that contains TSO commands.

Displaying Storage Analysis

The **storage** command prints an analysis of the program’s use of both heap and stack storage. This analysis can be useful for locating a memory overlay. To display a report of both heap and stack storage, type

```
Cdebug: storage
```

The following output is an example of a report:

```
At      SUB1(PGM1)     entry    ------
  SIZE: FREE/USED  SIZE: FREE/USED  SIZE: FREE/USED  SIZE: FREE/USED
    24:    0/116     32:    0/118     40:    0/18      48:    0/69
    56:    0/10      64:    0/7       72:    0/59      80:    0/10
    88:    0/9       96:    0/4      104:    0/3      120:    0/2
   144:    0/5      152:    0/1      160:    0/1      280:    0/1
   512:    1/0      792:    1/0     1152:    1/0     1472:    1/1
  1792:    0/2     8192:    0/1
No corruptions found in heap.

  SIZE  NUMBER  SIZE  NUMBER  SIZE  NUMBER  SIZE  NUMBER  SIZE  NUMBER

   152:    1     168:    1     208:    1     248:    1     256:    1
   296:    2     672:    1
 Total unused space in stack (bytes): 1768
No corruptions found in stack.
```

Halting Your Running Program

The **attn** command generates a **SIGINT** interrupt signal in programs that execute under the debugger. When the **SIGINT** signal occurs, it sends a message to the debugger output. The format of the **attn** command is

```
attn
```

You generate the **SIGINT** signal under TSO by pressing the attention (PA1) key, and under CMS by using the immediate command IC. However, when you are using the debugger, pressing the attention key under TSO, IC under CMS, or CTRL+C under UNIX System Services gives control back to the debugger. To actually send a **SIGINT** signal to the executing program under the debugger, you must use the **attn** command.

Note: When using the debugger in full-screen mode under CMS, you might not be able to issue an immediate command. Another method of interrupting the debugger under CMS is to press the attention key to give control to the VM control program CP, and then issue the CP command EXTERNAL DB. This sends an external interrupt of the debugger, which interprets it as an attention. △

See “attn” on page 195 for more details on the **attn** command. See the discussion of signal handling in the SAS/C Library Reference, Volume 1 for details on the characteristics of signals.

CHAPTER 11

Debugging C++ Programs

Introduction

For the most part, debugging C++ programs is the same as debugging C programs. There are only a few differences, which are the focus of this chapter. This chapter does not attempt to teach you how to set up and use the basic features of the debugger.

Specifying C++ Function Names

One of the unique features of the SAS/C C++ Development System is that the debugger accepts and understands C++ function names, including multitoken and overloaded function names. This section describes how to specify C++ function names in debugger commands.

If you are specifying a nonoverloaded, single-token function name in a debugger command, you do not have to do anything differently from when you are specifying a C function name. For example, you could issue the following command:

```
break func1 entry
```

There are additional rules, however, for specifying multitoken C++ function names and overloaded function names in debugger commands. This section also explains how to specify member function names and file-scope function names.

The rules for specifying constructor and destructor function names are unique to these types of functions and are covered separately. See "Functions in a Mix of C and C++ Code" on page 118 for information on how the debugger handles function names when you mix C and C++ code. Also, see "Translator-Generated Functions" on page 118 for information on how the debugger handles translator-generated functions (such as assignment operators and copy constructors).

Multitoken Function Names

Multitoken function names are function names that are not only a C++ identifier, but that contain other items such as the scope operator (::) or two-word function names such as operator and conversion functions. Here are some examples of multitoken C++ function names:

- **`ABC::ABC`**
- **`myfunc::~myfunc`**
- **`operator int *`**
- **`ABC::operator >=`**

When you specify a multitoken C++ function name in a debugger command, the function name must be enclosed in double quotes. Here is an example of the **`break`** command and a multitoken function name. This command specifies to break at the entry to member function **`func1`** in **`class ABC`**:

```
break "ABC::func1" entry
```

Spaces around tokens that are not identifiers are optional.

Overloaded Function Names

One of the things that sets C++ apart from C is that C++ supports overloaded functions. However, overloaded functions present a challenge for the debugger because the debugger has to determine which function you want to access.

When you specify an overloaded function name in a debugger command, you are presented with a numbered list of C++ function names with arguments. Determine which number represents the function you want to access, and reissue the debugger command by appending a parenthesized number after the function name. For example, suppose you have the following three constructors declared in **`class myclass`**, in this order:

```
myclass(char);
myclass(short);
myclass(long);
```

If you issue a **`break "myclass::myclass" entry`** command, the debugger shows you the following list:

```
1  myclass::myclass(char)
2  myclass::myclass(short)
3  myclass::myclass(long)
```

You can place a breakpoint on entry to the constructor that takes a **`short`** integer by specifying the following **`break`** command:

```
break "myclass::myclass"(2) entry
```

As long as you do not relink your program, the subscript numbers for overloaded functions remain the same. For example, you can define a debugger macro or alias using the subscripts and use it throughout your debugging session.

Instead of choosing a particular number, you can specify that the command apply to all instances of the function by using 0 as the parenthesized number. For example, the following command sets breakpoints on entry to any **`myfunc`** function in **`class myclass`**, regardless of the argument type:

```
break "myclass::myfunc"(0) entry
```

However, a subscript of 0 is valid only at entry hooks, return hooks, call hooks, or * (that is, all line hooks). The only commands that permit a subscript of 0 are **`break, trace, ignore, runto`**, and **`on`**.

See “Interpreting C++ Demangled Names” in the SAS/C C++ Development System User’s Guide for more information on interpreting the overloaded function names.

File-Scope and Member Functions

The debugger uses the scope operator **`(::)`** to determine if you want to access either a filescope or a member function.

If you have declared both a filescope **`myfunc`** and a member function **`myfunc`** in **`class ABC`**, use the scope operator to tell the debugger which function you mean when you issue debugger commands as follows:

`"::myfunc"`
refers to a file-scope function of name **`myfunc`**.

`"ABC::myfunc"`
refers to a member function of name **`myfunc`** in **`class ABC`**.

If you have only a file-scope function named **`myfunc`**, or only one member function named **`myfunc`** (but not both a file-scope function and a member function), you can omit the scope operator and specify only the function name in the debugger command.

Note: If the debugger is stopped in a member function when you issue a debugger command that includes only the function name (and no scope operator), the command works as though the debugger were not stopped in a member function. That is, the debugger does not automatically prefix the function name with the class name of the class whose member function you are stopped in. This is slightly different from the behavior for data objects, in which the class name is automatically prefixed. See “Searching for Data Objects” on page 121. △

Constructors and Destructors

When you specify a constructor or destructor in a debugger command, it must be in one of the following two forms:

"***`class-name::class-name`***"
indicates a constructor.

" ***class-name::~class-name*** "
: indicates a destructor.

Here is an example of setting a breakpoint on entry to the destructor for **class ABC**:

```
break "ABC::~ABC" entry
```

Functions in a Mix of C and C++ Code

If your load module contains at least one C++ compilation, your load module also contains a list of all function names that are visible to C++ compilations. If you issue a debugger command that refers to a function name that is not in this list, the debugger issues a warning message and assumes the function is a C function.

For example, suppose you have the following construct, in which **a()** is a C++ function and **b()** and **c()** are C functions:

a() calls **b()** calls **c()**

There is a function prototype for **b()** in the compilation containing **a()**. Because **b()** is visible to a C++ compilation, it is contained in the debugger's list of visible function names. Because no function prototype for **c()** is visible in any C++ compilation, **c()** is not contained in the list of function names that are visible to the debugger. If you use **c()** in a debugger command (such as in a **break** command), the debugger issues a message that it cannot resolve the function name by using the debugger file. The debugger assumes that **c()** is a C function.

Note: If you see the warning message about unresolved function names yet you know that your program consists of only C++ functions, check the spelling of function names in your debugger commands. △

Translator-Generated Functions

The translator creates a number of functions automatically. These functions are required by the C++ language and follow the usual C++ rules. The functions that the translator may create include constructors, copy constructors, assignment operators, and destructors. The translator creates such a function when there is not a user-defined version of the function. The list of overloaded constructors that is displayed by the debugger when you issue a debugger command may include a translator-generated constructor as well as the user-defined constructors.

The following list shows the declarations for translator-generated functions:

***class*::*class*()**
: is the default constructor for class ***class***. This constructor is called whenever an object of type **class** is defined without an explicit initializer.

***class*::*class*(const|volatile *class*&)**
: is the default copy constructor for class ***class***. A default copy constructor is created for any **class**, **struct**, or **union** that does not have a user-defined copy constructor. The copy constructor is called to initialize an object of type **class** with another object of the same class. The presence of **const** or **volatile** depends on the characteristics of the class.

class& class::operator=(const|volatile class&)
: is the default assignment operator. This operator is called when an object of type **class** has an object assigned to it. The presence of **const** or **volatile** depends on the characteristics of the class.

class::~class()
: is the default destructor. This destructor is called when an object of type **class** goes out of scope.

You may occasionally step into one of these translator-generated functions as you debug your code. When this happens, the Source window displays the source text at the class definition and the Status window displays the function name.

Specifying Expressions

The debugger supports the use of operators, types, and casts that are specific to C++. This section delineates these items and explains how expressions are evaluated for C++ programs in the debugger. Note that only standard C++ operators are supported in expressions. That is, user-defined overloaded operators cannot be used. This includes the use of complex and I/O stream operators. For example, you cannot specify **`print (a+b)`** where **`a`** and **`b`** are complex.

Operators

The following operators are supported in debugger expressions:

`::` (unary scope)
indicates the scope operator (identifies the object or function as file-scope).

`::` (binary scope)
indicates the scope operator (identifies the object or function as a member of a class).

`->*`
indicates a member-pointer.

`.*`
indicates a member-pointer.

Only one level of **`::`** is supported after a **`.`** or **`->`** operator. For example, the following is not valid syntax in a debugger command:

```
p->A::BB::c
```

Casts

In addition to the syntax for casts supported for C in the debugger, the keyword **`class`** is supported as in

```
(class TAG*)ADD
```

Casts to reference types are not supported. If the two classes that you are referencing are related (that is, one is derived from the other), the debugger performs the cast and issues a message that indicates address translation may have occurred.

Data Types

All debugger commands that support expressions support the following C++ data types:

- pointers to base or derived classes
- member-pointers

- references
- classes.

The following sections detail any special considerations for using debugger commands with C++ data types. Static members of class objects do not participate in any **assign**, **copy**, **dump**, **monitor**, **print**, **return**, or **watch** commands that handle objects of type **class**. For example, because all classes share the same **static** data, if you copy a class with the **copy** command, you do not modify static members.

A member-pointer is not considered a pointer in the C or C++ sense. Therefore, it is invalid to specify an expression of type member-pointer in a command (such as **dump**) that takes an address for an operand.

assign Command

The **assign** command can be used to assign a pointer to an object of a derived class to a pointer to the base class. When multiple inheritance is used, this can cause the values of the derived pointer and the base pointer as printed by the **print** command to differ. This can also occur for assignments that involve member-pointers.

An assignment to a reference assigns to the referenced object.

An assignment to a class object is permitted using an initializer list or a class object only if the class does not have base classes and if no user-defined constructors need be invoked to initialize the class object.

dump Command and Dump Window

The **dump** command and the Dump window support all the data types that were listed earlier in this section. The **dump** command takes an address as an argument and, by default, dumps memory corresponding to the size of the object. A member-pointer can be one of two sizes, depending on whether it is a data pointer or a function pointer:

data pointers	are 4 bytes long.
function pointers	are 12 bytes long.

A member-pointer is not considered to yield an address type. Therefore, the following command is not valid:

```
dump member-pointer
```

The following command dumps the referenced object:

```
dump reference-object
```

monitor Command

You cannot monitor an object through a reference variable. You can monitor only the reference variable itself. If you set a monitor on a class object that contains a reference, only the storage allocated for the class object (which includes the storage allocated to the reference variable) is monitored. The referenced object is not monitored.

For objects of derived classes, the base objects are also monitored. Because the entire storage of an object is monitored, it is possible for the monitor to be triggered without any change in the printed value. That is, for some data types (such as function member-pointers), some parts of the value may not be reflected in the value produced by the **print** command. The debugger attempts to detect corruption of control data and hidden pointers to virtual base classes (that is, members of a class other than those members you have explicitly created). If your program is erroneously overwriting memory, it could overwrite some of these control data or hidden pointers. Even though this corruption does not affect the value that is printed, the monitor is triggered.

return Command

For a function that returns a reference, the **`return`** command returns a reference. A function may return a class using an initializer list or a class object only if the class does not have base classes and if no user-defined constructors are needed to initialize the class.

transfer Command

For a reference type object, use the C++ notation for references (such as **`class myobject&`**) in the **`typeof`** keyword of the **`transfer`** command.

whatis Command

The **`whatis`** command uses the C++ notation for references (such as **`myobject&`**) in its output.

For classes, **`whatis`** displays the following information, in addition to the usual C information:

- base classes
- access attributes
- all non-C member types that can occur as class members (classes, **`enums`**, **`typedefs`**, and functions, including function prototypes).

Expression Evaluation

Normal C++ rules are followed in expression evaluation, with the following exception. If a **`static`** member is dereferenced using the member selection operator (**`->`**) or the selection operator (**`.`**), the expression to the left of the **`->`** or **`.`** operator is evaluated by the debugger.

Searching for Data Objects

Normal C++ scoping rules apply to most searches in the debugger. When you are stopped at a line of code in a member function, you do not have to specify this **`->`** to access class members. If your source code is structured so that classes are nested within classes, the debugger also searches any lexically enclosing scopes. The debugger applies normal C++ rules for ambiguity resolution.

You can access **`static`** members by using the ***`class-name`*`::`*`member-name`*** syntax.

Debugging Initialization and Termination Functions

A typical C++ program contains initialization functions in each compilation at program startup. These functions are called at program startup to initialize static and extern data defined in that compilation. If you want to debug one of these initialization functions, you can set a breakpoint on **`_ _init`** (i.e., **`break _ _init entry`**). You will then be presented with a numbered list of initialization functions, one for each compilation using the sname for that compilation. See the discussion of the sname option in Chapter 3, "Translator Options," of theSAS/C C++ Development System User's Guide for more information. Select the initialization function that you want to debug.

See the section "Overloaded Function Names" on page 116 for information on how to select an overloaded function from the list.

Similarly, you can use **`_ _term`** for a numbered list of termination functions. By default, the first function name shown in the Status window is one of these initialization functions. While the debugger is stopped in one of these functions, you can debug the initialization of static and extern variables. As you step through the initialization functions, each function is, in turn, shown in the Status window. Note that the initialization and termination functions are not shown in the Source window, as they do not exist in user C++ code.

Bypassing Initialization Functions

If you do not want to debug your program's initialization functions, you can bypass them in one of the following ways:

- ☐ set breakpoints in functions that are of interest and issue a **`go`** command
- ☐ put a **`Break main entry`** command in your debugger profile and issue a **`go`** command (this causes debugging to begin at the **`main`** function)
- ☐ issue a **`Runto main entry`** command from the command line (this causes your program to advance to **`main`**).

Setting Breakpoints in Dynamically Loaded C++ Modules

The debugger keeps track of load modules that contain C++ code as they are loaded and unloaded. When you specify a function in a debugger command, the debugger first looks for the function in the current load module. If it fails to find it there, the debugger searches the list of modules that have been loaded. If you want to set a breakpoint in a module that has not yet been loaded, you can set a breakpoint on entry to **`_dynamn`**. Then, when you reach this breakpoint, set the desired breakpoint in the module. Note, however, that **`_dynamn`** is called only after constructors for static and extern objects in the loaded module have been run.

C++ Debugging Example

This section provides an example of running the debugger with a C++ program. Some of the features the example illustrates include

- ☐ the need to enclose most C++ function names in double quotes.
- ☐ how to specify overloaded C++ function names using a subscript.
- ☐ how to specify all overloaded functions at once using a subscript of 0.
- ☐ how to set breakpoints in special C++ functions, such as constructors and destructors. (Specifically, this example shows that user-defined constructors and destructors can be debugged even when they are driven by the C++ **`new`** and **`delete`** operators.)
- ☐ how to debug C++ class member functions.

The important thing to remember is that debugging C++ programs with the SAS/C Debugger is virtually the same as debugging C programs. The debugger looks and feels the same, and in general the commands are the same. When you have completed this example debugger session, you should be ready to debug your own C++ programs.

Note: This example requires you to allocate the DDname DBGSLIB to the location of your standard header files. See the SAS/C Library Reference, Volume 1 for more information about the DBGSLIB DDname. △

Example Source Code

Here is the source code for the example. The program declares **`class X`**, which includes one data object, two constructors, one member function, and a destructor. The member function multiplies the data object by 2. The destructor checks the value of the data object and prints an error message if the data object is not between 7 and 10. The **`main`** function uses the constructors to create several instances of class X, calls the member function four times (once for each instance of **`class X`**), and then deletes the instances of **`class X`**.

```
#include <iostream.h>
class X
{
public: int i;
      // first X constructor
   X(int ia)
   {
      i = ia;
   };
      // second X constructor
   X(int ib, int jb)
   {
      i = ib + jb;
   };
       // X member function myfunc()
   void myfunc()
   {
      i *= 2;
   };
      // X destructor
   ~X()
   {
      int id;
      id = i / 2;
      if (id < 7 || id > 10)
      printf("Error - Out of range\n");
   };
};
int main()
{
   X *x1 = new X(7);
   X *x2 = new X(6,2);
   X *x3 = new X(9);
   X *x4 = new X(9,1);
   x1->myfunc();
   x2->myfunc();
   x3->myfunc();
   x4->myfunc();
   delete x1;
   delete x2;
```

```
    delete x3;
    delete x4;
    return 0;
}
```

Sample Debugger Session

In this example debugger session, you use the **break, go, query, drop**, and **on** debugger commands to complete the program. The numbered steps that follow tell you what to type in and show the results of your commands in the various debugger windows.

You first need to translate, link, and run the sample program. Be sure to specify the **debug** option when you translate.

When you run your program, the debugger stops on a line of code in **iostream.init** function, as described in "Debugging Initialization and Termination Functions" on page 121.

Issue the following debugger commands in sequence. Debugger commands are issued after the **Cdebug:** prompt located in the Command window at the bottom of the screen.

1 **break "X::X"(0) entry**

   This command sets breakpoints at entry to all constructors for **X**. The subscript of 0 is necessary because the constructors are overloaded functions. Because the function name has a multitoken name, double quotes are necessary.

2 **break main 37**

   go

   These two commands first set a breakpoint at line 37 of the **main** function and tell the debugger to advance to the first breakpoint. The debugger stops on line 37 of **main**.

3 **go**

   The program proceeds until it enters the first constructor.

4 **go**

   Yet another **go** causes the debugger to stop at the entry to the next constructor.

5 **query**

   This command requests the Log window to show all actions and monitors in effect. Two breakpoints are shown; of special interest is the first breakpoint, which shows the subscript of 0, indicating a breakpoint is in effect for all instances of the overloaded constructor, X.

6
```
drop 1
break "X::X"(1) return
break "X::X"(2) return
on myfunc entry print i
 on myfunc return print i
b "X::~X" return
query
```

   The **drop** command drops breakpoint #1 (it is no longer necessary). Next, set breakpoints on the return of both versions of the overloaded constructor, using the subscripts 1 and 2. The two **on** commands tell the debugger to print the value of the **X** member function **i** on the entry to and return from the **myfunc** member function. The next command sets a breakpoint on the return of the destructor (**b** is an abbreviation for **break**). Finally, the **query** command shows all the actions and monitors that are in effect. Of special interest are the third and fourth breakpoints. These breakpoints show the prototypes for the first and second constructors (the first takes a single **int**; the second takes two **int**s).

7 go

This **go** command causes the debugger to proceed until it reaches the return from the second constructor.

8 go

Another **go** causes the debugger to proceed until it reaches the return from the first constructor.

9 go

This **go** command causes the debugger to stop again at the return from the second constructor.

go

After this **go** command, the Log window shows the output from eight **print** commands (4 from the entry to **myfunc** and 4 from the return from **myfunc**). The values printed are 7, 14, 8, 16, 9, 18, 10, and 20. Also, notice that the debugger stops at the return from the destructor.

10 go

go

go

At each of these **go** commands, the debugger stops at the return from the destructor.

11 break "X::Y" entry

This command illustrates the warning message that is issued by the debugger when it cannot find a function in the debugger's list of visible functions. Because **X::Y** is not a valid function name for this program, a warning message appears in the Log window.

See "Functions in a Mix of C and C++ Code" on page 118 for more information about when you may see this warning message.

12 go

After this last **go** command, the execution completes, and the debugger terminates.

PART 4

Reference

CHAPTER
12

Using Debugger Commands

Introduction

This chapter provides background information you need to use SAS/C Debugger commands effectively. These commands provide information about what is happening at various points in an executing program and enable you to tell the debugger which parts of the program you want to work on.

The way that you issue debugger commands depends on the method that you use to run the debugger. In full-screen mode you use the Command window, Log window, and PF keys; in a line-mode session you use the command line; and in a batch session, debugger commands are part of your JCL job stream.

Command Formats Used with the Debugger

This section explains how formats are used to describe debugger command syntax. It also provides guidelines and constraints that are useful when you invoke the debugger, and a description of the various symbols used in commands.

Formats

Some commands consist of a single word, for example, **`exit`**. Others, such as **`print`**, can be issued as one word or with one or more arguments. Some command arguments can be nested. Given the possible length and complexity of the commands that are issued to the debugger, syntax descriptions do not show the complete syntax of the command in a single string. Rather, for commands that take many arguments, a separate prototype, or format, is created for each possible configuration of arguments.

For example, the syntax of the **`print`** command can appear in the following three formats:

Format 1:	**`print`**
Format 2:	**`print`** EXPRESSION
Format 3:	**`print`** EXPRESSION [(PTYPE)][%FMT] [COUNT][,EXPRESSION[(PTYPE)] [%FMT][COUNT]]

The following list gives typographical and syntax conventions:

- The name of the command is **`print`**, as indicated by the lowercase, bold type.
- Four different types of arguments can be used in several different combinations with the **`print`** command (EXPRESSION, PTYPE, %FMT, and COUNT).

Arguments are indicated by uppercase letters. These arguments are described in "Arguments in the Debugger Formats" on page 134.

- Several optional arguments can be used with the third format, as indicated by the square brackets.
- The PTYPE argument must be enclosed in parentheses. The parentheses are part of the command syntax, as described in "Command Symbols" on page 132.

Chapter 14, "Command Directory," on page 187 shows the entire syntax of a command by presenting a series of formats that illustrate combinations of arguments, from simple to complex.

Guidelines

Chapter 14, "Command Directory," on page 187 is a detailed reference for all commands and command formats. It illustrates the syntax of debugger commands. The following guidelines apply to all command formats.

Characters

You can use upper- or lowercase characters for debugger commands and keywords. Check with your SAS Installation Representative for SAS/C products to determine if special characters such as braces or brackets have been assigned alternate representations at your location. See Appendix 2, "Character Set Defaults for Special Characters," on page 299 for more information.

Constant Pointer Specification

Type constant pointers with a leading `0p`, such as `0p0001b123`.

Syntax Errors

When you type a command incorrectly (depending on the command and the arguments that you enter with it), the debugger may reject the command, attempt to execute the command, or enable you to reissue the command. Appendix 1, "Error Handling," on page 297 describes some of these possibilities and the messages that you receive.

Multiple Commands on One Line

You can issue more than one command on a single line. If you do, however, you must separate the commands with semicolons.

If you issue several commands on one line and any one of them has a syntax error, that command and all subsequent commands on the line are rejected. You receive a message, and the debugger prints all the rejected commands.

Command Continuation

In full-screen mode you can use the Log window to type exceptionally long commands. (Debugger commands can be issued from any location inside the border of the Log window.) With the exception of the `on` command, there is no way to continue a long command in the Command window. You can use a backslash (\) to continue commands across lines. You can split a command and its arguments across lines, provided that the backslash is the last character on the line being continued. The backslash (\) can also be used to continue commands across lines when you are running the debugger in line mode.

Note: You cannot use the backslash (\) character to continue commands in the Command window. Long CMD-LST arguments to the **on** command can be continued in the Command window, provided that you type the initial brace before you press the ENTER key. The CMD-LST is continued until the closing brace is entered. Commands in the CMD-LST cannot be split between lines. △

Identical Requests

For the **break**, **trace**, and **ignore** commands, *identical requests* are uses of the command that apply to the same line range and have the same WHEN clause if there is a WHEN clause. For the **on** command, identical requests apply to the same line range, have the same WHEN clause (if there is one), and have the same text in the command list. For the **monitor** command, identical requests monitor the same program element in the same environment.

When you type a request that is identical to an existing request (as defined above), the debugger does not honor the second request, but sends a message. If the existing request is disabled when the identical request is made, the identical request is dropped, and the existing request is silently enabled.

Abbreviations

You can use abbreviations for commands and arguments. The accepted short form (or forms) for each command is listed in Chapter 14, "Command Directory," on page 187.

Many keywords can also be abbreviated in debugger commands. The following are a few examples:

a{ll}

c{alls}

e{ntry}

r{eturn}

s{tr}

The characters that precede the braces are the acceptable abbreviations. You can type the abbreviation plus any characters within the braces.

Command Symbols

The command symbols that are described in this section can be typed as part of certain debugger commands. Do not confuse command symbols with similar symbols that are used to describe syntax. The following command symbols have special meanings in the debugger syntax:

;	The semicolon separates multiple commands on a single line.
,	The comma separates arguments in commands, such as the **drop** and **print** commands.
{ }	Curly brackets surround commands in an **on** command.
( )	Parenthesis surround some arguments (such as a section name) in certain commands.
:	The colon is used in the following two contexts: □ It indicates a line number range. The first line in the range is on one side of the colon; the last line is on the other side, for example, 100:120.

- It indicates a variable in a function that is different from the function that you are in when the colon is preceded by a function name and followed by an identifier in expressions.

%	The percent sign indicates a print output format that is similar to the format specifier string of the **`printf`** function, when % is used in the **`print`** command and followed by a **`printf`** format specification. See "print" on page 241 for more information.
\	The backslash is used as an escape symbol. It is used in the **`print`** command to escape the modulus operator (%) so that the debugger does not interpret % as a format marker, and in the **`copy`** and **`whatis`** commands with certain arguments it is used. Also, in line mode or when you are issuing commands from the Log window, you can use the backslash to continue a debugger command to a new line if it is the last character on the line.
*	The asterisk requests breakpoints or actions at hooks. See Table 2.1 on page 28 for more information.
–	The minus sign indicates a relative direction toward the beginning of your source code. This symbol is used with the **`list`** command, and it has a different effect in line mode than it has in full-screen mode.
+	The plus sign indicates a relative direction toward the end of your source code. This symbol is used with the **`list`** command, and it has a different effect in line mode than it has in full-screen mode.

Notice that the last three symbols (*, –, and +) are also valid operators in expressions that are used in debugger commands. The way that you use the symbol determines if it is interpreted as a command symbol or an operator. The debugger accepts the operators and symbols that are listed in Table 12.2 on page 139. It also accepts any special characters that are defined for your site. See Appendix 2, "Character Set Defaults for Special Characters," on page 299. You also can use the concatenation operator (| |) for brackets [ ]and angle brackets (< >) for { }.

Requesting Breakpoints and Actions

Commands such as **`break`**, **`trace`**, and **`on`** request actions or breakpoints. Both breakpoints and actions cause an interruption in program execution at a hook. However, after a breakpoint you must restart program execution; by contrast, after an action, execution continues automatically. When you request a breakpoint with **`break`**, for example, you can then issue additional debugger commands (such as **`print`**, to print the value of a variable) and then restart execution by using the **`go`** command.

Conditional Breakpoints and Actions

In addition to the argument that specifies where execution should be interrupted, the **`break`**, **`on`**, and **`trace`** commands can be used conditionally. In this way, breakpoints or actions are requested when a special condition is met. The WHEN clause is used to issue one of these commands conditionally. See "break" on page 199, "on" on page 239, and "trace" on page 269 for descriptions of the WHEN clause and how it is used.

Reviewing Requests

Once you have requested several breakpoints, actions, and monitors, you may need to review a list of requests in effect. The **query** command lists the requests. Use the **drop** command to drop requests that you no longer need, or use the **ignore** command to ignore requests temporarily at certain locations. Also, with the **disable** command, you can deactivate certain requests temporarily. Then you can use the **enable** command to reactivate them.

Arguments in the Debugger Formats

There are several argument types that are common to many commands. You should become familiar with these arguments in order to effectively use the commands that are described in Chapter 14, "Command Directory," on page 187. The following common arguments are explained here:

- SECTION-NAME and FUNCTION-NAME
- HOOK-TYPE
- EXPRESSION
- SCALAR-TYPE-EXPRESSION
- AGGREGATE-TYPE-EXPRESSION
- VALUE
- COUNT
- PTYPE
- %FMT

Arguments that can be used only with one command are explained with that command in Chapter 14, "Command Directory," on page 187.

SECTION-NAME and FUNCTION-NAME Arguments

Request breakpoints or actions in specific functions or program sections by using the FUNCTION-NAME or SECTION-NAME arguments.

SECTION-NAME (which must be enclosed in parentheses) is the section name for each source file as specified with the **sname** compiler option (or default). For example, the following **drop** command uses a SECTION-NAME argument to drop all requests for the **comp23** compilation:

```
drop (comp23) all
```

Section names cannot exceed seven characters in length.

Note: The terms section name and compilation are used interchangeably in this book. △

FUNCTION-NAME is the name of a function. For example, the following **break** command sets a breakpoint on line 17 of the **main** function:

```
break main 17
```

If extended name support is selected with the **auto extname** command, function names can be as long as 225 characters; otherwise, function names are limited to 8 characters.

If you omit a function name or section name, the debugger responds differently, depending on where you are in the program and on which argument you use:

- For *, calls, entry, and return (see Table 12.1 on page 136), if you do not specify a section name or function name, a breakpoint (or action) is requested at
  - all lines in the program (*)
  - calls by all functions (calls)
  - entries to all functions (entry)
  - returns from all functions (return).

  For example, because the following command does not specify a function name, breakpoint requests are issued for all entries to functions:

  ```
  break entry
  ```

  However, this command issues one breakpoint request for the entry to function **`suba`**:

  ```
  break suba entry
  ```

- If you specify a line number or line-number range by itself, the breakpoints or actions are requested only in the current function. The following two examples give identical results only if you are already in the function **`suba`**:

  ```
  break suba 3:6

  break 3:6
  ```

  Breakpoints will be requested at lines 3 through 6 in **`suba`** only.

HOOK-TYPE Argument

The HOOK-TYPE argument specifies places (or hooks) in a program in whichyou want execution to be interrupted.

What Is a Hook?

A hook is a location in a program where control of execution can be transferred to the debugger. If control is transferred from the executing program to the debugger, it happens before the code that is on that line is executed. Note, however, that certain lines of code, such as IF and FOR statements, have more than one hook.

Hooks are created at each source line in a SAS/C program at compile time (when the program is compiled with the **`debug`** option). In addition to these line hooks, the debugger also enables you to gain control at function calls, function entries, and function returns.

No hooks are generated for blank lines or for lines that contain only comments. Certain hooks are generated for code that is included via a **`#include`** file.

Rules for hook generation Here are some rules for hook generation:

- If there are multiple simple statements on a line (no transfer of control or function call), one hook is generated before the first statement.
- If there are statements on the line that can cause a transfer of control based on a condition, then a second hook is generated after the condition test. The following is an example of such a statement:

  ```
  if (a == b) i = 2;
  ```

  One hook is generated before the IF statement and a second before the assignment. If the condition evaluates to be true, the second hook is executed also. If the condition evaluates to be false, the second hook is not executed.

- A FOR statement generates up to three hooks. If the statement begins a line, there is one hook before the statement. Hooks are always generated before the test and increment portions.

  You can use hooks that are generated in a FOR statement by coding a FOR statement in the following manner:

  ```
  1    for (i = 0;
  2              i < 10;
  3                        i++)
  4        {
           .
           .
           .
           }
  ```

  In this example, the debugger stops at line 1 as it falls into the loop. Then, the debugger stops at line 2 before performing the test. After executing the body of the FOR statement, the debugger stops at line 3 (**i** is examined before it is incremented) and then at line 2.

- If braces appear by themselves on a line, hooks are generated at the braces. You can use this feature, for example, to to set a breakpoint on teh ending brace of an if-then-else series, which then enables you to perform some other action, such as examining variables.

Function calls Hook generation for function calls differs, depending on whether the function call spans lines. If the entire function call fits on a line, a function call is no different from most simple statements. A hook is produced on the line, as long as there is no other hook on that line (because another statement preceded the function call on the same line).

If a function call spans lines, one hook is always generated on the line in which the parenthesis occurs that closes the function call. On the other lines, hooks are generated only if code is generated. (Because a certain amount of optimization is performed even when the **debug** option is used, you cannot always determine which statements cause machine code to be generated.)

Specifying Hooks with the HOOK-TYPE Argument

As mentioned in the previous section, you can specify places (hooks) in a program in which you want to interrupt execution. You do this with the HOOK-TYPE argument. Table 12.1 on page 136 shows values that you use for the HOOK-TYPE argument.

Table 12.1 Values for the HOOK-TYPE Argument

Specifier	Location[1]
*	at every line-hook in every function or section (compiled with **debug**).
line-num	at a source line number *line-num* in the current function. (*line-num* is an integer.)
line-num1:line-num2	at all source lines between *line-num1* and *line-num2*, inclusive in the current function. (*line-num1* and *line-num2* are integers.)

Specifier	**Location**[1]
c{alls}	at all calls by function, and at all returns to the functions. The context in both cases is that of the calling function.
e{ntry}	at entry to all functions. The context is that of the called function. Automatic variables of the called function are not yet allocated and, therefore, cannot be examined. The values of formal parameters, however, can be examined.
r{eturn}	at all returns from the called functions. The context is that of the returning function.
FUNCTION-NAME/ (SECTION-NAME) *	at every hook in the specified function or section (compiled with the **debug** option).
FUNCTION-NAME/ (SECTION-NAME) *line-num*	at a source line number in the specified function or section. (*line-num* is an integer).
FUNCTION-NAME/ (SECTION-NAME) *line-num1:line-num2*	at all source lines between *line-num1* and *line-num2*, inclusive in the current function or section. (*line-num1* and *line-num2* are integers.)
FUNCTION-NAME/ (SECTION-NAME) c{alls}	at calls from the specified function, or at calls from all functions in the specified section. For each call, the debugger breaks twice: when the specified function calls and on return to the specified function. In both cases, the calling function determines the context.
FUNCTION-NAME/ (SECTION-NAME) e{ntry}	at entry to a called function or to all called functions in a section. Automatic variables of the called function are not yet allocated and, therefore, cannot be examined. The values of formal parameters, however, can be determined.
FUNCTION-NAME/ (SECTION-NAME) r{eturn}	at return from a called function or from all called functions in a section. The context is that of the returning function or functions.

1 Some source lines, such as lines that contain a FOR statement, may actually contain more than one hook, as described in the previous section.

Examples of Specifying Hooks

The following examples of the **break** command illustrate the use of the HOOK-TYPE argument:

break *
sets breakpoints at all line hooks; that is, at all program lines with executable code.

break 25
sets a breakpoint at line 25 of the function that you are in.

break 18:22
sets breakpoints at lines 18 through 22 of the function that you are in.

break calls
sets breakpoints at calls from all functions.

break entry
sets breakpoints at entry to all functions.

break return
sets breakpoints at return from all functions.

break suba *
sets breakpoints at all line hooks in the function called **suba**.

break main 15
sets a breakpoint at line 15 of the function named **main**.

break main 101:135
sets a breakpoint at each line in the function **main** between line 101 and line 135, inclusive.

break (sub1) *
sets a breakpoint at every line in the compilation called **sub1**.

break checkup c
sets breakpoints at calls from the function **checkup**. The context is that of **checkup**. For each call, there is a breakpoint at the call from **checkup** at the return to **checkup**.

EXPRESSION Argument

The EXPRESSION argument is used with several commands, including **assign**, **copy**, **dump**, **monitor**, **print**, **transfer**, **watch**, and **whatis**. In addition, this argument can be used as part of a WHEN clause to express a condition in commands, such as **break**, **trace**, and **on**.

The EXPRESSION argument can be certain valid C and C ++ expressions. An expression is a sequence of operators and operands that

- specifies computation of a value
- designates an object or a function
- produces side effects
- performs a combination of the above.

In other words, an expression is a construction consisting of one or more operators and operands, ranging from the simple to the complex. Variable names, function calls, constants, literals, array names and references, and structure references are all considered expressions. The SAS/C Debugger supports expressions such as the following examples:

```
i
i + j
i * (a + b)
!strcmp(s, d)
((struct xyz *) 0p12345678)->a.b
arr[i] .xyz->b
strlen(s) == strlen(d)
p == &x && q != &y
```

The expression's order of evaluation follows the operator's rules of precedence and associativity. Table 12.2 on page 139 provides a list of the operators that are supported in expressions and operator precedence.

You make an integral constant unsigned by specifying the **U** suffix. Similarly, use the **L** suffix to make an integral constant long, using **L** either alone, or combined with **U**.

Some restrictions exist on what you can substitute for the EXPRESSION argument with certain debugger commands. For example, what you substitute for EXPRESSION with the **`dump`** command must evaluate to a pointer, address, or array. Also, the resulting size for some commands (such as **`dump`**) depends on the type of EXPRESSION. For example, with the **`dump`** command, if the EXPRESSION is a pointer, then the size of the pointed-to object is dumped; if it is an array, the result is the size of one item of the array, and so on.

Constant pointers in an expression must be typed with leading **`0p`**. A value such as **`0p12345678`** behaves the same as **`((char *)0x12345678)`** in expressions. **`0p12345678`** is a short form for **`((char *) 0x12345678)`**.

Operators Supported in Expressions

Table 12.2 on page 139 lists the operators that you can use in expressions. They are grouped in order of highest to lowest precedence. The associativity is indicated by R-L (right-to-left) or L-R (left-to-right).

Table 12.2 Operators Supported in Expressions

Operators	Meaning	Associativity
[ ]	array element	L-R
( )	function call	L-R
.	structure/union member reference	L-R
->	structure/union member pointer	L-R
`sizeof`	size of an object	R-L
(type-name)	cast (type conversion)	R-L
~	bitwise negation	R-L
!	logical negation	R-L
-	unary minus	R-L
+	unary plus	R-L
&	address of	R-L
*	pointer	R-L
*	multiplication	R-L
/	division	L-R
%	modulus	L-R
-	subtraction	L-R
+	addition	L-R
<<	left shift	L-R
>>	right shift	L-R
<	less than	L-R
>	greater than	L-R
<=	less than or equal to	L-R

Operators	Meaning	Associativity
>=	greater than or equal to	L-R
==	equal to	L-R
!=	not equal to	L-R
¬	bitwise exclusive OR	L-R
|	bitwise OR	L-R
&&	logical AND	L-R
||	logical OR	L-R
::	scope resolution	L-R

For more information about these C operators, see a C language manual.

Functions That Can Be Used in Expressions

You can use the following functions in expressions:

- **memcmp**
- **strcmp**
- **strlen**

Casts

A cast is an operator (unary) that converts the value of its operand to a specified type. For commands that take expressions and for the WHEN clause, you can cast an expression or object identifier to a different type by explicitly specifying the type within parentheses. The debugger accepts the following casts:

(TAG *)
(**struct** TAG *)
(**union** TAG *)
(**enum** TAG *)
(basic-arithmetic-data-type *).

Zero or more `*` can be specified, as long as the result is semantically valid. In many cases, the TAG format for arguments is useful with these commands. Specifying an argument that is an address, such as ***(struct *tag***) ***address*** enables you to refer to storage that starts at any absolute address (ADDRESS) with the mapping of a structure, union, or enumeration of the type named by TAG. For example, the following refers to storage starting at the address in **0p23456789** with the mapping of **struct listall**:

```
*(struct listall *) 0p23456789
```

Note: The **struct**, **union**, or **enum** keywords can be dropped from the cast if no ambiguity is introduced. △

Structure and Union Members

You can access a member of a structure by specifying the pointer and the structure member name. For example, suppose **ptag** is declared to be a pointer to a structure named **TAG0** that has a member named **mem0**. You can access **mem0** as follows:

```
ptag->mem0
```

You can also access memory, using the mapping of a different structure, by casting the pointer to the appropriate pointer type. For example, **`((struct TAG1 *)ptab)`** is a pointer to structure **`TAG1`**. You can access **`mem1`** as follows:

```
((struct TAG1 *) ptab)->mem1
```

If you are working with absolute addresses, you can specify an absolute address with a cast to obtain a pointer to a structure. **`((struct TAG2 *)0p00123456)`** is a pointer to structure **`TAG2`**. A member **`mem2`** of the structure can be accessed as follows:

```
((struct TAG2 *)0p00123456)->mem2
```

Macros

Expressions can contain macros that are defined (with a #DEFINE statement) in your program. Macros can be specified in any arithmetic expression. At the time of expression evaluation, the debugger replaces any macros in the expression with their replacement text. The maximum length of the replacement text that is supported by the debugger is 1,536 characters. If the replacement text is another macro, further substitution takes place. The debugger is capable of processing only macros without arguments.

The debugger is sensitive to the case of the macro and distinguishes between upper- and lowercase macro names.

To access macros in your program from the debugger, you must compile your program with the **`dbgmacro`** option. See Chapter 4, "Compiler Options," on page 57. In addition, except for the debugger **`whatis`** command, in order to do macro substitution in an expression, you must set the debugger keyword **`cmacros`**. This keyword is set with the **`auto`** command. By default, **`cmacros`** is not set.

The **`whatis`** command is used to display the replacement text of a macro. With **`whatis`**, macro replacement is always done, regardless of the setting of the **`cmacros`** keyword. If you set the **`cmacros`** keyword, however, you also can use the **`print`** command to display the replacement text for a macro.

Debugger Variables

Debugger variables are used to represent the value that is contained in one of the 16 general purpose registers, one of the 4 floating-point registers, or the current instruction address. You can use debugger variables in any command that supports expressions; however, they cannot be modified. Table 12.3 on page 141 lists the debugger variables along with the type that is assumed by the debugger.

Table 12.3 Debugger Variables

Variable Name	Description	Type
$r0 to $r15	General purpose registers 0 through 15	char *
$R0 to $R15	Aliases for $r0 to $r15	
$f0, $f2, $f4, $f6	Floating-point registers 0, 2, 4, or 6	double

Variable Name	Description	Type
$F0, $F2, $F4, $F6	Aliases for $f0, $f2, $f4, and $f6	
$iad	Current instruction address ($iad is not case sensitive.)	char *

If the debugger is stopped at the calls or entry hook of a function call, the debugger variables contain the caller's values. However, if the debugger is stopped at the return hook, they contain the callee's values.

If your program returns an integral or pointer value, $r15 contains the return value. $f0 contains the return value if the program returns a floating-point value. In either case, the return address in the calling function is provided by $r14.

Summary of Types of Expressions

Table 12.4 on page 142 shows the types of expressions that you can use in the **`print`** command and other commands. See also the individual commands in Chapter 14, "Command Directory," on page 187 for any restrictions that exist in substitutions for the EXPRESSION argument.

Table 12.4 Types of Expressions That Can Be Used in print and Other Commands

Type		Description

1	arithmetic	For example, a variable (**`v`**) is declared:
		`long v;` **`unsigned long v;`**
		`long long v;` **`unsigned long long v;`**
		`int v;` **`unsigned int v;`**
		`short v;` **`unsigned short v;`**
		`char v;` **`unsigned char v;`**
		`double v;` **`enum TAG v;`**
		`float v;`

2	pointer	For example, a variable (**`v`**) is declared as a pointer to any type (**`T`**): **`T *v;`**

3	structure, union or class	A variable (**`v`**) is declared:
		`struct {...} v;`
		`union {...} v;`
		`class {...} v;`
		`struct TAG v;`
		`union TAG v;`
		`class TAG v;`

4	array	A variable (**`v`**) is an array of any type (**`T`**) declared: **`T v[CONSTANT];`** (**`CONSTANT`** is a decimal integer).
5	address	The value (**`v`**) is specified to the debugger as an absolute address (any integer constant) or is a variable name preceded by the & operator (**`&v`**) (function names are excluded).
6	enum constant	The value (**`v`**) is an enumeration constant.

Type		Description
7	bitfield	A variable (**`v`**) is a bitfield, for example: **`unsigned v : BITS;`** (where **`BITS`** is a decimal integer).
8	function	If **`V`** is a function, then **`*V`** is a function pointer.

Output Formats

When you issue the **`print`** command with only a single argument (EXPRESSION), the value of the expression that you specify is formatted according to its type (defined in Table 12.4 on page 142). The output formats are described in Chapter 14, "Command Directory," on page 187.

Note: The format that is natural for the value's type is always used.

SCALAR-TYPE-EXPRESSION Argument

SCALAR-TYPE-EXPRESSION is an expression whose type is arithmetic, pointer, or bitfield. For example, the following **`assign`** command assigns the address of **`int_variable`** to a pointer name **`int_ptr`**.

```
assign int_ptr = &int_variable
```

In this example, **`int_ptr`** is a SCALAR-TYPE-EXPRESSION argument and **`int_variable`** is a VALUE argument. See "VALUE Argument" on page 143 for more information.

AGGREGATE-TYPE-EXPRESSION Argument

AGGREGATE-TYPE-EXPRESSION is an expression of type structure or union. (It cannot be an array.) For example, the following **`assign`** command assigns values from a value list to a structure named **`numbers`**:

```
assign numbers = {1,2,3}
```

In this example, **`numbers`** is an AGGREGATE-TYPE-EXPRESSION argument, and **`{1,2,3}`** is a VALUE-LIST argument.

VALUE Argument

VALUE is an expression whose type is one of the following:

- constant
- address
- expression of scalar type
- enumeration constant
- array name.

You can assign a value to any scalar expression that is visible at the point at which you issue the **`assign`** command. To refer to an identifier in an active function that is different from the function that you are in, precede the identifier by the function name and a colon. As discussed in "Specifying Identifiers Outside the Current Function" on page 147, the general form of this argument is FUNCTION-NAME:IDENTIFIER.

COUNT Argument

With the **print** command, COUNT is an integer constant that specifies the number of items to be printed. For example, if **records** is a structure, the following **print** command prints the first five elements of the structure in the format of the element.

```
print records 5
```

You can also use the COUNT argument with the **dump** command to specify the number of bytes to dump. For example, if **ptr** is a pointer, the following **dump** command will dump the first ten bytes of storage beginning at the location pointed to by **ptr**.

```
dump ptr 10
```

PTYPE Argument

A PTYPE argument can be one of the following:

- □ a native C type
- □ a **structure**, **union**, or **enum** type
- □ a type created with a **typedef**
- □ a pointer to any of the above.

PTYPE can be specified for an EXPRESSION argument that belongs to type 1, 2, or 3 in Table 12.4 on page 142. You cannot use a PTYPE argument with the other types.

Native C types include

```
char          signed char         unsigned char
int           signed int          unsigned int
long          signed long         unsigned long
long long     signed long long    unsigned long long
short         signed short        unsigned short
double
float
```

When you specify a PTYPE, the EXPRESSION argument acquires the type indicated by PTYPE and is displayed using the rules for that type. If you do not specify a PTYPE, the default is the type that is associated with the expression in the program. For dereferenced absolute addresses, the default is **char**.

Note: One caution is necessary about structure types and **typedefs**. If you have a structure in your program and a **typedef** with the same type identifier, then using the identifier by itself (with no **struct** identifier) refers to the **typedef**, not the structure. For example, with a structure type of **buf** and a **typedef** of **buf**, the following command prints the storage that is referenced by **newvar** with the PTYPE of the **typedef buf**:

```
print newvar (buf)
```

△

Use **struct** to obtain the PTYPE of the structure **buf**, as in the following example:

```
print newvar (struct buf)
```

If you specify only a type identifier, the debugger searches for a **typedef** first. If the identifier is not found in the list of **typedefs**, the debugger searches the list of structures, unions, and enumerations next.

To find a description of a type, the debugger follows the normal C-scoping rules that are applicable to the place at which execution is halted. As with expressions and identifiers, you specify a PTYPE as a type that is defined within the scope of a function, FUNCTION-NAME, using the following syntax:

FUNCTION-NAME:TYPE

FUNCTION-NAME is any function in the calling sequence. TYPE is a **struct**, **union**, or **enum** type identifier, or a type that is defined with a **typedef**. For example, the following command prints the value of **x** using the type **XYZ**:

```
print x (sub1:XYZ)
```

XYZ is a type within the scope of **sub1**.

The following are more examples of the PTYPE argument. Some examples are acceptable; others (marked with a •) are unacceptable. Each example includes a declaration and a list of **print** commands and results.

Example 1

Declaration:

```
int arr[5] ; /* assume arr begins at 0p15880 */
```

Command	Result
p arr	**arr : 0p00015880**
· **p arr (int)**	LSCD148 Array variable cannot be displayed using a TYPE/cast.
· **p arr (XXX)**	LSCD148 Array variable cannot be displayed using a TYPE/cast.

Example 2

Declaration:

```
struct XXX int a; short b,c; ;
typedef int INTARR[2]
int x,y,z;  /* assume x begins at 0p10000 */
    /* assume x=10; y=20; z=30; */
```

Command	Result
p x	**x : 10 (0x0000000a)**
· **p x (INTARR)**	LSCD148 Casting to an array type (INTARR) is not allowed.
p x (int *)	**x : 0p0000000a**
· **p x (union XXX)**	LSCD148 Invalid TYPE/cast “XXX” specified.

Example 3

Declaration:

```
struct XXX {int a,b;} x={1,2};
struct YYY {int c,d,e,f;} y=3{,4,5,6};
```

Command	Result
`p x.a`	`x.a : 1 (0x00000001)`
`p x.a`	`x.b : 2 (0x00000002)`
`p x (int *)`	`x : 0p00000001`

`p y (XXX)`	`&y : 0p00015880`
	`y.a : 3 (0x00000003)`
	`y.b : 4 (0x00000004)`

Example 4

Declaration:

```
struct ZZZ  {
            int a;
            double d1;
            };
struct ZZZ yyy;
struct ZZZ *xp = &yyy;
/* assume  xp→a = 5 and xp→d1=3.24} */
```

Command	Result

`p *xp`	`&(*xp) : 0p00015880`
	`(*xp) .a : 5 (0x00000005)`
	`(*xp) .d1: 3.24`

`p xp→`	`xp→a : 5 (0x00 000005)`

Example 5

Declaration:

```
enum fruit apple, orange, pear, peach;
enum fruit f1, f2;
int i = 1, j = 5;
f1 = apple;
f2 = pear;
```

Command	Result
`p f1`	`f1:0 (apple)`
`p f2`	`f2:2 (pear)`
`p i (enum fruit)`	`i:1 (orange)`
`· p j (enum fruit)`	`j:5 (constant not in list for enum type fruit) ·`
`p peach`	`peach:3 (enum constant)`

%FMT Argument

The %FMT argument is any of the format specifiers that you can use with the **`sprintf`** function. They include the following:

`c`	single character
`d`	decimal signed integer
`e` or **`E`**	exponential floating point
`f`	fixed-decimal floating point
`g` or **`G`**	**`f`** format or **`e`** format
`ll`	long long
`o`	octal integer
`s`	character string
`u`	decimal unsigned integer
`x`	hexadecimal integer (lowercase)
`X`	hexadecimal integer (uppercase)

The format specifier must result in the item being formatted in 256 characters or fewer. Also, if the EXPRESSION argument in the **`print`** command contains a modulus operator `%`, you must escape the modulus with a backslash `\`; otherwise, the debugger interprets the modulus as a format specifier. The only exception to this rule is modulus operators that are inside parenthesized expressions.

Specifying Identifiers Outside the Current Function

For arguments of the types that are used in a WHEN clause or those accepted by commands that take expressions, you can specify a variable or a tag in a function in the calling sequence that is different from the current function.

To specify a function that is different from the function that you are in, use the following argument format:

FUNCTION-NAME:IDENTIFIER

FUNCTION-NAME is the name of a function in the calling sequence. The function must be active. (Use the **`where`** command [placeholder] to determine if a function is in the active calling sequence.)

For example, suppose your program has three functions: **sub1**, **sub2**, and **sub3**. **sub1** calls **sub2**, and **sub2** calls **sub3**. A variable named **check** is declared in **sub2** as **auto int check**. Assuming that **check** is visible at the point that **sub2** called **sub3**, you can print the value of **check** from **sub3** using the following command:

```
print sub2:check
```

The colon is part of the syntax.

To print the sum of two variables in different functions, **a** in **suba** and **b** in **subb** , use the following command:

```
print suba:a + subb:b
```

Note: You can also specify identifiers outside your current function by changing command scope. The **scope** command is used to change command scope. See "Using the Status Window" on page 20 for a discussion of command scope and Chapter 14, "Command Directory," on page 187 for information about the **scope** command. △

ANSI Differences

The SAS/C Debugger treats expressions differently from the ANSI Standard in the following ways:

pointer subtraction
: **p1→p2** is permitted even if **p1** and **p2** point to types that are not compatible, provided that objects pointed to have the same sizes.

relational/equality operators
: **p1 operator p2** is permitted, even if **p1** and **p2** point to types that are not compatible. **p operator i** is permitted, where **p** is of pointer type and **i** is of integral type and is positive.

sizeof expression
: **expression** is actually evaluated in the **sizeof** calculation.

Windowing Interface and Command Execution

This section describes **window** subcommands, how to issue commands, and how placeholders are used in debugger commands.

Window Operations and Window Subcommands

This section explains **window** commands that are issued outside the PROFILE and configuration files. See Chapter 3, "Debugger PROFILEs, Configuration Files, and EXECs," on page 41 for information about the **window** command in a PROFILE or configuration file.

The **window** command is used to perform a variety of window functions: clearing windows, closing windows, moving windows, opening windows, resizing windows, scrolling windows, and so on. Each of these functions is performed by a subcommand of the **window** command. For example, to use the **scroll source up** subcommand, you type **window scroll source up**; **scroll** is a subcommand of the **window** command.

On completing its execution of most window-oriented commands, the debugger does not move the cursor out of the window. Therefore, the following commands, which are issued when the debugger is running, are useful when they are assigned to PF keys:

Table 12.5 Useful Commands to Assign to PF Keys

Command	Description
autopop	automatically pops (unobscures) the window when updated.
border	specifies the characters used to form window borders.
clear*	clears the named window.
close*	closes the named window.
color*	changes the color, attributes, and intensity of various areas in a window.
context*	controls the amount of context information around the highlighted line in the Source window.
move*	moves the named window.
next	positions the top window at the bottom of the stack. The window below the window just moved becomes the top window.
off	switches the debugger to line mode.
on	switches the debugger to full-screen mode.
open*	opens the named window.
previous	positions the bottom window at the top of the stack and makes it the top window.
resize*	resizes (grows or shrinks) the window.
scroll*	scrolls windows (up and down, and left and right), as well as changes the scroll amount.
top*	positions the named window at the top of the stack. The physical cursor appears in this window at the place of the logical cursor. The window becomes the top window.
trace}*	controls the production of trace lines in the Log window.
zoom*	zooms the named window. If already zoomed, the window is unzoomed.

* These subcommands take a window name as a parameter.

The usual abbreviation conventions, which apply to all debugger commands, also apply to these subcommands. A complete description of each subcommand is provided in Chapter 14, "Command Directory," on page 187.

Methods of Issuing Commands

Commands can be issued by typing them in the Command window and pressing the ENTER key. However, this can be quite cumbersome for frequently used commands, for example, **`scroll up`**, **`scroll down`**, and **`close`**.

Window commands, like most other commands, can be assigned to PF keys. See "Using PF Keys" on page 27 for details about default PF key assignments and how they may be modified.

Placeholders in Commands

Many commands, including **window** subcommands and commands that take expressions as arguments, can be assigned to PF keys using a placeholder to indicate the position at which an argument is to be inserted. If a command contains a placeholder, then the position of the cursor is used to determine the value of the argument that is inserted in place of the placeholder.

Using Placeholders in Window Subcommands

Several of the **window** subcommands take WINDOW-NAME as a command argument. If you want to use the cursor to point to the window, a placeholder (syntax < >) is used in these subcommands instead of the window name.

The following example illustrates the use of placeholders with **window** subcommands. Most windows can be scrolled back using the following command:

```
window scroll WINDOW-NAME up
```

You could assign this command to a PF key as a text string, substituting the name of a window for WINDOW-NAME. Whenever you press the assigned PF key, the window would be scrolled up. However, you can also assign the following command to a PF key:

```
window scroll < > up
```

Pressing the PF key that is assigned to this command causes the window that the cursor is in to scroll up.

Placeholders in Commands That Take Expressions

You can also use placeholders with commands that take expressions, thus simplifying the execution of frequently used expression-handling commands. One or more placeholders, < >, can be specified in a command that is assigned to a PF key. When you issue this command by pressing the PF key, you can use the cursor to point to the expression that replaces all occurrences of the placeholder in the command.

For example, the **print** command can be submitted, like any other command, in the Command window. However, as **print** is one of the most commonly used commands to display expressions, you can assign **print** < > to a PF key. Moving the cursor to an expression and pressing that PF key would cause the debugger to replace the placeholder with the expression, and issue the **print** command.

Similarly, in an application that performs much text-string manipulation, you may frequently issue a **dump expression str** command. You can assign **dump < > str** to a PF key. Moving the cursor to an expression and pressing that PF key would cause the debugger to replace the placeholder with the expression, and dump the string.

Expression extraction Expressions occupying contiguous characters on one line in a window can be substituted for a placeholder in a command that is assigned to a PF key. The process of extracting an expression for substitution in a command is referred to as *expression extraction*. If the area of the window that contains the expression is unprotected, you can modify the expression before extraction.

The debugger uses an algorithm to identify the portion of the expression that replaces the placeholder. The rules for determining the extracted expression are described in the following section.

Extracting Expressions

The debugger uses an algorithm to extract expressions from the surrounding text. The debugger first does a preliminary backward scan until it encounters one of the following characters: a left parenthesis, a right parenthesis, a left bracket, or a character that is valid in an identifier. For debugger purposes, **$** is also considered to be a character that is valid in an identifier. (If the debugger is already on such a character, this preliminary scan is not needed.) If the backward scan cannot find one of these characters, there is no expression to be extracted. Processing then depends on the character that is encountered:

- If the character is a left parenthesis, the debugger scans forward through all text until it sees a matching right parenthesis that is considered part of the expression. Both parentheses are also included in the expression.
- If the character is a right parenthesis, the debugger scans backward through all text until it sees a matching left parenthesis that is considered part of the expression. Both parentheses are also included in the expression.
- If the character is not a parenthesis, it must be a left bracket or a character that is valid in an identifier. This position is marked as the tentative end.

  Note: If this character that is valid in an identifier is not part of an identifier, results may be unpredictable. △

  A backward scan now begins, either from the character in question (if it is valid in an identifier) or from the character that is before the character in question (if on a left bracket). The debugger scans over characters that are valid in identifiers, dot operators, and arrow operators. White space terminates the scan. If it encounters any right brackets in the backward scan, all text (including white space) up to the corresponding left bracket is automatically included, and the scan then resumes. Any right parenthesis that immediately precedes a dot or arrow has the result that all text up to the corresponding left parenthesis is automatically included. The position at which the scan terminates marks the start of the expression to be extracted.

  A forward scan is done next from the tentative end to obtain the actual end of the expression to be extracted. If the tentative end was a left bracket, the forward scan ends at the corresponding right bracket; otherwise, the rest of the identifier is scanned.

Table 12.6 on page 152 illustrates how the expression to be extracted depends on cursor position. When you read the table, assume the cursor is at the position shown in the expression in the first column. The third column shows the extracted text. The examples illustrate what would happen if the cursor were in one of several positions. In those cases, the cursor appears somewhere within the range indicated by the text shown in the second column of Table 12.6 on page 152.

Table 12.6 Placeholder Expression Extraction

Window Text	Cursor Position[1]	Extracted Text
`pntr[str.ind][5]->mem1.mem2`	`mem2`	`pntr[str.ind] [5]->mem1.mem2`
`pntr[str.ind][5]-> mem1.mem2`	`mem1`	`pntr[str.ind] [5]->mem1.`
`pntr[str.ind][ 5]->mem1.mem2`	`5]->`	`5`
`pntr[str.ind] [5]->mem1.mem2`	`[`	`pntr[str.ind][5]`
`pntr[str. ind][5]->mem1.mem2`	`ind]`	`str.ind`
`pntr[ str.ind][5]->mem1.mem2`	`str.`	`str`
`pntr [str.ind][5]->mem1.mem2`	`[`	`pntr[str.ind]`
`pntr[str.ind][5]->mem1.mem2`	`pntr`	`pntr`
`(a+b-c )`	`)`	`(a+b-c)`
`(a+b-c)`	`(`	`(a+b-c)`
`(p=5)-> aa.bb`	`aa.`	`(p+5)->aa`
`(*p). aa.bb`	`aa.`	`(*p).aa`

1 All of the ranges listed in the Cursor Position column include a leading blank space except for **`mem2`** and **`pntr`**. Placing your cursor in the leading blank returns the extracted text the same as placing your cursor within the rest of the range.

As shown by Table 12.6 on page 152, arbitrary expressions may be extracted by surrounding them with parentheses.

When the cursor is moved under the expression that is to replace the placeholder, and one of the methods described in “Methods of Issuing Commands” on page 149 is used to execute the command, the debugger extracts the expression and substitutes it for the placeholder in the command being issued.

The algorithm used by the debugger to extract expressions does not handle numeric constants; positioning the cursor on numeric constants may lead to unpredictable results. However, numeric constants can be specified as part of an expression enclosed in parentheses.

CHAPTER 13 Window Directory

List of Windows

This chapter describes the following windows that compose the windowing interface for the SAS/C Debugger:

Browse	browses text files and displays the output of the **`browse`** command.
Command	issues debugger commands.
Config	displays or changes window configuration settings.
Dump	dumps memory contents.
Find	searches for strings.
Help	provides debugger help information.
Keys	displays or changes PF key settings.
Log	displays the session log.
Message	displays debugger messages that do not request information.

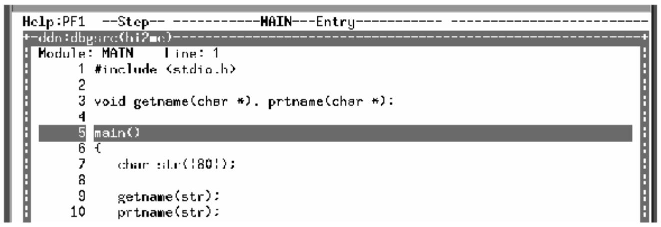
Popup	requests additional information.
Print	displays the value of an expression.
Register	displays register contents.
Source	displays source code.
Status	displays status information.
Termin	accepts operater input for terminal input requests.
Termout	displays terminal output.
Watch	displays changing values of expressions.

A summary of each of these windows is provided in the following sections. See Chapter 2, "The Windowing Interface," on page 11 for additional information about using windows.

Browse Window

The Browse window, shown in Figure 13.1 on page 154, is used to browse text files and to display the output of the **browse** command.

Figure 13.1 Browse window



```
Help:PF1  --Step-- -----------MAIN---Entry----------- ------------------------
+-ddn:dbgsrc(hi?me)-------------------------------------------------------------+
 Module: MAIN    Line: 1
       1 #include <stdio.h>
       2
       3 void getname(char *). prtname(char *):
       4
       5 main()
       6 {
       7    char str[80];
       8
       9    getname(str):
      10    prtname(str):
```

The Browse window has the following characteristics:

- □ The window border is optional.
- □ The top line contains the name of the displayed file and the next line contains the number of the top-most line that is displayed in the window.
- □ The lower portion of the window contains two areas that display the line number of the file that is displayed and the text of the file.
- □ The text area is 252 characters wide, only a portion of which is visible at one time.
- □ The maximum text file line length that is supported in full-screen mode is 252.
- □ The amount of memory that is used for browse buffers is controlled through the **window memory** command or the Config window. If the amount of memory for buffers is changed, any Browse windows that are opened after the change will have the new value.

Opening a Browse Window

The following command opens a Browse window:

```
window open browse
```

As many as six Browse windows can be open simultaneously.

You can also use the **`browse`** command to open up a Browse window. See the section “browse” on page 200 for more information.

Filename Syntax

If an explicit style precedes the filename that is specified in the **`File`**: field of the Browse window, the debugger uses the style that is specified. If not, the debugger assumes a **`tso:`** style filename under OS/390 and a **`cms:`** style filename under CMS. See SAS/C Library Reference, Volume 1 for more information about filename specifications.

Moving Around the Text File

Scrolling vertically by using the **`window scroll up`** and **`window scroll down`** commands moves the data in both the line-number and the text areas of the Browse window. Scrolling horizontally by using the **`window scroll left`** or **`window scroll right`** commands moves only the text and not the line numbers.

Scrolling is a simple and fast way to move to lines that are close to the lines that are currently displayed. A faster way to move to distant parts of a text file is to type the line number in the **`Line`**: field of the Browse window.

You can see a different text file in the Browse window by typing its filename in the **`File`**: field.

If you type either an invalid filename in the **`File`**: field or an invalid line number in the **`Line`**: field, a pop-up window opens that enables you to correct the invalid input.

The **`window find`** command is also supported in the Browse window.

Order of Processing

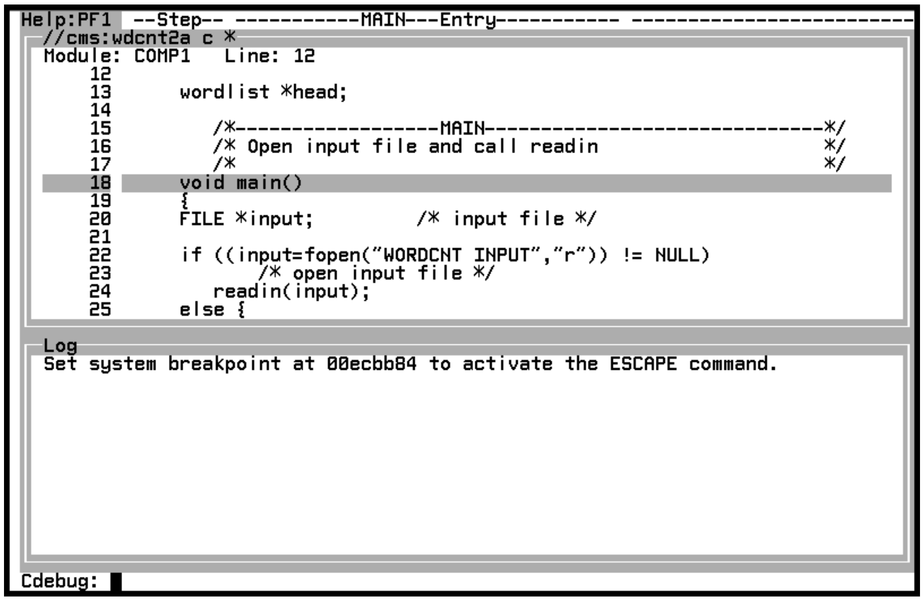
If you specify input in more than one field of the Browse window, the data is processed in the following order:

1 **`File:`** field

2 **`Line:`** field.

Command Window

Display 13.1 Command Window



```
Help:PF1  --Step-- -----------MAIN---Entry----------- ------------------------
  //cms:wdcnt2a c *
 Module: COMP1   Line: 12
       12
       13      wordlist *head;
       14
       15         /*-----------------MAIN-----------------------------*/
       16         /* Open input file and call readin                 */
       17         /*                                                  */
       18      void main()
       19      {
       20      FILE *input;        /* input file */
       21
       22      if ((input=fopen("WORDCNT INPUT","r")) != NULL)
       23            /* open input file */
       24         readin(input);
       25      else {

  Log
 Set system breakpoint at 00ecbb84 to activate the ESCAPE command.

Cdebug: █
```

DESCRIPTION

The Command window issues debugger commands. It is one of the four primary windows. The other primary windows are Status, Source, and Log. In Display 13.1 on page 156 the Command window is identified by the **`Cdebug:`** prompt on the bottom line. Normally the Command window is displayed as shown, without a border and on the bottom line of the display. Like all the primary windows, the Command window is always open, even when it is obscured by an overlying window. Therefore, for purposes of the **`window`** command, it is a class 1 window. See "window" on page 280.

Issuing commands The Command window has two fields: a protected field that is used to display the **`Cdebug:`** prompt and a nonprotected command entry field. Debugger commands are typed into the command entry field, which starts immediately to the right of the **`Cdebug:`** prompt. You issue the command when you press the ENTER key. If a command is too long for the window, you can type it into the Log window by using a backslash for continuation.

Recalling commands As you issue commands, they are maintained by the debugger in a circular list. However, issuing the same command more than one time in succession results in only one copy of the command being maintained in the list. You can cycle through the list by using the **`window scroll < > up`** (PF19) and **`window scroll < > down`** (PF20) commands. You can recall previously issued commands by pressing the ENTER key. You can change recalled commands by typing over them before they are issued.

The previously issued commands are accessed in the order they were issued; as the list fills up, the oldest commands are deleted. You can use the **`window clear`** command to clear this list. The number of commands maintained in the circular list depends on the amount of memory that is allocated for the Command window. (The amount of memory that is allocated for window buffers cannot be changed during a session.) Each command requires one byte of storage for each character in the command plus three bytes for overhead information. You can specify the amount of memory that is used for this list with the **`window memory`** command.

ADDITIONAL DISCUSSION
“Using the Command Window” on page 19

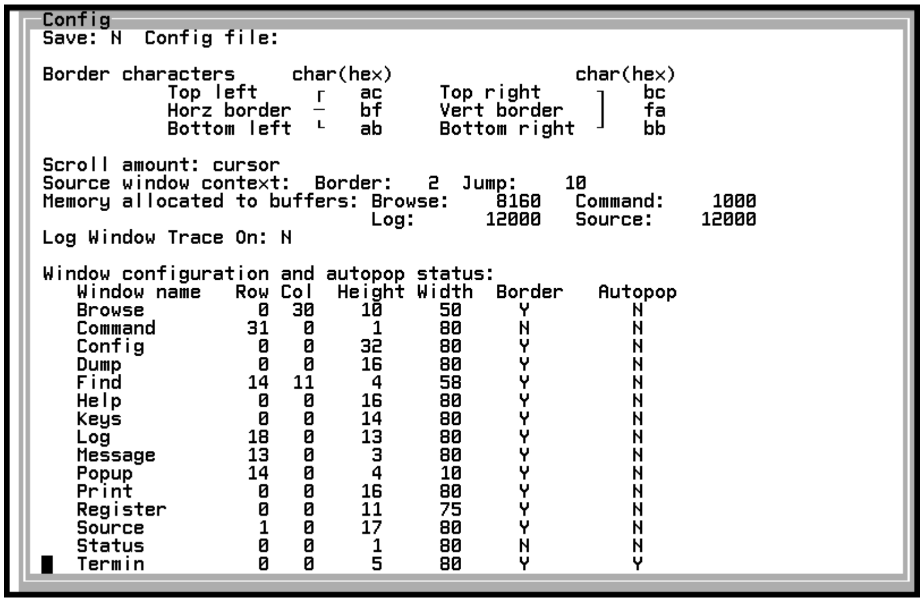
SEE ALSO
The **`window`** command for the following:

- □ “Log Window” on page 167
- □ “Source Window” on page 173
- □ “Status Window” on page 177

Config Window

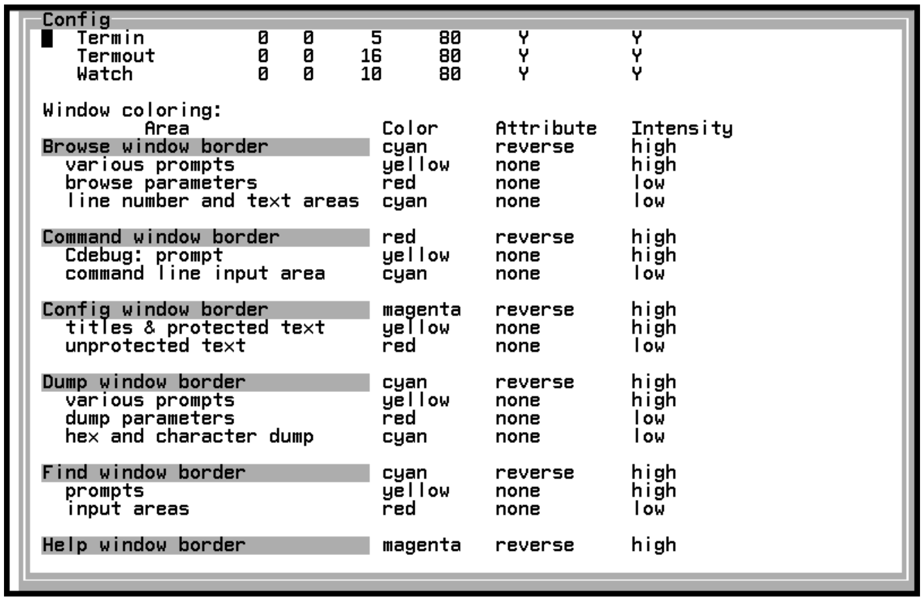
The following displays show the Config window. After you open the window, use the **`window scroll < > up`** (PF19) and **`window scroll < > down`** (PF20) commands to display the entire Config window.

Display 13.2 Config Window, View 1



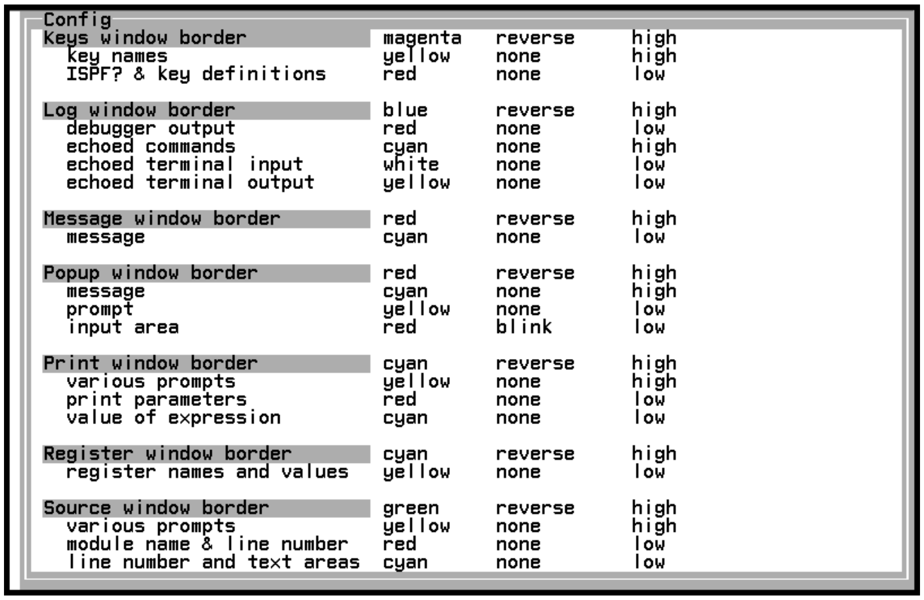
```
Config
Save: N  Config file:

Border characters     char(hex)                char(hex)
           Top left     ┌   ac     Top right    ┐   bc
           Horz border  ─   bf     Vert border  │   fa
           Bottom left  └   ab     Bottom right ┘   bb

Scroll amount: cursor
Source window context:  Border:   2  Jump:    10
Memory allocated to buffers: Browse:    8160   Command:    1000
                             Log:      12000   Source:    12000
Log Window Trace On: N

Window configuration and autopop status:
   Window name   Row Col  Height Width  Border   Autopop
   Browse          0  30    10     50     Y         N
   Command        31   0     1     80     N         N
   Config          0   0    32     80     Y         N
   Dump            0   0    16     80     Y         N
   Find           14  11     4     58     Y         N
   Help            0   0    16     80     Y         N
   Keys            0   0    14     80     Y         N
   Log            18   0    13     80     Y         N
   Message        13   0     3     80     Y         N
   Popup          14   0     4     10     Y         N
   Print           0   0    16     80     Y         N
   Register        0   0    11     75     Y         N
   Source          1   0    17     80     Y         N
   Status          0   0     1     80     N         N
█  Termin          0   0     5     80     Y         Y
```

Display 13.3 Config Window, View 2



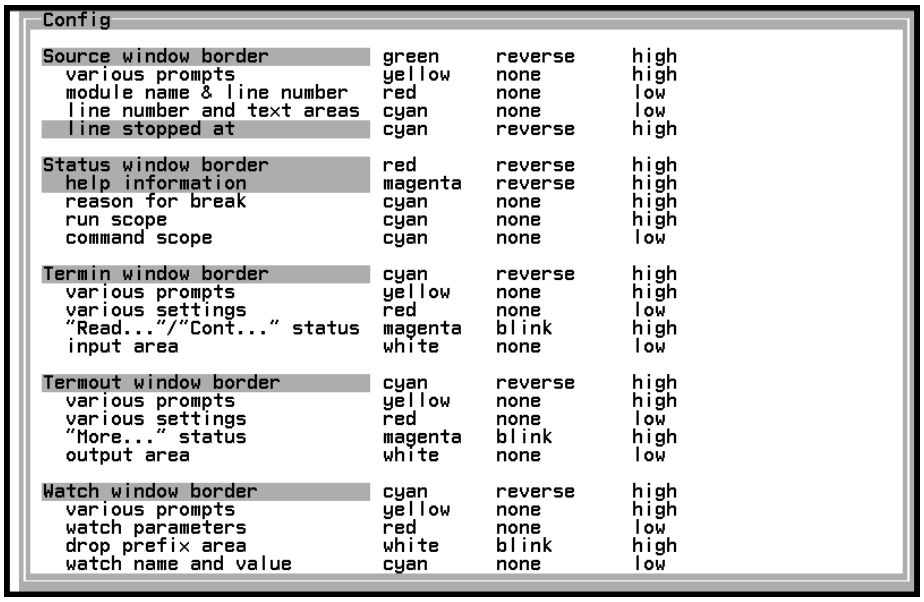
```
Config
█  Termin          0   0    5     80      Y          Y
   Termout         0   0   16     80      Y          Y
   Watch           0   0   10     80      Y          Y

Window coloring:
         Area                  Color     Attribute   Intensity
Browse window border           cyan      reverse     high
  various prompts              yellow    none        high
  browse parameters            red       none        low
  line number and text areas   cyan      none        low

Command window border          red       reverse     high
  Cdebug: prompt               yellow    none        high
  command line input area      cyan      none        low

Config window border           magenta   reverse     high
  titles & protected text      yellow    none        high
  unprotected text             red       none        low

Dump window border             cyan      reverse     high
  various prompts              yellow    none        high
  dump parameters              red       none        low
  hex and character dump       cyan      none        low

Find window border             cyan      reverse     high
  prompts                      yellow    none        high
  input areas                  red       none        low

Help window border             magenta   reverse     high
```

Display 13.4 Config Window, View 3



```
Config
Keys window border             magenta   reverse     high
  key names                    yellow    none        high
  ISPF? & key definitions      red       none        low

Log window border              blue      reverse     high
  debugger output              red       none        low
  echoed commands              cyan      none        high
  echoed terminal input        white     none        low
  echoed terminal output       yellow    none        low

Message window border          red       reverse     high
  message                      cyan      none        low

Popup window border            red       reverse     high
  message                      cyan      none        high
  prompt                       yellow    none        low
  input area                   red       blink       low

Print window border            cyan      reverse     high
  various prompts              yellow    none        high
  print parameters             red       none        low
  value of expression          cyan      none        low

Register window border         cyan      reverse     high
  register names and values    yellow    none        low

Source window border           green     reverse     high
  various prompts              yellow    none        high
  module name & line number    red       none        low
  line number and text areas   cyan      none        low
```

Display 13.5 Config Window, View 4



```
Config

Source window border          green     reverse     high
  various prompts             yellow    none        high
  module name & line number   red       none        low
  line number and text areas  cyan      none        low
  line stopped at             cyan      reverse     high

Status window border          red       reverse     high
  help information            magenta   reverse     high
  reason for break            cyan      none        high
  run scope                   cyan      none        high
  command scope               cyan      none        low

Termin window border          cyan      reverse     high
  various prompts             yellow    none        high
  various settings            red       none        low
  "Read..."/"Cont..." status  magenta   blink       high
  input area                  white     none        low

Termout window border         cyan      reverse     high
  various prompts             yellow    none        high
  various settings            red       none        low
  "More..." status            magenta   blink       high
  output area                 white     none        low

Watch window border           cyan      reverse     high
  various prompts             yellow    none        high
  watch parameters            red       none        low
  drop prefix area            white     blink       high
  watch name and value        cyan      none        low
```

DESCRIPTION

The Config window customizes the configuration of the windowing interface. You can also use the **`window`** command to customize these features; however, the Config window provides a more intuitive and easier method of making configuration changes.

Display 13.2 on page 157 shows the portion of the Config window that is visible when you first open the window. You can perform various functions by using the Config window.

Saving a configuration The top line of the window indicates your current configuration file and can be used to save the configuration. You can type any valid configuration filename following the **`Config file`**: prompt and then save the configuration to that file by typing a Y at the **`Save:`** prompt.

Changing border characters The Config window contains six fields that are used to select border characters: **`Top left`**, **`Top right`**, **`Horz border`**, **`Vert border`**, **`Bottom left`**, and **`Bottom right`**. These fields are shown with their default characters in Display 13.2 on page 157. You can change any of the border characters by typing in a new character or its hexadecimal value according to the EBCDIC code sequence.

All windows with borders use the same border characters. See Selecting window, size, position, and borders for details.

Setting scroll amount **`The Scroll amount:`** field can be used to specify the scroll distance. Valid values for this field are as follows:

`cursor`
scrolls to the position of the cursor

`half`
scrolls one half page.

`max`
scrolls to the maximum amount. (Max is a temporary setting that can be used for the duration of your current session. You cannot specify **`window scroll amount max`** in a configuration file.)

page
scrolls one page.

The size of a page is determined by the size of the window that you are scrolling. By default, **Scroll amount**: is set to cursor.

Setting Source window context The Source window always attempts to keep a number of lines of contextual information to surround the current (highlighted) line in that window. The Config window is used to control the number of contextual lines the debugger attempts to maintain and the number of lines the debugger is allowed to jump without centering the current line in the Source window.

The fields that are used to control Source window context are shown in Display 13.2 on page 157. The **Border:** field is used to enter the minimum number of lines that you would like to have displayed above and below the current line. If possible, the debugger maintains this minimum amount of contextual information within the Source window.

The **Jump:** field specifies the number of lines that can be jumped before the current line is centered in the Source window. Centering occurs whenever the next line to be executed is at least the number of lines that are specified away from the current line.

Allocating memory for window buffers The Command, Log, and Source windows each require buffer areas in memory. You specify the amount of memory to be allocated by typing the number of bytes at the **Command:**, **Log:**, or **Source:** prompts as shown in Display 13.2 on page 157. Default and minimum allocation sizes are provided in Table 14.15 on page 291.

Selecting Log window trace line As shown in the **Log Window Trace On:** field of Display 13.2 on page 157, trace lines are not displayed in the Log window by default. However, typing a **Y** in this field selects trace lines. If they are active, trace lines are displayed in the following cases:

- □ each time the debugger gives you control
- □ at the **n** – 1 hooks at which the debugger does not give you control, and in the **step n** or **continue n** case
- □ when an **on** command is executed, provided that **auto id** is active
- □ when a **monitor** request occurs.

Selecting window size, position, and borders The **Window configuration and autopop status:** fields, shown in Display 13.2 on page 157, are used to select window size, position, and borders for each of the 15 windows. Each window has the following fields:

Window name
identifies the window.

Row
controls the starting row from the top of the window. Rows start with **0** and are numbered from the top to the bottom of the display.

Col
controls the starting column from the left side of the window. Columns start with **0** and are numbered from the left to right sides of the display.

Height
controls window height.

Width
controls window width.

Border
determines whether the window is displayed with or without a border.

`Autopop`
: selects autopop status. If the status is set to **`Y`**, the window automatically becomes the top window whenever output is sent to the window and it is at least partially obscured by another window.

Changing window size, position, or border settings during a session has different effects depending on the type of window that is changed. The Dump and Print windows (class 4 windows, as described in “window” on page 280) that are already opened are not changed; however, any new appearances of these windows reflect the new settings.

All of the other windows (window classes 1 through 3 that are described in “window” on page 280) are automatically closed and reopened whenever window size, position, or border settings are changed. If a window is zoomed, it is zoomed when it is reopened.

Displaying borders requires additional display overhead on the part of the debugger and could result in exceeding the display limitations. These limitations are described in “Number of Open Windows” on page 37. If you add borders to the following (class 2 and 3) windows and the display limitations are exceeded, then the reopen will fail:

- Config
- Help
- Keys
- Message
- Popup
- Register
- Termin
- Watch

The following (class 1) windows are always reopened, even if the display limitations are exceeded:

- Command
- Log
- Source
- Status

However, if the display limitations are exceeded, these class 1 windows are opened without borders.

You should not use the configuration settings to temporarily move or resize a window. It is more effective to use the PF2 and PF14 keys, which have been assigned to the **`window move < >`** and **`window resize < >`** commands respectively. Using the Config window can result in moving the information in the window, or in placing the moved or resized window on top of the Config window.

Selecting window color, attribute, and intensity Each of the 15 windows comprises a number of areas with display characteristics that can be controlled by the Config window. The number of areas varies from one to five depending on the window. Display 13.2 on page 157 through Display 13.5 on page 159 show the areas that can be controlled for each of the 15 windows.

For each window area, you can set the following window coloring characteristics:

`Color`
: selects area color.

`Attribute`
: selects the display attribute for the area.

`Intensity`
: selects either high or low intensity.

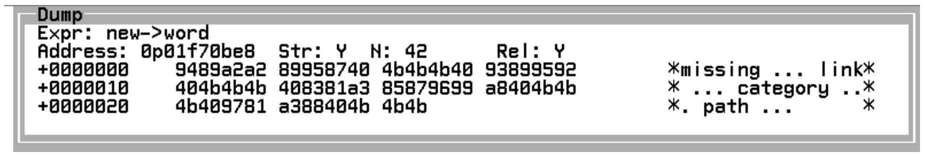
Each of these selections is limited by the capabilities of your terminal. When a selection is made, the characteristic is displayed in the area field of the Config window, enabling you to easily review the display characteristics of the entire window while the Config window is displayed.

ADDITIONAL DISCUSSION
"Changing the Window Configuration" on page 26

SEE ALSO
The **config** and **window** commands for "Keys Window" on page 166.

Dump Window

Display 13.6 Dump Window



```
Dump
Expr: new->word
Address: 0p01f70be8  Str: Y  N: 42      Rel: Y
+0000000    9489a2a2 89958740 4b4b4b40 93899592       *missing ... link*
+0000010    404b4b4b 408381a3 85879699 a8404b4b       * ... category ..*
+0000020    4b409781 a388404b 4b4b                    *. path ...      *
```

DESCRIPTION
The Dump window displays a dump of memory in both character and hexadecimal format. Output is directed from the **dump** command to the Dump window in much the same way as output from the **print** command is directed to the Print window. The redirect (**>** and **>>**) command prefixes are used with the **dump** command to direct the output from a memory dump to either a new or existing Dump window. See "Directing Commands to a Window" on page 26 for information about the **>** and **>>** command prefixes.

The Dump window provides the following information:

- □ The left side of the dump area provides the address of the first byte in each line. This address will be either absolute or relative. Type your preference at the **Rel:** field in the second line of the window. If you select relative addressing, the offset from the first byte dumped is displayed at the beginning of each line. The absolute address of the first byte in a dump is always displayed following the **Address:**field.
- □ The middle portion of the dump area provides a hexadecimal representation of the contents of memory. Each byte is represented by two hexadecimal characters. A total of either 8 or 16 bytes is represented on each line depending on the width of the Dump window.
- □ The right side of the dump area shows the printable EBCDIC characters that are contained in the dump.

Unprotected fields Any time a Dump window is displayed, you can use the following unprotected fields to alter the contents of the dump field:

Expr:
can be any expression within your current run scope that points to a memory location. This location identifies the first byte in the dump.

`Address:`
can be any valid hexadecimal address that is specified in **`0p`** format. This address corresponds to the first byte of the dump.

`Str:`
can be either **`Y`** (yes) to indicate a string type dump or **`N`** (no) to dump characters that are not in a string. If **`Y`** is selected, the number of bytes that are typed at the **`N:`** prompt is ignored. After a request to dump a string has been processed, the value of **`N:`** is updated to reflect the number of bytes that are in the string.

`N:`
can be any number of bytes to be dumped starting with the byte that is specified by the **`Address:`** field. **`Str:`** should be set to **`N`** (no) when you are dumping a known number of bytes.

`Rel:`
can be either a **`Y`** (yes) to select relative addressing or **`N`** (no) to select absolute addressing. The address area on the left side of the Dump window changes to reflect the type of addressing that is selected. If the area that follows the prompt is erased or blank, the type of dump is determined by the **`dumpabs`** setting of the **`auto`** command.

Issue a **`window scroll`** command or press the ENTER key without modifying any of these fields to refresh the display in the Dump window.

Typing invalid input in one or more of these fields causes the dump to fail; the Message window pops up with the reason for the failure, the dump area is blank, and the invalid input remains in the input fields.

The **`Expr:`** and **`Address:`** fields work together. If input is typed at both prompts, the expression that is typed at the **`Expr:`** prompt takes precedence and the address that is typed at the **`Address:`** prompt is ignored. However, if **`Expr:`** is not modified, and a new address is provided that follows the **`Address:`** prompt, the address is also displayed following the **`Expr:`** prompt when you press the ENTER key.

`Window size` The first time a Dump window is requested, the window height is based on the number of bytes being dumped. The window is opened with as many rows as needed to display the number of bytes specified, provided the height does not exceed the maximum for this window. The maximum is the height specified for the Dump window in a configuration file, or half the terminal height by default.

The width that is used is not dependent on the number of bytes that are dumped; it is always the Dump window width that is specified in your configuration file, or the terminal width by default. However, window width does determine the number of bytes that are dumped on each line. This number is always a power of two.

When a Dump window is reused, it is not automatically resized; however, it can be resized after the dump.

If you dump a large number of bytes, the dimensions may permit only a portion of the memory that is being dumped to be visible, but you can view the rest by scrolling the window.

ADDITIONAL DISCUSSION

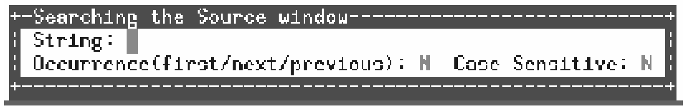
- □ "Using the Dump Window" on page 34
- □ "Directing Commands to a Window" on page 26

SEE ALSO

The **`dump`** and **`print`** commands for "Print Window" on page 171.

Find Window

Figure 13.2 Find Window



```
+-Searching the Source window--------------------------+
| String:                                              |
| Occurrence(first/next/previous): N  Case Sensitive: N |
+------------------------------------------------------+
```

DESCRIPTION

The find window searches for strings. When you issue the **window find** command, the Find window, illustrated in Figure 13.2 on page 164, is opened. If you have not changed your PF key assignments, you can also use PF17, which is assigned to the **window find < >** command, to open the Find window. Once the Find window is open, you can type the string to be searched for and the occurrence that you want: **f** (first), **n** (next), or **p** (previous). The search is started when you press the ENTER key or any PF key

If **n** or **p** is specified in the **Occurrence:** field, the search begins from the current position of the logical cursor. If this is the first time that the **window find** command is being executed in this window, or if the command was last executed in a different window, occurrence defaults to **P** for the Log window and **n** for the Source and Browse windows.

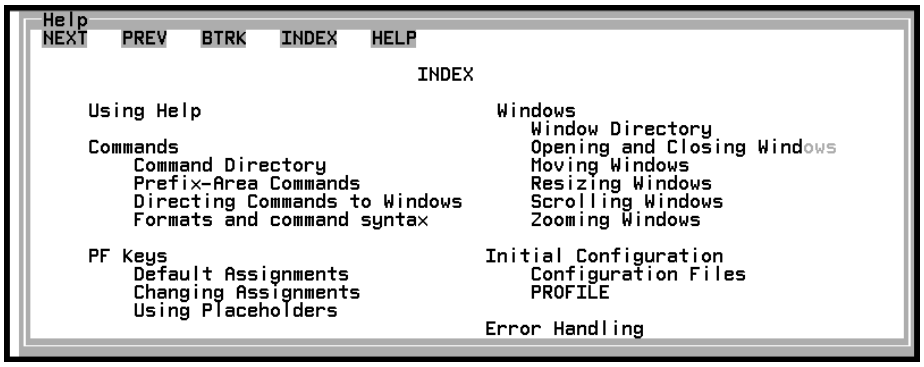
The search string may contain embedded blanks. By default, searches are not case sensitive.

In the current implementation, subsequent occurrences may be found by twice pressing the PF key that is assigned to the **window find < >** command (PF17 by default). Pressing the key the first time opens the Find window with the search string that was used the last time, and an appropriate occurrence parameter. Pressing the key the second time causes the debugger to accept the contents of the window and begin searching.

To abort a search after the Find window has been opened, erase the text of the string.

Help Window

Display 13.7 Help Window



```
Help
NEXT   PREV   BTRK   INDEX   HELP

                               INDEX

    Using Help                          Windows
                                           Window Directory
    Commands                               Opening and Closing Windows
        Command Directory                  Moving Windows
        Prefix-Area Commands               Resizing Windows
        Directing Commands to Windows      Scrolling Windows
        Formats and command syntax         Zooming Windows

    PF Keys                             Initial Configuration
        Default Assignments                Configuration Files
        Changing Assignments               PROFILE
        Using Placeholders
                                        Error Handling
```

DESCRIPTION

The Help window is used to access the debugger's help system. You can use the **`window open`** command to open the Help window; however, the **`help`** command provides the most efficient method of opening the window. By default, the **`help < >`** command is assigned to the PF1 key, which provides context-sensitive help that is based on cursor position.

Hypertext help system The help system is composed of a series of cards that are displayed in the Help window and linked together by logical concepts. You can move between cards by selecting the link that connects your current card with another card to which it is logically related. To select a link and move to another card, move the cursor to the link and press the ENTER key.

The initial card displayed in the Help window depends on which method you used to access the help system. The first method, issuing the **`help`** command from the Command window, displays the help system INDEX as shown above. Other methods of accessing the help system result in different initial cards. For example, pressing the PF1 key while the cursor is in the Command window displays the card for the Command window as the initial card in the Help window.

Using hypertext links Regardless of the method that you use to access the help system, the initial card displayed in the Help window always contains a series of five navigation links that are displayed on the top line. Navigation links are used to rapidly move through the help system following a predefined path or to backtrack through the cards you have previously displayed. The function of these navigation links is as follows:

`NEXT`
causes the next card in a predefined series of cards to be displayed.

`PREV`
causes the previous card in a predefined series of cards to be displayed.

`BTRK`
enables you to backtrack and review cards that you have already viewed.

`INDEX`
takes you to the help system INDEX card.

`HELP`
takes you to the first card in this series of cards providing information about the help system.

ADDITIONAL DISCUSSION

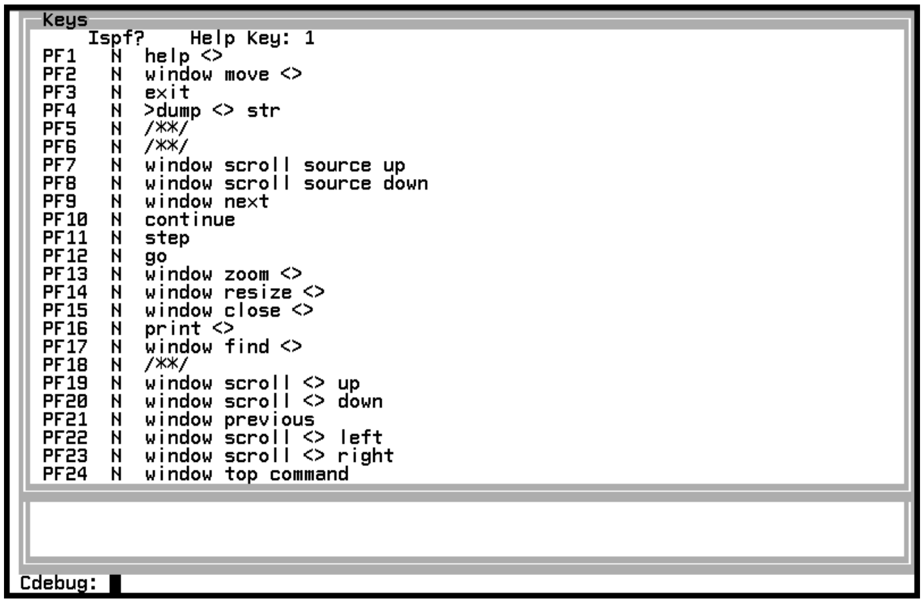
"Using the Help Window" on page 16

SEE ALSO

"help" on page 224

Keys Window

Display 13.8 Keys Window



```
 Keys
     Ispf?    Help Key: 1
 PF1   N  help <>
 PF2   N  window move <>
 PF3   N  exit
 PF4   N  >dump <> str
 PF5   N  /**/
 PF6   N  /**/
 PF7   N  window scroll source up
 PF8   N  window scroll source down
 PF9   N  window next
 PF10  N  continue
 PF11  N  step
 PF12  N  go
 PF13  N  window zoom <>
 PF14  N  window resize <>
 PF15  N  window close <>
 PF16  N  print <>
 PF17  N  window find <>
 PF18  N  /**/
 PF19  N  window scroll <> up
 PF20  N  window scroll <> down
 PF21  N  window previous
 PF22  N  window scroll <> left
 PF23  N  window scroll <> right
 PF24  N  window top command

Cdebug:
```

DESCRIPTION

The Keys window is used to control PF key assignments. As shown in Display 13.8 on page 166 , the window displays the debugger commands that are to each of the 24 PF keys. (Your terminal may not have all 24 keys.) The following fields are used to change default PF key assignments:

`Help Key:`

is used to assign the help key. By default, this field contains a 1, which assigns the **`help < >`** command to the PF1 key. To specify another PF key as the help key, type the PF key number in this field. The PF key that is assigned as the help key is always displayed in the left field of the Status window.

Typing a 0 in the Help Key field, or leaving it blank, indicates that you do not want a PF key assigned to the **`help < >`** command. This gives the PF key its default assignment and unprotects the field so that you can type a new debugger command.

key name area

is a protected area that identifies each of the 24 PF keys. Some terminals may not have all 24 keys. See your terminal documentation for more information.

`Ispf?`

is located between the key name and the key definition fields. The **`Ispf?`** field is used to specify that a particular PF key be handled by ISPF instead of the debugger. When set to **`Y`**, the key is handled by ISPF; **`N`** is the default, causing the key to be handled by the debugger.

To select **`Y`**, use the ISPF interface to invoke the debugger.

key definition area

is used to assign a debugger command to a PF key. Each PF key has a key definition field that contains the debugger command assignment for that key. For example, the key definition field for the PF2 key shows the **`window move < >`** command. Command assignments are made by typing over the default command. The only exception is the key definition field for the help key, which is assigned to PF1 by default. Before you can change the command that is assigned to the PF1 key, you must use the **`Help Key`**: field to select another help key. To return a PF key to its default assignment, erase the key definition field and press ENTER.

Any debugger command that is valid in full-screen mode may be assigned to a PF key. The symbols in the key definition field indicate that no debugger command has been assigned. The placeholder symbols, **`< >`**, are used to indicate that the name of the window in which the cursor is located is to be inserted at that point in the command.

ADDITIONAL DISCUSSION

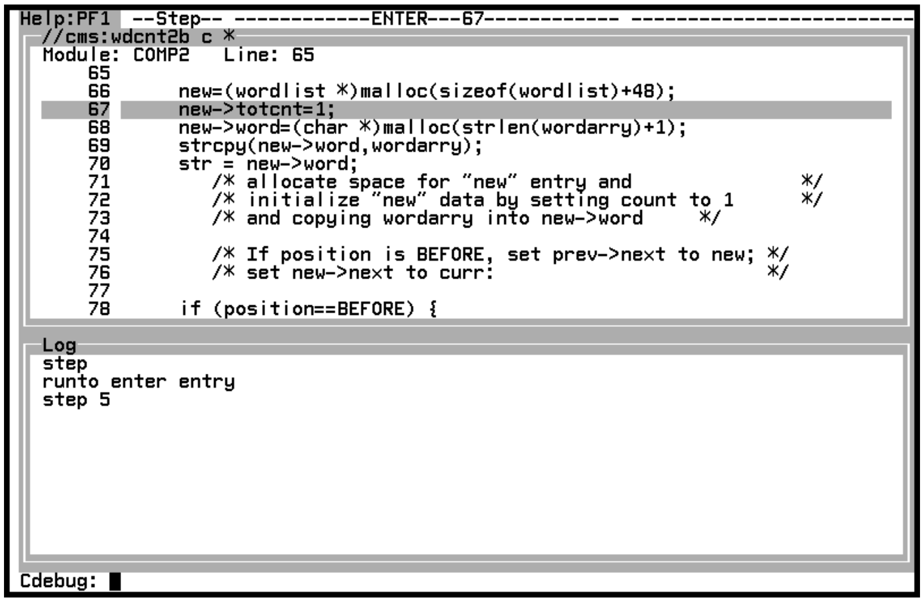
- □ "Using PF Keys" on page 27
- □ "Placeholders in Commands" on page 150

SEE ALSO

"keys" on page 230

Log Window

Display 13.9 Log Window



```
Help:PF1  --Step-- ------------ENTER---67------------ ------------------------
 //cms:wdcnt2b c *
 Module: COMP2   Line: 65
     65
     66      new=(wordlist *)malloc(sizeof(wordlist)+48);
     67      new->totcnt=1;
     68      new->word=(char *)malloc(strlen(wordarry)+1);
     69      strcpy(new->word,wordarry);
     70      str = new->word;
     71         /* allocate space for "new" entry and             */
     72         /* initialize "new" data by setting count to 1     */
     73         /* and copying wordarry into new->word     */
     74
     75         /* If position is BEFORE, set prev->next to new; */
     76         /* set new->next to curr:                         */
     77
     78      if (position==BEFORE) {
 Log
 step
 runto enter entry
 step 5

Cdebug:
```

DESCRIPTION

The Log window displays the session log information. It is one of the four primary windows. The other primary windows are Status, Source, and Command. In

Display 13.9 on page 167 the Log window is identified by its name in the upper-left border.

Like all the primary windows, the Log window is always open, even when it is obscured by an overlying window. Therefore, for purposes of the **`window`** command, it is a class 1 window. See “window” on page 280.

Viewing information in the Log window The Log window displays a log of your debugger session. Commands that are issued from the Command window, or the Log window itself, are displayed in the Log window. It is also used to display the output from commands such as **`print`** and **`dump`** and some error messages. The **`window scroll up`** and **`window scroll down`** commands can be used to view log lines that are not currently visible in the window. (By default, the PF19 and PF20 keys are dedicated to **`window scroll up`** and**`window scroll down`**, respectively.)

The length of each log line, with one exception, is determined by the **`auto linesize`** option. Long lines can be viewed by scrolling the window left and right using the **`window scroll left`** and **`window scroll right`** commands. (The PF22 and PF23 keys are dedicated, by default, to **`window scroll left`** and **`window scroll right`**, respectively.) The exception is output from the **`dump`** command, which is tailored to the size of the Log window.

Clearing the Log window and changing buffer size The **`window clear`** and **`window memory`** commands can also be used to clear the Log window and specify the amount of memory to be used for the session log. Like commands that are issued in the Command window, log lines are maintained in a first in, first out buffer. As the buffer fills up the older lines are deleted. You can increase the size of this buffer by using the **`window memory`** command in your configuration file, which increases the size of your session log. The amount of memory allocated for window buffers cannot be changed during a session.

Issuing debugger commands from the Log window You can also issue commands from the Log window by typing over a portion of a previously typed command that is displayed in the Log window and pressing the ENTER key. It is also possible to issue several commands at a time by separating commands with semicolons. Long commands can be continued at any point with a backslash (\ ); the next line that has been typed over is the continuation line.

By default, **`window trace`** lines do not appear in the Log window.

ADDITIONAL DISCUSSION

“Using the Log Window” on page 20

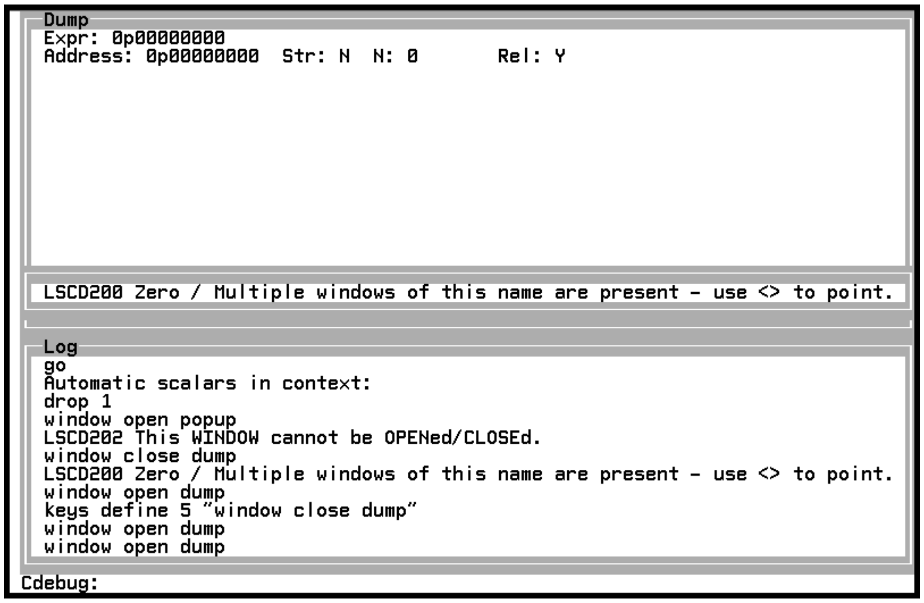
SEE ALSO

The **`window`** command for the following:

- “Command Window” on page 156
- “Source Window” on page 173
- “Status Window” on page 177

Message Window

Display 13.10 Message Window



```
Dump
Expr: 0p00000000
Address: 0p00000000  Str: N  N: 0       Rel: Y

LSCD200 Zero / Multiple windows of this name are present - use <> to point.

Log
go
Automatic scalars in context:
drop 1
window open popup
LSCD202 This WINDOW cannot be OPENed/CLOSEd.
window close dump
LSCD200 Zero / Multiple windows of this name are present - use <> to point.
window open dump
keys define 5 "window close dump"
window open dump
window open dump
Cdebug:
```

DESCRIPTION

The Message window displays error messages. If an invalid command is specified in either the Command or Log window, descriptive error messages are sent to the Log window. Error messages that are associated with all other windows are displayed in a Message window that is similar to the one shown above. The Message window is automatically opened by the debugger. After viewing the message, you can press ENTER or any PF key to close the window. Pressing a PF key that contains an invalid command also results in the display of a Message window that contains an error message.

The Message window is used to display error messages that do not accept input as opposed to the Popup window, which is used to request correction of invalid input.

ADDITIONAL DISCUSSION

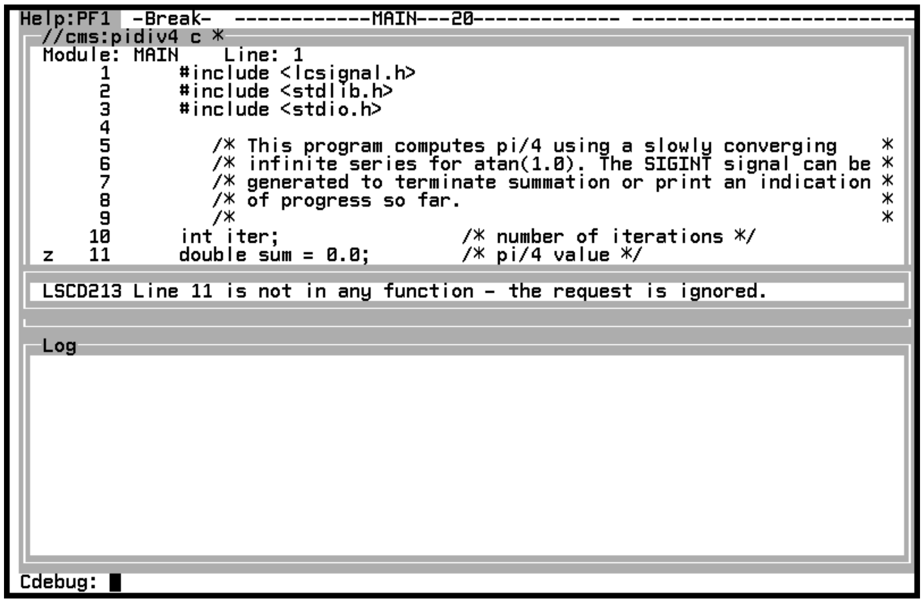
"Factors Affecting Your Full-Screen Session" on page 36

SEE ALSO

"Popup Window" on page 170

Popup Window

Display 13.11 Popup Window



```
Help:PF1  -Break-  ------------MAIN---20------------ -----------------------
  //cms:pidiv4 c *
 Module: MAIN    Line: 1
        1      #include <lcsignal.h>
        2      #include <stdlib.h>
        3      #include <stdio.h>
        4
        5         /* This program computes pi/4 using a slowly converging    *
        6         /* infinite series for atan(1.0). The SIGINT signal can be *
        7         /* generated to terminate summation or print an indication *
        8         /* of progress so far.                                     *
        9         /*                                                         *
       10      int iter;              /* number of iterations */
 z     11      double sum = 0.0;      /* pi/4 value */

 LSCD213 Line 11 is not in any function - the request is ignored.

 Log

Cdebug: █
```

DESCRIPTION

The Popup window requests additional information. If invalid input is specified in certain areas of certain windows, an alarm sounds, and a Popup window is automatically opened. For example, the Popup window shown in Display 13.11 on page 170 was displayed when an invalid prefix command was entered on line 11 of the Source window. The Popup window contains an error message, a prompt, and the old (invalid) text. At this point all other windows stop responding and the debugger will accept input only in the Popup window. You should either correct the error by typing in a valid value, or clear the field to restore the old value; then press ENTER or any PF key.

If input is specified in multiple windows, the debugger processes input in priority order. If the action taken for a particular window involves using a Popup window to correct invalid input, then any pending input in windows of a lower priority is processed after the error situation is corrected.

ADDITIONAL DISCUSSION

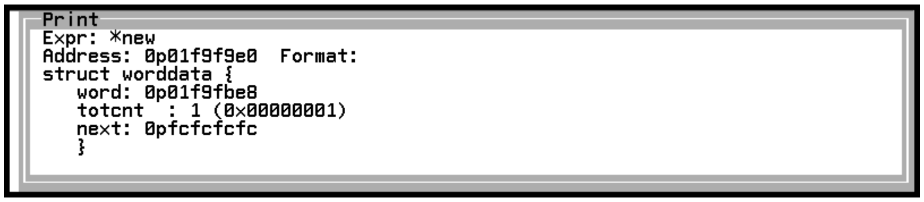
"Window and PF Key Priorities" on page 36

SEE ALSO

"Message Window" on page 169

Print Window

Display 13.12 Print Window



```
Print
Expr: *new
Address: 0p01f9f9e0  Format:
struct worddata {
   word: 0p01f9fbe8
   totcnt  : 1 (0x00000001)
   next: 0pfcfcfcfc
   }
```

DESCRIPTION

The Print window displays the value of an expression. The **>** and **>>** command prefixes are used with the **`print`** command to direct output from the command to either a new or an existing Print window. See “Directing Commands to a Window” on page 26 for information about the > and >> command prefixes. When you are directing the **`print`** command to the Print window, EXPRESSION and %FMT are the only arguments that can be used.

As shown in Display 13.12 on page 171, the top two lines of the Print window contain three fields:

`Expr:`
identifies the expression that is displayed.

`Address:`
shows the address of the object that is represented by the expression. This address may not be valid if the expression contains more that one variable. The **`Address:`** field is protected and cannot be used to specify the address of an expression to be displayed.

`Format:`
is the format that is used to display the value of the expression. Leaving the format area blank causes the debugger to use the default format.

Once a Print window has been opened, you can specify a new **`Expr:`** or **`Format:`** value by typing that value following the appropriate prompt and pressing the ENTER key.

The top two lines are followed by the value area, which contains one or more lines. The value area is always 200 characters wide, which usually enables you to display the entire value. Long lines in the value area can be viewed by scrolling the window left and right. Note that value area width cannot be changed and that it is independent of the window width, which you can change.

If the expression is of scalar type, and a new window is used to display its value, the value area is only one line high. If an existing window is used, no adjustment to the height is made when the value is displayed; however, you are free to adjust the size of the Print window or scroll the value area once it has been displayed.

If the expression is of aggregate (structure, union, or array) type and a new window is used to display its value, the height of the value area is determined by your default configuration. A structure is shown in the Print window in Display 13.12 on page 171. If an existing window is used, no adjustment to the height or width is made when the expression is displayed. If the value area is not sufficient to display all the members of the structure, union, or array, you can vertically scroll through the value area.

ADDITIONAL DISCUSSION

- “Using the Print Window” on page 33
- “Directing Commands to a Window” on page 26

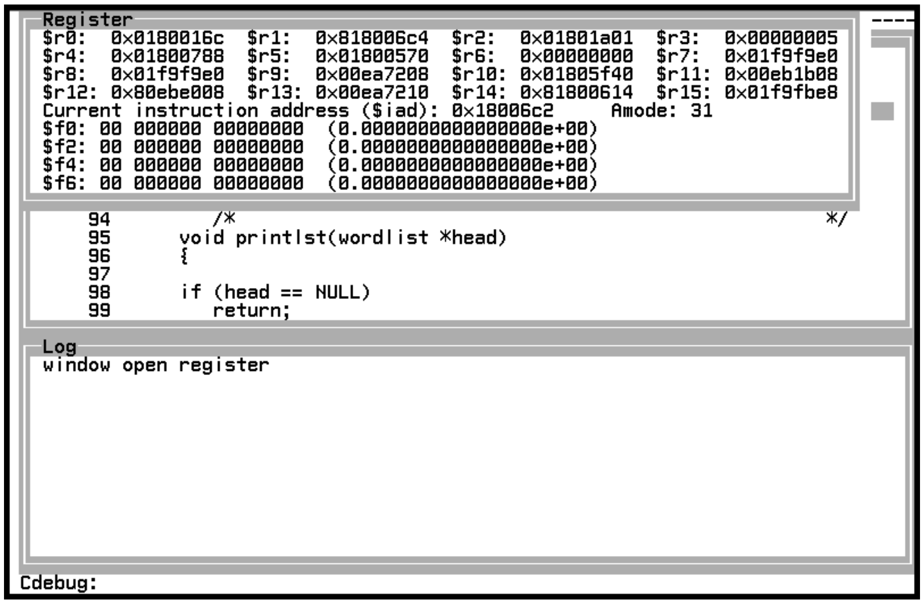
SEE ALSO

The **print** command for the following:

- □ "Dump Window" on page 162
- □ "Watch Window" on page 184

Register Window

Display 13.13 Register Window



```
Register
$r0:  0x0180016c  $r1:  0x818006c4  $r2:  0x01801a01  $r3:  0x00000005
$r4:  0x01800788  $r5:  0x01800570  $r6:  0x00000000  $r7:  0x01f9f9e0
$r8:  0x01f9f9e0  $r9:  0x00ea7208  $r10: 0x01805f40  $r11: 0x00eb1b08
$r12: 0x80ebe008  $r13: 0x00ea7210  $r14: 0x81800614  $r15: 0x01f9fbe8
Current instruction address ($iad): 0x18006c2     Amode: 31
$f0: 00 000000 00000000  (0.0000000000000000e+00)
$f2: 00 000000 00000000  (0.0000000000000000e+00)
$f4: 00 000000 00000000  (0.0000000000000000e+00)
$f6: 00 000000 00000000  (0.0000000000000000e+00)

           /*                                                      */
        void printlst(wordlist *head)
        {

        if (head == NULL)
           return;

Log
window open register

Cdebug:
```

DESCRIPTION

The Register window enables you to take a close look at the way your program uses the machine. You can view the following information:

- □ the contents of the 16 general purpose registers (debugger variables $r0 through $r15) in hexadecimal format
- □ the contents of the 4 floating-point registers (debugger variables $f0 through $f6) in both hexadecimal and floating-point format
- □ the address of the current instruction (debugger variable $iad) in hexadecimal format
- □ the address mode.

The only way to open the Register window is with the **window open register** command. Issuing a **window open register** command when the Register window is already open causes it to be updated. Pressing ENTER also updates it.

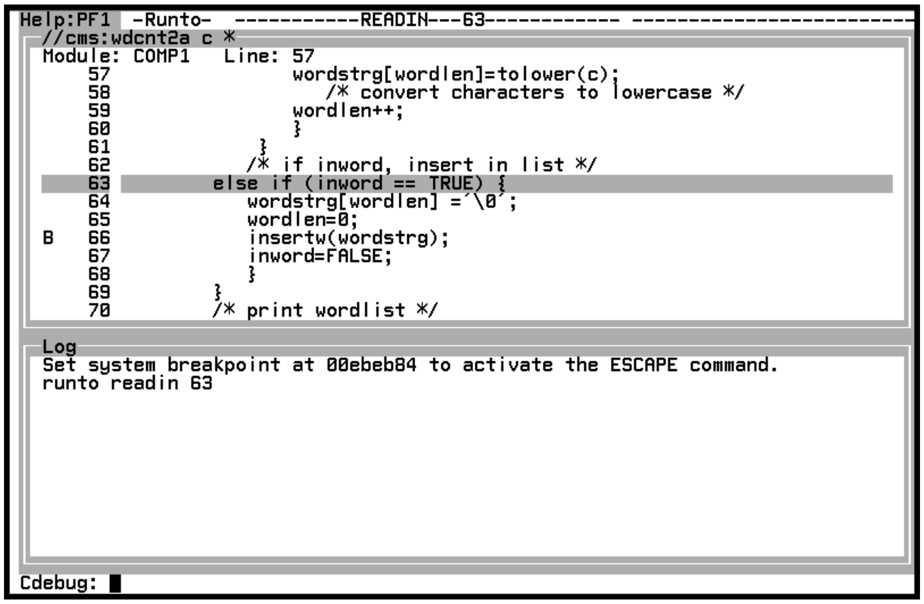
Each of the registers and the current instruction address is associated with a debugger variable. These variables are assigned values that are based on the contents of the register. For example, using the Register window as shown above, you can determine that debugger variable $r0 has a value of **0x0180016c**, which was derived from the contents of general purpose register 0. Debugger variables can be used as arguments to commands, such as the **print** and **dump** commands, that support expressions.

ADDITIONAL DISCUSSION

"Using the Register Window" on page 34

Source Window

Display 13.14 Source Window



```
Help:PF1  -Runto-  -----------READIN---63------------ -------------------------
 //cms:wdcnt2a c *
 Module: COMP1   Line: 57
     57                 wordstrg[wordlen]=tolower(c);
     58                    /* convert characters to lowercase */
     59                 wordlen++;
     60                 }
     61              }
     62             /* if inword, insert in list */
     63          else if (inword == TRUE) {
     64             wordstrg[wordlen] = '\0';
     65             wordlen=0;
 B   66             insertw(wordstrg);
     67             inword=FALSE;
     68             }
     69          }
     70          /* print wordlist */

 Log
 Set system breakpoint at 00ebeb84 to activate the ESCAPE command.
 runto readin 63

Cdebug: █
```

DESCRIPTION

The Source window displays source code. It is one of the four primary windows. The other primary windows are Status, Log, and Command. By default, the Source window is displayed below the Status window and above the Log window. Like all the primary windows, the Source window is always open, even when it is obscured by an overlying window.

Source filename The top border of the Source window, which is always present, contains the name of the source file. The format of the source filename is operating-environment specific. The Source window that is shown in Display 13.14 on page 173 was captured under CMS. If you are running under TSO, your source filenames follow TSO file-naming conventions.

Information displayed in the Source window The first line of the Source window contains the following fields:

`Module:`

identifies the compilation that is displayed in the Source window. Large programs can be broken down into several modules that are compiled separately to facilitate the debugging of manageably sized pieces of code. You can use the **`sname`** compiler option to specify a section name that is used as the module name.

`Line:`

displays the line number of the first line of source code that is displayed in the visible portion of the Source window text area. The line number changes as you scroll through your source code.

The remaining portion of the Source window contains two unprotected areas that are used to view your source code and issue prefix commands.

line number and prefix area
: is displayed on the left side of the window. This area displays line numbers that correspond to the line numbers from your source code file. You can use this area to type and display prefix-area commands. For example, in the Source window that is shown in Display 13.14 on page 173, one **`B`** (**`break`**) prefix command is displayed on line 66.

text area
: is located to the right of the line numbers. The text area is used to display source code from the source file that is identified in the top border of the window.

Run scope The highlighted line in the text area shows where your program has stopped. This location in your code determines the run scope, which is displayed in the Status window. For example, in the display that is shown in Display 13.14 on page 173, line 63 is highlighted. This line is located in the READIN function, as indicated by the run scope field that is in the Source window.

Scrolling through the Source window The **`window scroll`** command is used to scroll through the Source window. In addition to the PF19, PF20, PF22, and PF23 keys that are used to scroll through most windows, there are two additional PF keys that are dedicated to scrolling in the Source window. By default, PF7 is assigned the **`window scroll source up`** command and PF8 is assigned the **`window scroll source down`** command.

In order to use the PF19, PF20, PF22, or PF23 keys to scroll in the Source window, you must first move the cursor into the Source window. The PF7 and PF8 keys can be used to scroll the Source window when the cursor is located in any window.

When the Source window is scrolled vertically, both the line number and text area are scrolled. However, when the window is scrolled horizontally, only the text area is scrolled.

Jumping to a line The **`Module:`** and **`Line:`** fields can be used to jump directly to a specific location in your source code. This makes it possible to avoid scrolling over a large number of lines to view source code that is located far from your current position. To jump directly to a specific location in your code, type the module name and line number following the appropriate prompts on the first line of the window and press ENTER. If either the module name or line number is invalid, a Popup window is displayed that enables you to correct your mistake.

Amount of code displayed in the Source window Two factors affect the amount of code that is displayed in the Source window: the visible portion of the window determines how much code is shown at any one time, and the amount of memory that is allocated for source buffers determines the total size of the window (visible plus scrollable portions of the window).

The minimum width of the Source window is 32 columns and the maximum is the width of your terminal; however, the text area that is contained within the window is considerably wider. The text area is always 252 characters wide; only a portion of the text area is visible at one time.

Source buffers are used to hold the information that is displayed in the Source window text area. You can use the **`window memory`** command in your configuration file to control the amount of memory that is used for source buffers. Note that the amount of memory that is allocated for window buffers cannot be changed during a session. Increasing source buffer size increases the amount of source code that is held in memory, which, in turn, speeds up display of source code in the Source window. See “window” on page 280 for more information about the **`window`** command.

Context information in the source text area The debugger displays lines before and after the highlighted line so that the line is displayed in context. The debugger tries to minimize the vertical movement of the source code. Either the **`window context source`** command or the Config window can be used to control the number of lines of context information that are shown around the highlighted line.

The **`window context source`** command also enables you to specify how many lines in your source code can be jumped without centering the highlighted line in the text area. If the next line to be executed is at least as many lines as you have specified above the top or past the bottom of the window, the next line will be centered when it is highlighted. The **`window context source`** command is explained in "window" on page 280.

The **`Border:`** and **`Jump:`** fields in the Config window can also be used to control the Source window context. See "Config Window" on page 157 for details.

Prefix-area commands Commands that can be issued from the prefix area of the line number field are listed in Table 13.1 on page 175. These commands can be typed only between the lines that contain the opening and closing braces of a function. Only one command can be typed at any time on a line, though there can be more than one command in effect for that line.

Table 13.1 Prefix-Area Commands

Prefix-Area Command	Command Name	Action
`b`	**`break`**	break on this line
`d`	**`disable`**	disable the request in this line
`e`	**`enable`**	enable the request in this line
`g`	**`goto`** and **`resume`**	go to (resume execution at) this line
`q`	**`query`**	display all requests that apply to this line
`r`	**`runto`**	run to this line: install a temporary breakpoint and continue execution
`t`	**`trace`**	trace this line

If you enter an invalid prefix-area command, a window appears that enables you to correct the invalid input. Any attempt to issue a valid prefix-area command outside the opening and closing braces of a function causes the debugger to display an error message in a Message window.

Detailed reference information for each of the prefix-area commands is provided in Chapter 14, "Command Directory," on page 187.

Enabling and disabling breakpoints and action requests You can use the **`e`** (**`enable`**) and **`d`** (**`disable`**) prefix-area commands to enable and disable breakpoint and action requests, provided that only one request is assigned to that line.

The debugger's request system keeps track of your breakpoint and action requests by using a request number. The number is assigned when the request is made. However, the Source window is line-number oriented; therefore, the **`e`** and **`d`** prefix-area commands can be used only when exactly one request applies specifically to that line. The examples that follow illustrate this.

Suppose that you have issued the following commands to request breakpoints:

`break main 24`
assigns a breakpoint to line 24 in the **`main`** function.

`break main 20:30`
assigns breakpoints to lines 20 through 30 in the **`main`** function.

`break main *`
assigns breakpoints to every line in the **`main`** function.

Each of these commands applies to line 24, but only the first applies specifically to that line. Hence, the debugger will successfully process an **`e`** or **`d`** command for **`main`** 24.

Note: The **`query`** command can be used to display requests by request number. A disabled request is indicated by an asterisk (*) after the request number. △

For the next example, assume that **`smain`** is the section name of the compilation that contains **`main`**. Issue the following commands:

`break main 23`
assigns a breakpoint to line 23 of the **`main`** function.

`on main 23 print x`
uses the **`on`** command to assign a **`print`** command to line 23 in the **`main`** function.

`trace (smain) 23`
assigns a **`trace`** command to line 23 of the **`smain`** compilation.

All three requests apply specifically to line 23. Therefore, the debugger will not process any **`e`** or **`d`** commands in line 23; instead, a suitable message will be issued. In this situation you can use the **`drop`**, **`disable`**, and **`enable`** commands with the **`query`** command from the Command window to modify your breakpoint and action requests for line 23.

Specifying an entry or return suffix Except for the **`g`** (**`goto`**) command, any of the prefix-area commands can take an **`e`** or **`r`** suffix. The **`e`** suffix specifies that the command applies to the entry hook of the function within whose scope the command is issued. Conversely, the **`r`** suffix specifies that the command applies to the return hook of the function. For example, the following prefix-area commands can be specified:

`be`	Break on entry to this function.
`rr`	Run to the return hook of this function.
`dr`	Disable the request that applies to the return hook of this function.

Visual indication of break, ignore, on, and trace commands The debugger visually indicates the presence of **`break`**, **`ignore`**, **`on`**, and **`trace`** commands that are in effect for requests that apply to the following:

- a specific line in a function
- a specific line in a module (compilation)
- a specific range of lines in a function
- a specific range of lines in a module (compilation).

Indication consists of one or more of the following characters that appear in the prefix area:

`B`	**`break`**
`I`	**`ignore`**
`O`	**`on`**
`T`	**`trace`**

However, no visual indication is given for commands on function entry, return, or call hooks or commands that are specified with the * (all line hooks) parameter.

Input in multiple areas of the Source window If input is specified in more than one area of the Source window, the data is processed in the following order: first the prefix-area commands, then the module-name change, and finally the line-number change.

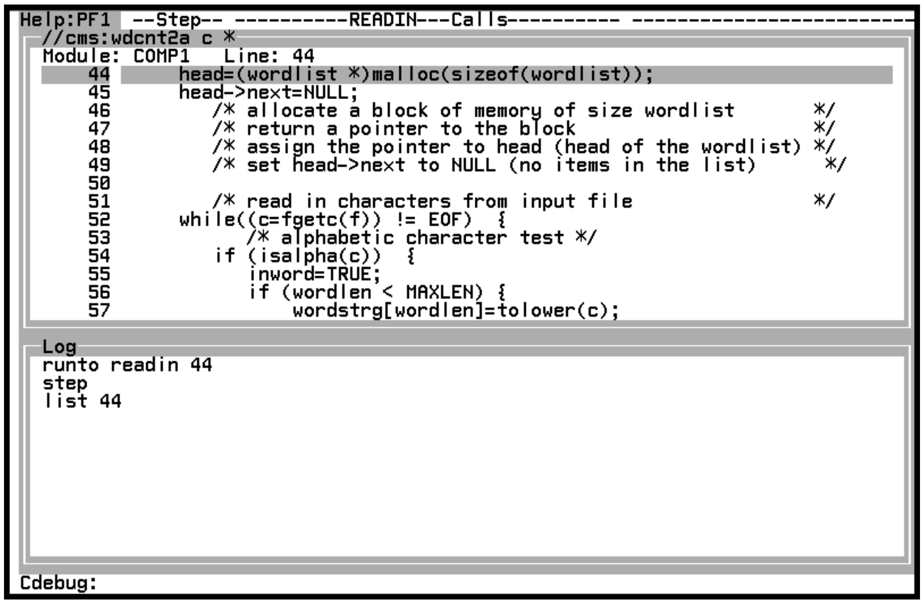
Extended name support The symbol tables that are used by the debugger to determine the function that was operated on have a 255-character limit. If a module has been compiled with the **`extname`** compiler option, the debugger attempts to use the extended name. When a prefix-area command is used to install a request, the debugger temporarily sets the **`auto extname`** option during the request installation process to match the setting that was used when the module was compiled. For example, if the module was compiled with the **`auto extname`** option in effect, a breakpoint that was installed by a **`b`** in the prefix area uses extended names. When the request is installed, the setting of the **`auto extname`** option is restored to its previous setting. See "auto" on page 196 for additional information.

ADDITIONAL DISCUSSION
"Using the Source Window" on page 22

SEE ALSO
- □ "Command Window" on page 156
- □ "Log Window" on page 167
- □ "Status Window" on page 177

Status Window

Display 13.15 Status Window



```
Help:PF1  --Step-- ----------READIN---Calls---------- ------------------------
  //cms:wdcnt2a c *
  Module: COMP1   Line: 44
      44      head=(wordlist *)malloc(sizeof(wordlist));
      45      head->next=NULL;
      46         /* allocate a block of memory of size wordlist      */
      47         /* return a pointer to the block                    */
      48         /* assign the pointer to head (head of the wordlist) */
      49         /* set head->next to NULL (no items in the list)      */
      50
      51         /* read in characters from input file               */
      52      while((c=fgetc(f)) != EOF)  {
      53            /* alphabetic character test */
      54         if (isalpha(c))  {
      55            inword=TRUE;
      56            if (wordlen < MAXLEN) {
      57                wordstrg[wordlen]=tolower(c);
  Log
  runto readin 44
  step
  list 44

Cdebug:
```

DESCRIPTION
The Status window displays status information. It is one of the four primary windows. The other primary windows are Source, Log, and Command. By default,

the Status window is displayed on the top line without a border. Like all the other primary windows, the Command window is always open, even when it is obscured by an overlaid window.

The Status window displays information about the current status of your debugging session. This information is displayed in the following fields:

`Help:`
identifies the PF key that is assigned as the help key. Pressing the help key opens the Help window to display cards from the help system. You can use the Keys window to change the help key assignment.

reason for entering the debugger
is displayed in the second field from the left. In the above display, **`Step`** is displayed because control was transferred from the executing program to the debugger as the result of a debugger **`step`** command.

run scope
is displayed in the third field from the left. In Display 13.15 on page 177 the run scope is located at the entry hook for the READIN function. This is also shown by the highlighted line in the Source window. Run scope is the location in your code at which control was transferred to the debugger. Run scope is identified by function name and either line number, or, in the case of function calls, the side of the call on which the run scope is located: calls, entry, or return.

command scope
is displayed in the fourth field from the left. In Display 13.15 on page 177 the command scope is located in line 24 of the **`main`** function. Certain commands, such as **`break`**, **`goto`**, and **`runto`**, use command scope to determine default function and section names. Command scope is identified in the same manner as run scope.

Changing command scope The **`scope`** command can be used to change command scope. However, if you have more than one function in your calling sequence, you can move the cursor into the Status window and use the PF19 or PF20 keys to change command scope. The **`window scroll < > up`** or **`window scroll < > down`** commands are assigned to these keys by default. The PF19 key causes the run scope to move up in the calling sequence, and the PF20 key causes the run scope to move down in the calling sequence.

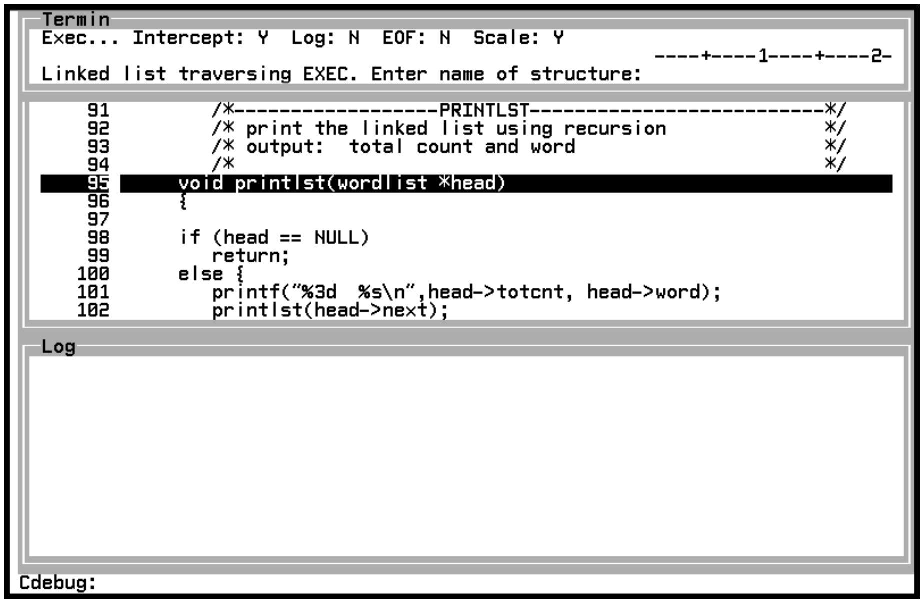
ADDITIONAL DISCUSSION
"Using the Status Window" on page 20

SEE ALSO
The **`scope`** command for the following:

- □ "Keys Window" on page 166
- □ "Source Window" on page 173

Termin Window

Display 13.16 Termin Window



```
Termin
Exec... Intercept: Y  Log: N  EOF: N  Scale: Y
                                                   ----+----1----+----2-
Linked list traversing EXEC. Enter name of structure:

        /*-----------------PRINTLST-------------------------*/
        /* print the linked list using recursion             */
        /* output:  total count and word                     */
        /*                                                   */
     void printlst(wordlist *head)
     {

     if (head == NULL)
        return;
     else {
        printf("%3d  %s\n",head->totcnt, head->word);
        printlst(head->next);
Log

Cdebug:
```

DESCRIPTION

The Termin window is used to type terminal input requested by a program, CLIST, or EXEC. By default, the debugger intercepts program input requests, opens the Termin window, and prompts you for input. As shown in Display 13.16 on page 179, there are three areas in the Termin window: a status and prompt line, an optional scale or ruler line, and the input field.

Status and prompt line The Termin window is controlled by the following fields located in the status and prompt line:

`Read...`, **`Exec...`**, or **`Cont...`** prompt

flashes to indicate that input is required. This prompt displays **`Read...`** or **`Exec...`** when the window is first opened. **`Read...`** displays to indicate that the input is requested by a program, and **`Exec...`** displays to indicate that input is requested by a CLIST or an EXEC. If you continue input, as described later in Entering terminal input, this prompt changes to **`Cont...`**, indicating that additional input is required.

`Intercept:`

controls the input intercept. By default, Intercept is set to **`Y`** (yes), which causes the Termin window to automatically intercept terminal input requests. Typing **`N`** (no) in this field turns off the input intercept.

Returning to line mode also turns off the intercept. Returning to full-screen mode automatically restores the previous status of the intercept.

`Log:`
determines whether input that is typed into the Termin window is copied to the Log window. By default, Log is set to **`N`** (no). Typing **`Y`** (yes) in this field sends a copy of the input to the Log.

Note: The Config window can be used to select a unique color for terminal input that is copied to the Log window.

`EOF:`
is normally set to **`N`** (no) but can be temporarily set to **`Y`** (yes) to signal an end-of-file condition when the Termin window is prompting you for input. If you intend to continue a line, specify **`Y`** only on the last display.

The **`EOF:`** prompt is also used to return a condition code to a CLIST or EXEC. See Using the Termin window with the dbinput command later in this section.

`Scale:`
displays a scale, or column ruler, on the second line of the Termin window. By default, this field is set to **`Y`** (yes), displaying the scale. If this field is set to **`N`** (no), the scale is not displayed and the window shrinks by one line. Setting it back to **`Y`** causes the window to grow by one line and the scale to be redisplayed.

Typing Terminal input   Terminal input is typed into the input field following the **`:`** prompt, which is located directly under the scale prompt. If there is not sufficient space in the input field, you can resize the Termin window horizontally or you can continue input using the backslash (\) as the continuation character. If input is continued in this manner, the flashing **`Read...`** or **`Exec...`** prompt is replaced by a flashing **`Cont...`** prompt. Another \ character can be used to repeat the process. You can type up to 255 characters. If more than 255 characters are entered, the debugger truncates to 255.

terminal input prompt   If the prompt is terminated with a **`\ n`**, then the **`\ n`** appears at the end of the prompt area as two separate characters, a **`\`** and an **`n`**. Input prompts greater than 64 characters long are truncated. If a **`\`** and an **`n`** had been the 64th and 65th characters, then the prompt would have been truncated after 63 characters.

Restrictions while input is pending   You can move between windows and examine variables while either the flashing **`Read...`** or **`Cont...`** prompt is displayed in the Termin window; however, you cannot issue any command that continues execution. Satisfying the input request by typing input at the Termin window is the only way to return control to the program and automatically continue execution. Also, you cannot close the Termin window while the **`Read...`**, **`Exec...`**, or **`Cont...`** prompt is displayed.

Turning off the Intercept prompt   If it is not possible to open the Termin window, the debugger turns off the intercept. This can occur when you have too many windows open at the same time. You can change the default setting of the **`Intercept:`**, or any other prompt, by making the selections you want in the Termin window and saving the changes to your configuration file. Changes are saved with the **`config save`** command. See “Setting Up a Configuration File” on page 46 for more information.

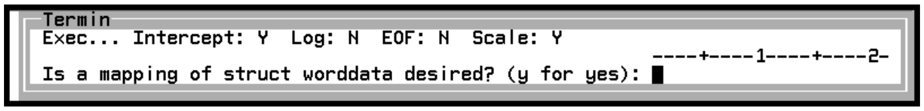
Resizing the Termin window   You can resize the Termin window horizontally to enable additional information to be input on a line as described earlier. However, you cannot resize the window vertically.

Using the Termin window with the dbinput command   In full-screen mode, the Termin window is used to type input required by a **`dbinput`** command. In this case the **`:`** prompt is changed to display the prompt string that is specified by the STRING argument of the **`dbinput`** command. For example, Display 13.17 on page 181 shows the Termin window as it would be displayed if the intercept were caused by the following **`dbinput`** command:

```
dbinput EXEC_VAR ''Is a mapping of struct worddata desired? (y for yes):''
```

In this case, input entered in the Termin window is assigned the CLIST or EXEC variable named EXEC_VAR.

Display 13.17 Display 11.17 Termin Window Prompt String



```
Termin
Exec... Intercept: Y  Log: N  EOF: N  Scale: Y
                                               ----+----1----+----2-
Is a mapping of struct worddata desired? (y for yes): █
```

When you are using the Termin window in conjunction with the **dbinput** command, the **EOF:** prompt is used to return a condition code to the CLIST or EXEC that contains the **dbinput** command. Setting the **EOF:** prompt to **Y** causes a condition code of –13 to be returned to an EXEC or +13 to a CLIST. It also causes the stack to be flushed if the return code is being passed to a CLIST.

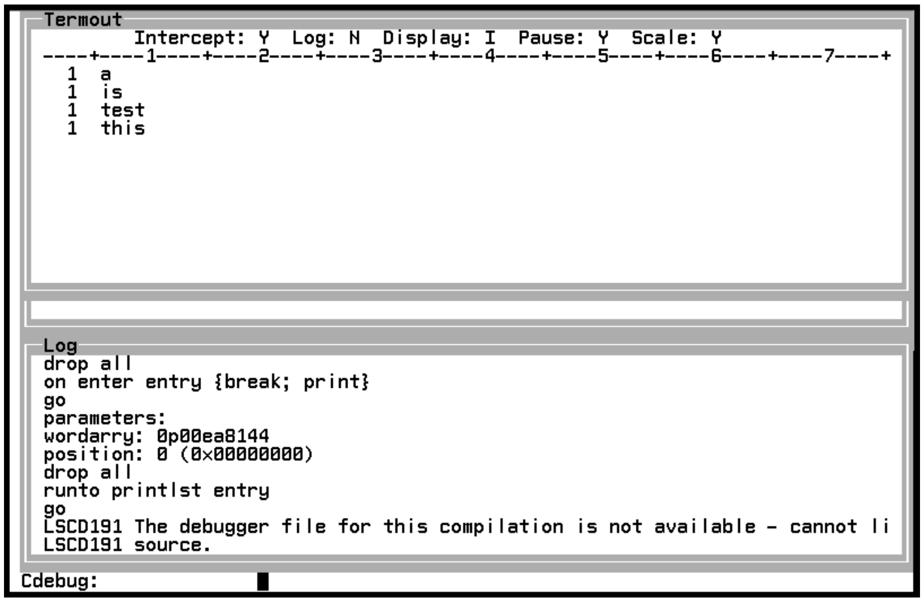
When too many windows are opened, the debugger might not be able to open the Termin window. If the Termin window cannot be opened, a prompt is displayed, and you can still type the input that is required by the **dbinput** command called from a CLIST or EXEC. Once input is typed, you can continue your debugging session.

ADDITIONAL DISCUSSION
"Looking at Terminal I/O" on page 31

SEE ALSO
The **dbinput** command for the "Termout Window" on page 181

Termout Window

Display 13.18 Termout Window



```
Termout
        Intercept: Y  Log: N  Display: I  Pause: Y  Scale: Y
----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+
  1  a
  1  is
  1  test
  1  this

Log
drop all
on enter entry {break; print}
go
parameters:
wordarry: 0p00ea8144
position: 0 (0x00000000)
drop all
runto printlst entry
go
LSCD191 The debugger file for this compilation is not available - cannot li
LSCD191 source.
Cdebug:             █
```

DESCRIPTION

The Termout window displays output that is directed to the terminal by your program. By default, the debugger intercepts output from functions such as **`printf`**, opens the Termout window, and displays the terminal output in the window. As shown in Display 13.18 on page 181, there are three areas in the Termout window: a status and prompt line, an optional scale or ruler line, and an output area.

Status and prompt line The Termout window is controlled by the following fields that are located in the status and prompt line:

`More...` prompt

flashes to indicate that more information is waiting to be displayed in the Termout window. (See **`Pause:`** below)

`Intercept:`

controls the output intercept. By default, **`Intercept:`** is set to **`Y`** (yes), which causes the Termout window to automatically intercept terminal output. Typing **`N`** (no) in this field stops the output intercept.

`Log:`

determines whether intercepted output is copied to the Log window. By default, **`Log:`** is set to **`N`** (no). Typing **`Y`** (yes) in this field sends a copy of the output to the Log.

Note: The Config window can be used to select a unique color for terminal output that is copied to the Log window. △

`Display:`

can be set to one of the following values:

`Y` (yes)

displays intercepted output in the Termout window. The output appears the next time that control is transferred to the debugger.

`N` (no)

does not display intercepted output in Termout window.

`I` (intercept)

displays intercepted output immediately. **`I`** is the default setting for the **`Display:`** field.

Regardless of whether **`Y`** or **`I`** is in effect, the debugger does not display intercepted output until there is a complete line, since line length is determined by the usable window area. A **`\ n`** in the output also completes a line.

`Pause:`

pauses the debugger if the output area is full and the debugger needs to output another line. By default, this field is set to **`Y`** (yes), which enables the pause feature; setting the field to **`N`** (no) disables the feature.

If the debugger is paused, you are notified of this state by the appearance of a flashing **`More...`** prompt on the first line of the Termout window. Pressing the ENTER key when the cursor is inside the Termout window causes the display to be cleared and the new line is displayed.

While the debugger is paused, you can move between windows and examine variables; however, you cannot issue any command that continues execution. Also, you cannot close the Termout window while the debugger is in the paused state. Pressing the ENTER key in the Termout window is the only way to cause the debugger to return control to the program to automatically continue execution. If you are intercepting output in the Termout window with pause in effect, then if there is any output that you have not seen at program completion time, the debugger stops to let you view it.

`Scale:`
displays a scale, or column ruler, on the second line of the Termout window. By default, this field is set to **`Y`** (yes), displaying the scale. If set to **`N`** (no), the scale is not displayed and that area becomes an additional line that is available for output. Changing the setting while output is displayed has the following effects:

- □ If the setting is changed from **`Y`** to **`N`**, the output is not moved. However, after the field is full, the debugger uses the scale line to display output.
- □ If the setting is changed from **`N`** to **`Y`**, the scale appears on the first line of the output field, overwriting any output that appears on that line.

Intercepting output When the appropriate settings are in effect (**`Intercept: Y`**, **`Display: Y`** or **`I`**), the Termout window is automatically opened whenever there is a complete line of output to be displayed. However, if it is not possible to open the window, the debugger stops the intercept.

If you do not want to intercept terminal output, change the default prompt settings by making the selections that you want in the Termout window and save the changes to your configuration file. Changes are saved with the **`config save`** command. See "Setting Up a Configuration File" on page 46 for more information.

Clearing the Termout window The output area is cleared each time the window is opened. The **`window clear`** command also clears the output area.

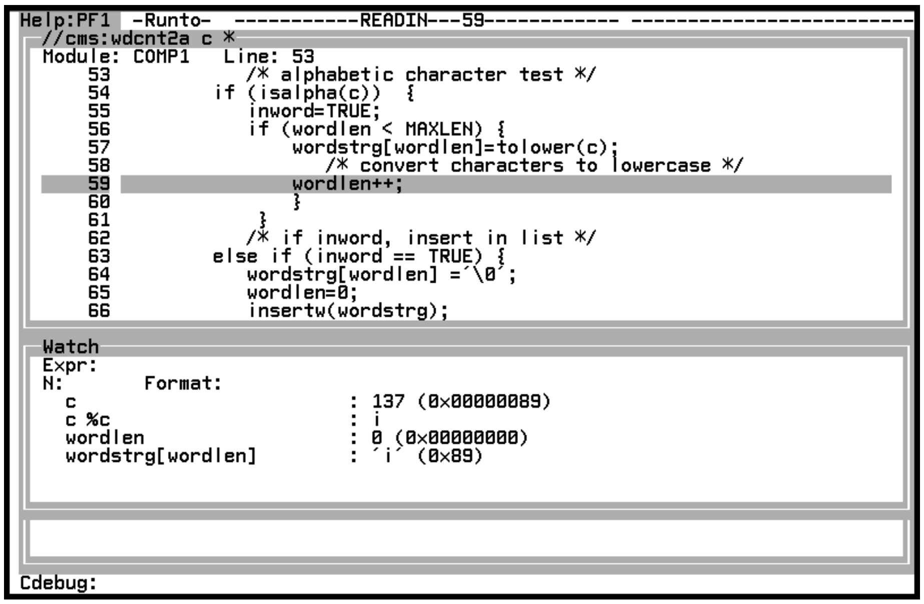
Resizing the Termout window Resizing the window to a smaller width can cause data to be truncated on the right. Resizing to a larger size does not cause the truncated data to reappear. Furthermore, resizing the window to a smaller size can cause the output area to be cleared. If the resized window has fewer lines in the output area than are currently displayed, the output is cleared.

ADDITIONAL DISCUSSION
"Looking at Terminal I/O" on page 31

SEE ALSO
"Termin Window" on page 179

Watch Window

Display 13.19 Watch Window



```
Help:PF1  -Runto-  -----------READIN---59----------- ------------------------
 //cms:wdcnt2a c *
 Module: COMP1   Line: 53
                 /* alphabetic character test */
               if (isalpha(c))  {
                  inword=TRUE;
                  if (wordlen < MAXLEN) {
                      wordstrg[wordlen]=tolower(c);
                         /* convert characters to lowercase */
                      wordlen++;
                      }
                  }
                 /* if inword, insert in list */
               else if (inword == TRUE) {
                  wordstrg[wordlen] =´\0´;
                  wordlen=0;
                  insertw(wordstrg);
 Watch
 Expr:
 N:       Format:
   c                       : 137 (0x00000089)
   c %c                    : i
   wordlen                 : 0 (0x00000000)
   wordstrg[wordlen]       : ´i´ (0x89)

Cdebug:
```

DESCRIPTION

The Watch window tracks values of expressions or areas of memory. It acts as an automatic **print** or **dump** command, displaying the expression or area of memory each time control is transferred to the debugger. As shown in Display 13.19 on page 184, the Watch window contains several fields that are used to control the window, as well as two areas that are used to display the expressions or areas of memory being watched.

Selecting a watch   You can select as many as 40 expressions or areas of memory to be watched. The watches are completely independent of each other and the debugger does not check for duplicates. Each watch is updated as the value of the expression or the contents of the memory location change. The following fields are used to select an expression or area of memory to be watched:

Expr:
specifies an expression to be watched. Any expression that is valid as an argument to the **print** or **dump** command can be issued, provided that it can be evaluated at the time that the watch is entered. When the ENTER key is pressed, the expression is displayed in the watch name field, and its value is displayed in the expression value area.

N:
specifies the number of bytes to be watched. The maximum value for this field is 64, which dumps 64 bytes of memory starting at the address that is indicated by the expression in the **Expr:** field. The **N:** field be used can only when you are typing an expression that points to an area of memory.

Format:
specifies a display format to be used to format the value that is displayed in the expression value area. You can specify a format when you are watching

the value of an expression; however, you cannot use a format when you are watching an area of memory.

Display area After a watch is selected, it is displayed in the following areas:

prefix area
: drops a watch from the Watch window. The prefix area is one column wide and is located immediately before the watch name area. By typing a **d** in the prefix area and pressing ENTER you can drop any of the watches that are displayed in the window.

watch name area
: describes the expression to be watched. In the Watch window in Display 13.19 on page 184, the expression **c** is displayed in the first two watch name areas. The first watch for **c** uses the default format, the second watch for **c** is formatted by %c.

If an expression that you are watching belongs to a function that is different from that of the command scope, the expression that is displayed in the watch name area is prefixed with the name of the function to which it belongs. For example, if you were to step through the program that is shown in the Watch window in Display 13.19 on page 184, each of the expressions that are being watched would be prefixed with MAIN: in the event that you stepped into another function. The reason for this is that the command scope would no longer be the same as the scope of the expressions that are being watched.

If the expression that you are watching is longer than the width of the watch name area, the expression is clipped on the right. Resizing the window by increasing its width displays more of the expression.

expression value area
: displays the value of the expression that you are watching. As shown by the Watch window that is illustrated in Display 13.19 on page 184, the format of the information that is displayed in the expression value area depends on the type of expression and how it was typed. Conversion specifiers that were typed in the **Format:** field affect the format of information that is in the same way as they do when they are used with the **print** command. Typing a number in the **N:** field causes the information to be formatted in a manner similar to the output from the **dump** command.

For **print** style watches, if no format is specified, scalars are displayed by using the same default format as would be used if the expression were displayed by the **print** command.

Unions, structures, and arrays are displayed as a list of values separated by commas and enclosed by braces. If a format is specified, it is used to display the value of all items of the aggregate; if no format is specified, **%d** is used for signed integral items, **%u** is used for unsigned integral items, **%g** for floating-point items, and **0p%08x** for pointers. If there is not sufficient space in the expression value area, the list is ended with an ellipsis (...).

Single-dimensional character arrays are formatted in **dump** style if no format is specified; a maximum of 64 bytes can be displayed using this format. If a format is specified for a single-dimensional character array, the information that is contained in memory is displayed in a list that is similar to that used for other arrays.

For **dump** style watches, the address is displayed and it is followed by the contents of memory. A hexadecimal and character representation of the information in memory is displayed in a format similar to the output from the **dump** command.

The prefix, watch name, and expression value areas can be scrolled up or down. You can also scroll right and left through the expression value area.

Calling sequence You can watch an expression that is anywhere within the calling sequence for your program. However, when you set the watch it must be located inside your command scope. You cannot use the FUNCTION-NAME:IDENTIFIER format to specify an expression that is outside the command scope; you must change your command scope before you type the watch. This format is described in "Specifying Identifiers Outside the Current Function" on page 147. The **`scope`** command can be used to change your command scope to any function that is in your calling sequence.

Thus, each watch has a function that is associated with it. The scope of this function is set each time the debugger evaluates the watch expression, which occurs each time that control is transferred from your program to the debugger. Watches that are based on variables of the **`extern`** or **`static`** storage class can always be evaluated; watches that are based on an automatic variable or parameter can be evaluated only if the function is in the calling sequence. As soon as the function ends, the automatic variable or parameter-based watch is deactivated; the next time the function is typed, the watch is reactivated. This also applies to watches that are based on automatic variables or parameters in a recursive function. Since the watch is reactivated only if inactive, recursive invocations of the function do not set additional watches.

Syntactically invalid input If an invalid input is detected at the time a watch is set, a Popup window is automatically opened and you can correct the mistake. The error does not affect the debugging session in any way.

ADDITIONAL DISCUSSION

"Using the Watch Window" on page 33

SEE ALSO

The **`dump`**, **`print`**, and **`watch`** commands for the following:

- □ "Dump Window" on page 162
- □ "Print Window" on page 171

CHAPTER
14

Command Directory

Introduction

This chapter provides complete reference information for all of the debugger commands. You should also refer to Chapter 12, “Using Debugger Commands,” on page 129 for additional details on the syntax and arguments used in the command formats.

The context-sensitive help system is also designed to provide online help for any of the debugger commands. See “help” on page 224 and “Using the Help Window” on page 16 for information on how to access the help system.

List of Commands

This chapter describes the following debugger commands:

%	execute a CLIST or an EXEC (OS/390 only).
?	list debugger commands.
ab{ort}	abort program execution.
a{ssign}	assign a value to an expression.
at{tn}	generate a SIGINT signal.
au{to}	set debugger modes.
b{reak}	request a breakpoint.
browse	browse the area of the source file where the name being browsed is declared.
catch	request to catch all exceptions.
conf{ig}	assign/identify the configuration file and save current configuration.
c, con{tinue}	continue execution to next line-number hook without stepping into functions.
co{py}	copy one or more items to a new location.
dbi{nput}	called from a CLIST or EXEC for information input.
dbl{og}	called from a CLIST or EXEC to output information.
de{fine}	define a debugger macro.

di{sable}	disable requests.
dr{op}	drop requests.
du{mp}	dump memory contents.
en{able}	enable requests.
es{cape}	transfer control to the operating system debugger.
exe{c}	execute an EXEC or a CLIST.
exi{t}	terminate program execution.
g{o}	start/restart program execution under the debugger.
got{o}	alias for **resume**.
h{elp}	access debugger online help.
i{gnore}	ignore breakpoint or action requests, or signals.
in{stall}	assign or list user-defined commands.
k{eys}	assign or list PF key commands.
l{ist}	output a source line listing.
m{onitor}	check for changes made to an object.
o{n}	perform one or more commands at specified locations.
p{rint}	print the value of an expression.
q{uery}	display breakpoint/action requests.
res{ume}	resume program execution.
ret{urn}	return immediately from a function.
ru{nto}	resume execution and request a temporary breakpoint.
sc{ope}	change command scope.
se{t}	control file access.
s{tep}	restart execution and break at next hook.
sto{rage}	display storage analysis.
sy{stem}	execute a CMS command or a TSO command.
t{race}	trace program flow.
tran{sfer}	transfer debugger/program values to CLIST/EXEC variables.
u{ndef}	undefine a debugger macro.
wa{tch}	assign expressions to the Watch window.
wha{tis}	display type information.
w{here}	produce a traceback.
wi{ndow}	perform window management functions.

%

Execute a CLIST or an EXEC (OS/390 only)

ABBREVIATION
none

FORMAT
`%` CLIST-NAME | EXEC-NAME [ARGUMENTS]

DESCRIPTION
The `%` command executes a CLIST or a REXX EXEC specified by the CLIST-NAME or EXEC-NAME argument. CLIST-NAME or EXEC-NAME is the name of a member in a partitioned data set containing command procedures (CLISTs) or a REXX EXEC. For CLISTs this data set must be allocated to the DDname SYSPROC. EXECs can be allocated to either SYSPROC or SYSEXEC. When you use the `%` command, the TSO EXEC command is issued and SYSEXEC and SYSPROC are searched. SYSEXEC can contain only EXECs; however, SYSPROC can contain either EXECs or CLISTs.

The CLIST specified by CLIST-NAME can contain CLIST statements or SAS/C Debugger commands or both. Similarly, the REXX EXEC specified by EXEC-NAME can contain REXX statements or SAS/C Debugger commands or both.

ARGUMENTS are any arguments that you need to pass to the CLIST or EXEC.

You cannot use another debugger command on the same line following the `%` command. The rest of the line following the `%` command is passed to the CLIST or EXEC as arguments. However, a command can precede `%` on the same line.

Similarly, no other debugger command can be issued on the same line after a `%` command that is used as an argument to the **on** command. Any command on the same line after the `%` is ignored.

See the IBM publication OS/390 V2R9.0 TSO/E Command Reference SC28-1969 for a discussion of CLISTs. See the IBM publication OS/390 V2R9.0 TSO/E REXX User's Guide SC28-1974 for information about REXX EXECs.

ADDITIONAL DISCUSSION
Chapter 3, "Debugger PROFILEs, Configuration Files, and EXECs," on page 41

SYSTEM DEPENDENCIES
The `%` command is valid for OS/390 only.

COMMAND CAN BE ISSUED FROM

PROFILE	yes
configuration file	no
Source window prefix	none

SCOPE
The `%` command is not affected by changes in scope.

RETURN CODES SET
Successful: code set by CLIST or EXEC called
Unsuccessful: parsing error, 1; otherwise, **`execopn()`** code

SEE ALSO
"escape" on page 218

?

List Debugger Commands

ABBREVIATION
none

FORMAT
`?`

DESCRIPTION
The **`?`** command outputs a list of valid names of debugger commands and their abbreviations.

SYSTEM DEPENDENCIES
none

COMMAND CAN BE ISSUED FROM

PROFILE	no
configuration file	no
Source window prefix	none

SCOPE
The **`?`** command is not affected by changes in scope.

RETURN CODES SET
not applicable

SEE ALSO
“help” on page 224

abort

Abort Program Execution

ABBREVIATION
`ab{ort}`

FORMAT
`abort`

DESCRIPTION
The **`abort`** command abnormally ends execution and exits both from the program and the SAS/C Debugger. The program terminates with user ABEND 1220. You receive a traceback from the point in your program where execution stopped.

The **`abort`** command has the same effect (except for the ABEND code) as calling the **`abort`** function within your program. When you issue an **`abort`** command, output buffers are not flushed as files are closed. Therefore, output data can be lost.

SYSTEM DEPENDENCIES

The specific message that you receive from the operating system depends on the operating system and the context. For example, here is a CMS message:

```
DMSABM155T USER ABEND 1220 CALLED FROM address
```

Here is an example of a TSO message:

```
CDEBUG ENDED DUE TO ERROR
```

COMMAND CAN BE ISSUED FROM

PROFILE	yes
configuration file	no
Source window prefix	none

SCOPE

The **abort** command is not affected by changes in scope.

RETURN CODES SET

not applicable

SEE ALSO

- "exit" on page 222
- "where" on page 279

assign

Assign a Value to an Expression

ABBREVIATION

a{ssign}

FORMATS

Format 1: **assign** SCALAR-TYPE-EXPRESSION = VALUE

Format 2: **assign** AGGREGATE-TYPE-EXPRESSION = 1 {VALUE-LIST} | AGGREGATE-TYPE-EXPRESSION

DESCRIPTION

The **assign** command assigns the value (or values) specified by VALUE, VALUE-LIST, or AGGREGATE-TYPE-EXPRESSION to the expression identified by SCALAR-TYPE-EXPRESSION or AGGREGATE-TYPE-EXPRESSION. In both formats, the equal sign is required.

Format 1: Format 1 assigns a value, specified by VALUE, to the arithmetic, pointer, or bit-field object identified by SCALAR-TYPE-EXPRESSION. The

SCALAR-TYPE- EXPRESSION argument is an expression whose type is arithmetic, pointer, or bit-field.

VALUE is an expression whose type is one of the following:

- □ constant
- □ an expression of scalar type
- □ address
- □ enumeration constant
- □ array name.

See Table 12.3 on page 141 for details on argument types that are used with the **`assign`** command. If the VALUE argument and the SCALAR argument have different types, a conversion is made according to the SAS/C Compiler's rules for conversions.

You can assign a value to any expression of scalar type visible at the point where you issue the **`assign`** command.

Format 2: Format 2 assigns the source values on the right side of the assignment operator (=), specified by VALUE-LIST or AGGREGATE-TYPE-EXPRESSION, to a target aggregate identified by AGGREGATE-TYPE-EXPRESSION on the left side of the assignment operator.

The AGGREGATE-TYPE-EXPRESSION arguments can be an expression of type structure or union, provided that both source and target are declared with the same tag in the same compilation. (AGGREGATE-TYPE-EXPRESSION cannot be an array; use the **`copy`** command with arrays.)

The VALUE-LIST argument can also be used to assign values to the target AGGREGATE-TYPE-EXPRESSION. Any of the items in a VALUE-LIST can be a structure or union. If a union is specified, VALUE-LIST must contain exactly one item that is assigned to the first member of the union. For example, in the following statement, 5 is assigned to the first member of **`some_union`**:

```
assign some_union = { 5 }
```

To assign a value (for example, 9) to the second member of **`some_union`**, use **`assign`** as follows:

```
assign some_union.member2 = 9
```

The AGGREGATE-TYPE-EXPRESSION argument can be a structure with members that are aggregates.

The VALUE-LIST argument contains any or all of the following items, enclosed by braces:

- □ one or more VALUE arguments (as described under **`Format 1`**) separated by commas:

{VALUE, VALUE, VALUE, . . .}

- □ one or more NULL-INITIALIZERs. A NULL-INITIALIZER is an empty pair of braces: {}.
- □ one or more VALUE-LISTs (this means that a VALUE-LIST can contain nested VALUE-LISTs).

Rules for assigning VALUE-LISTs to aggregate objects:

- □ An aggregate must be assigned values via a VALUE-LIST.
- □ The order of the elements in a VALUE-LIST must correspond to the order of members in the aggregate.
- □ If an aggregate has more members than there are elements in the VALUE-LIST to be assigned to it, no assignment is made to the extra members.
- □ If a VALUE-LIST has more elements than there are members in the aggregate to which it is assigned, the extra elements in the VALUE-LIST are ignored. (In this case, you receive a message.)
- □ If the VALUE-LIST contains a NULL-INITIALIZER, the corresponding member of the aggregate is not modified.

One of the elements of the aggregate can be an aggregate, in which case, the rules above apply (recursively) to that element. Therefore, if a member of a structure is a structure, the debugger expects to find a VALUE-LIST nested in another VALUE-LIST. For example, consider the following structure:

```
struct a {
        int b;
        struct ccc d;
        long e;
        };
```

The structure needs a VALUE-LIST, as follows, with the values to be assigned to **struct ccc d** in the inside set of braces:

```
{b-value, {d-value, d-value, d-value, . . .}, e-value}
```

EXAMPLES

The **assign** examples in this section are based on the following declarations and **#define** statement:

```
#define B_MAX 9

int ival;
char *p;
struct XXX int a; short b,c;d;
```

The semantics in the examples are the same as for the assignment statement. In other words, **assign i=5** does the same as **i=5;** in C.

assign ival = 3+B_MAX
assigns 12 to **ival**.

assign p = &ival
assigns the address of **ival** to a pointer object (scalar).

assign ival = 20
assigns a constant to an arithmetic object (scalar).

assign d = {1,2}
assigns values from a value list to a structure **d** (**d.a=1,d.b=2**) .

assign d = {1,ival,3}
assigns values from a value list to a structure **d** (**d.a=1,d.b=20,d.c=3**).

assign d = {d.b}
assigns **d.b** (a short) to the first element of structure **d** (**d.a =20**).

SYSTEM DEPENDENCIES
none

COMMAND CAN BE ISSUED FROM

PROFILE	no
configuration file	no
Source window prefix	none

SCOPE
The **assign** command uses command scope to resolve references to all identifiers.

RETURN CODES SET

Successful: 0

Unsuccessful: 1

SEE ALSO

- □ "copy" on page 205
- □ "return" on page 248
- □ "scope" on page 252
- □ "transfer" on page 271

attn

Generate a SIGINT Signal

ABBREVIATION
at{tn}

FORMAT
attn

DESCRIPTION
The **attn** command generates a SIGINT signal in programs executing under the debugger. A message is sent to the terminal to confirm that the signal is raised. See the discussion of signals in Chapter 17, "Signal-Handling Functions," in the SAS/C Library Reference, Volume 1.

SYSTEM DEPENDENCIES
A SIGINT signal is an interruption from the terminal normally generated under TSO when you press the attention key or under CMS when you issue the command **IC**. However, the attention key (under TSO) or the **IC** command (under CMS) gives control of execution back to the debugger. Thus, the **attn** command is a way of sending a SIGINT signal to an executing program.

COMMAND CAN BE ISSUED FROM

PROFILE	yes
configuration file	no
Source window prefix	none

SCOPE

The **attn** command is not affected by changes in scope.

RETURN CODES SET

Successful: 0

Unsuccessful: 1

auto

Set Debugger Modes

ABBREVIATION

au{to}

FORMAT

auto KEYWORD KEYWORD . . .

DESCRIPTION

The **auto** command is used to specify several characteristics of output produced by the debugger. The **auto** KEYWORDs are as follows:

c{macros}/noc{macros}
Default: nocmacros

When the debugger evaluates expressions in commands, macros defined in your program and used in those expressions can be substituted with their text. To do macro substitution in expressions, you must first set the **cmacros** keyword. You must also have compiled your program with the **dbgmacros** option (Chapter 4, "Compiler Options," on page 57). This keyword affects all debugger commands that use expressions except the **whatis** command.

c{xx}/noc{xx}
Default: nocxx

If you issue an **auto cxx** command, the debugger automatically saves the current status of **auto** command's **extname** keyword and turns on **auto extname**. When **auto cxx** is turned off, the most recently saved value of **extname** is restored. **auto extnam**e cannot be turned off while **auto cxx** is on. The **query** command displays the state of the **cxx** keyword. The **transfer** command supports the **cxx** keyword. If all **auto** command settings are transferred, **cxx** appears between **extname** and **linesize**. The minimum lengths you can specify for the **cmacros** and **nocmancros** keywords of the **autos** command are now two and four, respectively.

d{umpabs}/nod{umpabs}
Default: nodumpabs

The **dumpabs** keyword controls the method of addressing used in output from the **dump** command. Relative addressing is the default, which is selected by the **nodumpabs** keyword. Absolute addressing is selected by the **dumpabs** keyword.

The method of displaying addresses is affected regardless of whether the **dump** command output is displayed in the Log window or directed to the Dump window.

{e}cho/{noe}cho
Default: noecho
The **echo** keyword echoes debugger commands (except for **break** and **trace** used with no arguments) in an **on** command's list. The **echo** keyword is used following the **auto** command in an **on** command's CMD-LIST. Subsequent commands in the CMD-LIST are then echoed. Similarly, the **noecho** keyword is used to turn off echoing of subsequent commands in a CMD-LIST.

ex{ececho}/noex{ececho}
Default: noexececho
The **exececho** keyword echoes each line of the EXEC or TSO CLIST, supplied to the debugger by the subcomm interface (used by the debugger to communicate with the EXEC or CLIST) before the debugger parses and executes the line. This echoing behavior occurs both in line mode and full-screen mode. In line mode, the line goes through the normal ouput interface; in full-screen mode, it appears in the Log window.

ext{name}/noext{name}
Default: noextname
The **extname** keyword selects extended name support. If selected, function names used with commands such as **break**, **on**, and **trace** can be as long as 255 mixed-case characters. This enables the debugger to use extended names that are contained in the debugger symbol table associated with modules that have been compiled with the **extname** compiler option. Refer to the SAS/C Compiler and Library User's Guide for additional information about extended name support.

The default setting, **noextname**, limits function names to eight characters. Also, with the default setting, function names that are eight characters or less in length and do not contain any uppercase characters are converted to all uppercase when they are issued as part of a debugger command.

Changing this keyword during a session only affects requests that are installed after the change is made. Previously issued requests are not affected. The output from the **query** command shows the function names as installed.

i{d}/noi{d}
Default: id
These keywords determine whether a line is produced for **on** commands. The **noid** keyword suppresses both a trace and a source line. For **id**, the type of line that you receive depends on whether you also specified **list** or **nolist**. The **nolist** keyword produces a trace line; **list** (the default) produces a source line. However, because the **on** command generates an output line before executing its CMD-LIST, **id** or **noid** in the CMD-LIST does not affect the format of the current output line, but it is in effect the next time a breakpoint is hit.

lin{esize} nnn
Default: 75
The **linesize** keyword sets the line size of debugger output to the value specified by **nnn**. By default, the debugger displays, at most, 75 characters on each output line. **nnn** must specify between 40 and 251 characters (TSO) or between 40 and 130 characters (CMS, OS/390 batch), inclusive. Nonsource lines greater than the **linesize** value are wrapped. For source lines, see the wrap keyword.

lis{t}/nol{ist}
Default: list
In line mode, the **list** or **nolist** keywords determine the type of identifying line that the debugger outputs, either when the user gains control due to a **break**,

step, **continue**, or **on** command or as the result of executing a **trace** or **on** command. (The **list** and **nolist** keywords are ignored in full-screen mode.) The **list** keyword produces a listing line; the **nolist** keyword produces a trace line.

The **list** or **nolist** keywords can be used with the **auto** command in the CMD-LIST of an **on** command. However, because the **on** command generates an output line before executing its CMD-LIST, **list/nolist** in the CMD-LIST does not affect the format of the current output line, but it is in effect the next time a breakpoint is hit.

If the CMD-LIST contains the **trace** command, the format of the output line may be changed, depending on whether a line was generated when the **on** command took effect and the line's format.

n{ullptr}/non{ullptr}
Default: nonullptr

The **nullptr** keyword enables you to dereference null pointers. These keywords affect expressions that are used in debugger commands.

w{rap}/now{rap}
Default: wrap

In line mode, the **wrap** and **nowrap** keywords only affect the way the debugger displays output source lines. The **wrap** keyword wraps an output source line greater in length than specified by the **linesize** keyword to the next line. The **nowrap** keyword truncates the line at the line length specified by the **linesize** keyword or at the default line length of 75.

If you specify conflicting keywords referring to the same option (such as **auto noid id**), the last keyword of the pair goes into effect (in this case, **id**).

The **query** command can be used to check the settings of the **auto** command keywords.

EXAMPLES

auto linesize 100
displays 100 characters on each debugger output line.

auto echo
turns on echoing of debugger commands.

auto cmacros nullptr
substitutes macros in expressions; allows dereferencing of null pointers.

SYSTEM DEPENDENCIES

See the discussion of the **linesize** keyword earlier in this section.

COMMAND CAN BE ISSUED FROM

PROFILE	yes
configuration file	no
Source window prefix	none

SCOPE

The **auto** command is not affected by changes in scope.

RETURN CODES SET

Successful: 0

Unsuccessful: 1

SEE ALSO
- □ "query" on page 244
- □ "transfer" on page 271

break

Request a Breakpoint

ABBREVIATION
`b{reak}`

FORMATS

Format 1: **`break`**

Format 2: **`break`** HOOK-TYPE [when (EXPRESSION)] 1 [**`count`** *n*]

DESCRIPTION

The **`break`** command requests breakpoints at line-number hooks in a program.

Format 1: The **`break`** command is used without arguments only within an **`on`** command because you specify the location of breakpoints as part of the **`on`** command syntax.

Format 2: See Chapter 12, "Using Debugger Commands," on page 129 for the details of the HOOK-TYPE argument, which is used to specify line-number hooks as breakpoints.

A when clause is used to request breakpoints conditionally; that is, a breakpoint is requested at every line-number hook only if the when clause is true when the line-number hook is reached.

The argument count *n* is optional. If count *n* is specified, the first *n*–1 times the line-number hook is reached, the count is decremented. The *n*th time it is hit, the command is executed. After the *n*th time, the command is executed every time the line-number hook is hit.

If a when clause is present, a hit is counted only if the when expression is true.

A **`query`** command, issued before the count drops to 1, displays the current value of count. The keyword count can be abbreviated to cou{nt}.

Identical requests: If a **`break`** request is made that is identical to an existing one, the identical request is not installed. This is true whether the request to be installed is within an **`on`** command or typed in at the command line.

If an identical request is issued and the original request is disabled, the identical request is discarded and the original request is automatically enabled without an indication.

If count *n* is used, the count is ignored in identical requests. If an identical request with a different count is entered, the count field of a **`query`** command is updated with the new count, and a message is produced.

EXAMPLES

`break *`
breaks at every line-number hook in a source file compiled with **`debug`**.

`break entry`
breaks on entry to all functions.

`break calls`
breaks at each call to a function and at each return from a function. (In the case of function calls, program execution is interrupted twice for each function called: on calls from functions and on return to the calling function.)

break main 45 count 10
breaks at line 45 of the **main** function the tenth time the line-number hook at that line is reached. After that, it breaks every time line 45 is reached.

break 53
breaks at line 53 of the current function.

break (comp23) entry
breaks on entry to any of the functions in the **comp23** section.

break func1 entry when (parm1 ==5)
breaks on entry to the **func1** function when the value of **parm1** is 5.

break func 23:46
breaks at lines 23 through 46 of the **func** function.

SYSTEM DEPENDENCIES
none

COMMAND CAN BE ISSUED FROM

PROFILE	yes
configuration file	no
Source window prefix	b

SCOPE
The **break** command uses command scope to resolve references to all identifiers, function names, and section names.

RETURN CODES SET

- □ Successful: 1 number of the action from the list produced by the **query** command.
- □ Unsuccessful: 0

SEE ALSO

- □ “disable” on page 212
- □ “drop” on page 213
- □ “enable” on page 216
- □ “goto” on page 223
- □ “ignore” on page 226
- □ “on” on page 239
- □ “query” on page 244
- □ “runto” on page 251
- □ “trace” on page 269

browse

Request to browse a source file

ABBREVIATION
bro{wse}

FORMAT
browse [struct | union | class | enum] **name**

DESCRIPTION

The browse command is used to browse the area of the source file where the name being browsed is declared. The format of the **browse** command is as follows:

browse [**struct|union|class|enum**] **NAME**

The **NAME** argument is a single identifier name, not an expression.

The class keyword is valid only if **auto cxx** is in effect. The **cxx** keyword is set automatically whenever the debugger detects C++ translated source code.

If the optional **struct**, **union**, **class**, or **enum** keyword is not specified, the debugger performs a search in the following order:

1. the list of preprocessor symbols, if present
2. the list of identifiers, typedefs, and enumeration constants
3. the list of struct, union, enum, or class tag names.

If one of these optional keywords is specified, only the list of tag names is searched.

Normal C scope rules apply to all searches; command **scope** is used. If a declaration for the name is found, a Browse window is opened on the file and positioned to the line containing the declaration.

The **browse** command may be preceded with a **>** or **>>** command prefix: a **>** opens a new Browse window; a **>>** or no prefix reuses the most recently used Browse window or opens one if none is open.

Note: The only way to issue the **browse** command is through the Command window (or a PF key). You cannot issue the **browse** command in the Browse window. △

EXAMPLES

browse i
browse the source file where **i** is declared.

browse struct s
browse the source file where structure **s** is declared.

SYSTEM DEPENDENCIES

none

COMMAND CAN BE ISSUED FROM

PROFILE	no
configuration file	no
Source command prefix	no

SCOPE

The browse command uses command scope to resolve the name.

RETURN CODES SET

- Successful: 0
- Unsuccessful:1

catch

Request to catch all exceptions

ABBREVIATION

ca{tch}

FORMAT

catch

DESCRIPTION

The **catch** command catches all exceptions thrown in your program. When an exception is caught by the debugger, a message window is opened and will indicate the type of exception that was caught. The source will be updated to the location that had thrown the exception.

SYSTEM DEPENDENCIES

none

COMMAND CAN BE ISSUED FROM

PROFILE	yes
configuration file	no
Source command prefix	none

SCOPE

The **catch** command is not affected by changes in scope.

RETURN CODES SET

Successful: 0

Unsuccessful: 1

SEE ALSO

- □ “disable” on page 212
- □ “drop” on page 213
- □ “enable” on page 216
- □ “query” on page 244
- □ “resume” on page 246

config

Assign or Identify the Configuration File and Save the Current Configuration

ABBREVIATION

conf{ig}

FORMATS

Format 1:	**config file**
Format 2:	**config file** FILENAME | (MEMBER)
Format 3:	**config save** [FILENAME | (MEMBER)]

DESCRIPTION

The **config** command can be used to assign a configuration file, display the name of the current configuration file, or save the current configuration. The FILENAME argument is used to specify the configuration file; under CMS it is a filename and under OS/390 it is an OS/390 data set name. Under OS/390 you can specify MEMBER instead of FILENAME. MEMBER specifies a member name in a data set named **`userid`**.CDEBUG.CONFIG.

Format 1: This format is valid during a debug session; it cannot be used in the PROFILE. The **`file`** keyword displays the name of the current configuration file.

Format 2: This format is valid only in the PROFILE; it cannot be used during a debug session. The FILENAME argument is used to assign a current configuration file to your debug session, which sets the initial configuration of your windows and PF keys. If running under OS/390, the MEMBER argument can be used instead of the FILENAME argument.

Format 3: This format is used only during a debug session; it is not valid in a PROFILE. The **`save`** keyword is used with the **`config`** command to save your session configuration to a configuration file. If issued without the FILENAME or MEMBER argument, this command saves the configuration to the current configuration file. You can use the FILENAME argument, or the MEMBER argument if running under OS/390, to specify a file other than the current configuration file. This new file then becomes your current configuration file.

The following information is saved to your current configuration file:

- key definitions in use
- autopop status of windows
- border characters used for windows, if different from the default characters
- configurations of windows
- colors of windows
- context amounts of the Source window
- intercept status of the Termin window
- open status of optionally open windows
- memory allocated to various window buffers
- scroll amount that appears in the Status window
- trace status of the Log window.

EXAMPLES

`config file`
displays the name of the current configuration file. (This cannot be used in a PROFILE.)

`config file` *myconfig*
assigns *myconfig* to be the current configuration file, assuming that *myconfig* is a valid CMS configuration filename. (This is only valid in a PROFILE.)

`config file` '*userid.config.files*(*myconfig*)'
assigns *userid.config.files(myconfig)* to be the current configuration file, assuming that the member *myconfig* is a valid configuration file under OS/390. (This is only valid in a PROFILE.)

`config file` (*myconfig*)
assigns *userid.*CDEBUG.CONFIG(*myconfig*) to be the current configuration file, assuming that the member *myconfig* is a valid configuration file under OS/390. (This is only valid in a PROFILE.)

`config save`
saves the configuration of your session to your current configuration file. (This is not valid in a PROFILE.)

config save *myconfig*
saves your session configuration to *myconfig*, assuming that *myconfig* is a valid CMS configuration filename. The current configuration file becomes *myconfig*. (This is not valid in a PROFILE.)

config save
config.files(*myconfig*) saves your session configuration to the data set *userid.config.files*(*myconfig*), assuming that the member *myconfig* is a valid configuration file under OS/390. The current configuration file becomes *userid.config.files* (*myconfig*). (This is not valid in a PROFILE.)

ADDITIONAL DISCUSSION AND EXAMPLES
"Setting Up a Configuration File" on page 46

SYSTEM DEPENDENCIES
The name of the file used as the FILENAME argument depends on the operating system. See DESCRIPTION.

Under OS/390, you can specify the MEMBER argument instead of the FILENAME argument. MEMBER must refer to a member in a partitioned data set named **userid**.CDEBUG.CONFIG.

Under CMS, the **config file** command cannot be followed by another debugger command on the same line. That is, the arguments to the **config file** command are assumed to extend to the end of the line, including any semicolons on the line.

COMMAND CAN BE ISSUED FROM

PROFILE	yes (format 2 only)
configuration file	no
Source window prefix	none

SCOPE
The **config** command is not affected by changes in scope.

RETURN CODES SET

Successful: 0

Unsuccessful: 1

continue

Continue Execution to Next Line-Number Hook without Stepping into Functions

ABBREVIATION
c, **con{tinue}**

FORMAT
continue [INTEGER]

DESCRIPTION
The **continue** command resumes execution of a program and breaks at the next line-number hook in the context of the current function. However, a **continue**

command issued at a function return is identical to a **`step`** command. The debugger breaks at the epilog, first at the callee's side and then at the caller's side.

The **`continue`** command can be thought of as a **`step over`** command because, when at a line hook, it steps over function calls. If a function is recursive, either directly or indirectly, **`continue`** does not break in a recursive invocation.

INTEGER is a nonnegative integer. (INTEGER can be 0.) Use the INTEGER argument to specify the number of times you want the **`continue`** command to be performed.

The **`continue`** command does not suppress breakpoints requested by other commands (for example, **`break`**). However, the occurrence of such breakpoints does not interfere with the eventual interruption of execution as requested by **`continue`**. (At such a breakpoint, enter **`go`**.)

If you issue **`continue`** with an INTEGER argument, and the debugger breaks before the **`continue`** command is completed, you can issue it again with a different value for INTEGER to change the number of times **`continue`** is performed. Suppose that the last **`continue`** issued is **`continue 7`**, and you reach a breakpoint after **`continue`** is performed four times. If you decide you want **`continue`** to be performed only once more, issue **`continue 1`** at the breakpoint. (The three pending **`continue`** commands are replaced with one **`continue`**.) **`continue 0`** discards the three pending **`continue`** commands and causes execution to resume. The **`go`** command causes the three pending **`continue`** commands to be executed.

EXAMPLES

`continue`
resumes execution and breaks at the next line-number hook without stepping into functions.

`continue 10`
resumes execution and breaks at the tenth line-number hook without stepping into functions.

SYSTEM DEPENDENCIES
none

COMMAND CAN BE ISSUED FROM

PROFILE	no
configuration file	no
Source window prefix	none

SCOPE
The **`continue`** command is not affected by changes in scope.

RETURN CODES SET
not applicable

SEE ALSO

- □ "go" on page 222
- □ "step" on page 260

copy

Copy One or More Items to a New Location

ABBREVIATION

co{py}

FORMATS

Format 1:	**copy**	DESTINATION, "STRING"
Format 2:	**copy**	DESTINATION, "STRING", BYTES
Format 3:	**copy**	DESTINATION, [(CTYPE1 | CTYPE2)] SOURCE [,COUNT]
Format 4:	**copy**	DESTINATION, SOURCE, **str**
Format 5:	**copy**	DESTINATION, [(CTYPE2)] {LIST}

DESCRIPTION

The **copy** command copies items from one location to another. The **copy** command is used to copy a string, address, array, pointer, or one or more expressions to a location. STRING is a string literal, set off by double quotation marks. DESTINATION is a pointer, address, or array name. In the following format discussions, SOURCE and DESTINATION are expressions.

Format 1: Format 1, similar to the **strcpy** function, copies a string literal specified by the STRING argument to the location specified by the DESTINATION expression. A comma must follow the DESTINATION argument.

Format 2: Format 2, similar to the **memcpy** function, copies a string literal. The string literal to be copied is specified in the STRING argument (set off by double quotes). BYTES is an integer that indicates the number of bytes to be copied to the location specified by the DESTINATION expression.

Format 3: Format 3 copies the item or items specified by the SOURCE expression to the DESTINATION expression. DESTINATION and SOURCE can be pointers, addresses, or arrays. COUNT, an integer, is the number of items to be copied. If COUNT is omitted, the number of items defaults to 1. Each item is the size specified by either CTYPE1 or CTYPE2.

CTYPE1
: is a structure/union tag or a type defined with a **typedef** function.

CTYPE2
: is one of the following arithmetic types:

```
long    signed long    unsigned long    double
int     signed int     unsigned int     float
short   signed short   unsigned short   enum
char    signed char    unsigned char
```

If you use a left parenthesis following the DESTINATION argument, the debugger assumes that you are beginning a CTYPE1 or CTYPE2 specification.

The result of using Format 3 is analogous to the following:

```
memcpy(DESTINATION,SOURCE,((sizeof (CTYPE1 or CTYPE2)) * COUNT)
```

Note that if you omit the CTYPE1 or CTYPE2 argument, the sizes of the expressions that SOURCE and DESTINATION point to must be the same. If the sizes are not the same, you receive a message, and the **copy** command is not performed.

CTYPE1 overrides the declared types of both the DESTINATION and SOURCE arguments. If the SOURCE or DESTINATION argument does not have a specific type, the type defaults to **char**. For absolute addresses, the default type is **char** and the size of the pointed-to expression is 1.

Format 4: Format 4 copies the contents of a memory location pointed to by SOURCE to the location specified by the DESTINATION expression until the null terminator **\ 0** is encountered in the SOURCE expression. The keyword **str** specifies a string copy that is similar to **strcpy**. SOURCE is a pointer, address, or array. The string delimiter **\ 0** is the last byte copied.

If the SOURCE argument begins with a left parenthesis, the parenthesis must be escaped with a backslash (\). If you do not escape the parenthesis, the parenthesis is assumed to begin a CTYPE*x*. Here is an example:

copy a+5, \ (b+2)

Format 5: Format 5 converts the items specified by LIST to CTYPE2 format and stores the values at the destination specified by the DESTINATION expression. LIST is one or more expressions. (If you use more than one, separate them with commas.)

DESTINATION is a pointer, address, or array name. CTYPE2 is any of the arithmetic types, as listed for Format 4. If you do not specify CTYPE2, the type of the object pointed to by the DESTINATION expression is the default.

EXAMPLES

The **copy** command examples are based on the following declarations:

```
char *cp, *s, *d;
struct XYZ {int a; double b;} xyz, xarr[5] , yarr[5] ;
int intarr[5] ;
```

copy cp, ''abcd''
copies the string **abcd** to the location pointed to by **cp**.

copy cp, ''abcd'', 4
copies 4 bytes (**abcd**) into the location pointed to by **cp** (the null terminator **\ 0** is not copied).

copy xarr, yarr, 5
copies the five elements of the array **yarr** into the array **xarr**.

copy &xyz, &xarr[2]
copies **xarr[2]** into **xyz.**

copy d, s, 10
copies 10 bytes from the location pointed to by **s** to the location pointed to by **d**.

copy d, s, str
copies the string pointed to by **s** into **d**.

copy intarr (int) {10,20,30,40,50}
copies the values 10, 20, 30, 40, 50 as integers into the array **intarr**.

SYSTEM DEPENDENCIES

none

COMMAND CAN BE ISSUED FROM

PROFILE	no
configuration file	no
Source window prefix	none

SCOPE
The **copy** command uses command scope to resolve references to all identifiers.

RETURN CODES SET
Successful: 0
Unsuccessful: 1

SEE ALSO
- □ “assign” on page 192
- □ “return” on page 248
- □ “scope” on page 252

dbinput

Called from a CLIST or EXEC for Information Input

ABBREVIATION
dbi{nput}

FORMAT
dbinput VARIABLE-NAME [“STRING”]

DESCRIPTION
The **dbinput** command can be called from a CLIST or an EXEC to input information into a CLIST or EXEC variable. This command, which can only be used in a CLIST or EXEC, causes the debugger to prompt for input using the prompt string specified with the STRING argument. STRING is a group of characters set off by double quotation marks. The command undergoes normal CLIST or EXEC processing, that is, symbolic substitution occurs. Text input is assigned to the CLIST or EXEC variable name specified in the command. Both the VARIABLE-NAME and STRING arguments are limited to 64 characters in length.

In line mode, the debugger reads the input from the terminal at the prompt specified by the STRING argument.

In full-screen mode, the Termin window is used. Instead of displaying a **Read**... prompt, the Termin window will display **Exec**... in the upper-left border. The characters specified by the STRING argument are displayed as the input prompt. Setting the EOF field to Y results in the condition code being set to –13. The **Intercept:** field has no effect on the **dbinput** command. The functions of other Termin window fields are unchanged by the **dbinput** command. If it is not possible to open the Termin window, the debugger will accept input from the terminal, in a similar manner to a line mode session.

EXAMPLES

dbinput *xyz*
reads terminal input into the EXEC or CLIST variable **xyz**.

dbinput *xyz* **''Please Enter Y or N:''**
displays the prompt string and reads terminal input into the variable *xyz*. In full-screen mode the prompt string is displayed in the Termin window. See "Termin Window" on page 179 for information about the Termin window.

SYSTEM DEPENDENCIES
The **dbinput** command is used to input a value to a CLIST or EXEC under TSO and an EXEC under CMS.

COMMAND CAN BE ISSUED FROM

PROFILE	no
configuration file	no
Source window prefix	none

SCOPE
The **dbinput** command is not affected by changes in scope.

RETURN CODES SET

Successful: 0

Unsuccessful: 1

SEE ALSO
"dblog" on page 209

dblog

Called from a CLIST or EXEC to Output Information

ABBREVIATION
dbl{og}

FORMAT
dblog [STRING]

DESCRIPTION
The **dblog** command can be called from a CLIST or EXEC to output information to the debugger. STRING is a string of characters that is not surrounded by quotation marks. If **dblog** is used in your CLIST or EXEC, it outputs the string of characters specified by the STRING argument. If used without the STRING argument, the output is a blank line. In line mode, output is sent to the session log displayed on the terminal; in full-screen mode, it is sent to the Log window. If sent to the Log window, it is displayed in the color specified for debugger commands (controlled by the Config window).

The **dblog** command cannot be followed by another debugger command on the same line. That is, the arguments to the **dblog** command are assumed to extend to the end of the line, including any semicolons on the line.

EXAMPLES

dblog
outputs a blank line to the Log window.

`dblog This is an output string.`
displays the string "This is an output string." in the Log window.

SYSTEM DEPENDENCIES
The **`dblog`** command is used to output information from a CLIST or EXEC under TSO and from an EXEC under CMS.

COMMAND CAN BE ISSUED FROM

PROFILE	no
configuration file	no
Source window prefix	none

SCOPE
The **`dblog`** command is not affected by changes in scope.

RETURN CODES SET
Successful: 0
Unsuccessful: 1

SEE ALSO
"dbinput" on page 208

define

Define a Debugger Macro

ABBREVIATION
`de{fine}`

FORMATS

Format 1:	**`define`** DMACRO REPLACEMENT TEXT
Format 2:	**`define`** DMACRO
Format 3:	**`define`** *

DESCRIPTION
The **`define`** command defines debugger macros, which you should not confuse with macros that are defined in a program with the preprocessor **`#define`** statement. Debugger macros are a way of defining a shorthand version for commands or

portions of a command that you plan to use often in a debugger session. SAS/C Debugger macros are invoked by prefixing the macro name with #.

Format 1: Format 1 defines a debugger macro (DMACRO) that can be any valid identifier. The REPLACEMENT TEXT, to be used when invoked, is also specified.

Format 2: Format 2 lists the replacement text for DMACRO.

Format 3: Format 3 lists all debugger macros that you have defined. Format 2 and Format 3 are helpful if you need to view your macro definitions.

Macro definitions can be dropped using the **`undef`** command. A macro can be redefined by issuing another **`define`** for the same macro, but with new substitution text. You do not need to use the **`undef`** command on a macro before redefining it.

Debugger macros cannot appear in the following commands:

- **`%`**
- **`config`**
- **`exec`**
- **`help`**
- **`system`**
- **`transfer`**
- user-installed commands.

EXAMPLES

`define pt print ptr --> token`
defines **`pt`** to be a debugger macro so that typing **`# pt`** at the Cdebug prompt is equivalent to typing **`print --> token.`**

Note: You also can type **`# pt.shrt`** to obtain **`print ptr --> token.shrt`** or **`# pt.lng`** to obtain **`print ptr --> token.lng`**. △

`define pt`
displays the text that **`pt`** replaces.

`define` *
displays all the debugger macros that you defined.

SYSTEM DEPENDENCIES
none

COMMAND CAN BE ISSUED FROM

PROFILE	yes
configuration file	no
Source window prefix	none

SCOPE
The **`define`** command is not affected by changes in scope.

RETURN CODES SET

Successful: 0

Unsuccessful: 1

SEE ALSO
“undef” on page 275

disable

Disable Requests

ABBREVIATION

`di{sable}`

FORMATS

Format 1:	**`disable`** ACTION-RANGE
Format 2:	**`disable`** ACTION-RANGE, ACTION-RANGE, . . .
Format 3:	**`disable`** FUNCTION-NAME | (SECTION-NAME) all
Format 4:	**`disable`** all
Format 5:	**`disable`** last

DESCRIPTION

The **`disable`** command disables requests. Requests are identified by request number as displayed by the **`query`** command. The ACTION-RANGE argument is either ACTION (a single request) or ACTION:ACTION (a request range).

Format 1: Format 1 disables one request or one request range specified by ACTION-RANGE.

Format 2: Format 2 disables several single requests and/or request ranges.

Format 3: Format 3 disables all requests for a section (SECTION-NAME) or function (FUNCTION-NAME).

To use this form of **`disable`**, specify the function name or, in parentheses, the section name followed by the keyword **`all`**.

Format 4: This format disables all requests for the entire program.

Format 5: This format disables the last request on the list.

After you disable a request, the request is not honored until you enable it again. (See “enable” on page 216.) Disabled requests are marked with an asterisk in the query list next to the request number. (If you try to disable a request that is already disabled, you receive a message.)

In contrast to **`drop`**, which permanently removes requests from the query list, **`disable`** only makes the command ineffective until you reenable it.

Identical requests: If you issue a request, disable it, and then issue an identical request, the identical request is discarded and the original request is automatically enabled.

EXAMPLES

`disable 3`
disables request number 3 in the query list.

`disable 3:6`
disables request numbers 3 through 6 in the query list.

`disable 4, 9:13, 7`
disables request number 4, request numbers 9 through 13, and request number 7 in the query list.

`disable func1 all`
disables all requests for the **`func1`** function.

`disable (comp23) all`
disables all requests for the **`comp23`** section name (compilation).

`disable all`
disables all requests for the program.

`disable last`
disables the last request on the list.

SYSTEM DEPENDENCIES
none

COMMAND CAN BE ISSUED FROM

PROFILE	yes
configuration file	no
Source window prefix	**`d`**

SCOPE
The **`disable`** command is not affected by changes in scope.

RETURN CODES SET

Successful: a nonzero number

Unsuccessful: 0

SEE ALSO

- □ “dump” on page 215
- □ “enable” on page 216
- □ “ignore” on page 226
- □ “query” on page 244

drop

Drop Request

ABBREVIATION
`dr{op}`

FORMATS

Format 1:	**`drop`** ACTION-RANGE
Format 2:	**`drop`** ACTION-RANGE, ACTION-RANGE, . . .
Format 3:	**`drop`** FUNCTION-NAME | (SECTION-NAME) **`all`**
Format 4:	**`drop all`**
Format 5:	**`drop last`**

DESCRIPTION
The **`drop`** command drops one or more requests from the list displayed by the **`query`** command. Requests are identified in the query list by request number. The numbers associated with dropped requests are not reused. The ACTION-RANGE

argument is either ACTION (a single request) or ACTION:ACTION (a range of requests).

Format 1: Format 1 drops a single request or request range from the list displayed by the **`query`** command.

Format 2: Format 2 drops several single requests and request ranges, which are separated by commas.

Format 3: Format 3 drops all requests for a function (FUNCTION-NAME) or a section (SECTION-NAME) from the query list.

To use this form of **`drop`**, specify the function name or, in parentheses, the section name followed by the word all.

Format 4: This format drops all requests for the entire program.

Format 5: This format drops the last request on the list displayed by the **`query`** command.

EXAMPLES

`drop 3`
drops request number 3 from the query list.

`drop 3:6`
drops request numbers 3 through 6 from the query list.

`drop 4, 9:13, 7`
drops request numbers 4, 9 through 13, and 7 from the query list.

`drop func1 all`
drops all the requests for the **`func1`** function.

`drop (comp23) all`
drops all the requests for the **`comp23`** compilation.

`drop all`
drops all the requests for the entire program.

`drop last`
drops the last request on the query list.

SYSTEM DEPENDENCIES
none

COMMAND CAN BE ISSUED FROM

PROFILE	yes
configuration file	no
Source window prefix	none

SCOPE
The **`drop`** command is not affected by changes in scope.

RETURN CODES SET

Successful: a nonzero number

Unsuccessful: 0

SEE ALSO

- □ “disable” on page 212
- □ “enable” on page 216
- □ “ignore” on page 226
- □ “query” on page 244

dump

Dump Memory Contents

ABBREVIATION
du{mp}

FORMATS

Format 1:	**dump** EXPRESSION relative/absolute
Format 2:	**dump** EXPRESSION COUNT relative/absolute
Format 3:	**dump** EXPRESSION relative/absolute **str**

DESCRIPTION

The **dump** command dumps the contents of storage pointed to by EXPRESSION. Note that for the **dump** command, EXPRESSION is either a pointer, an address, or an array.

Format 1: Format 1 dumps the contents of storage pointed to by the argument EXPRESSION with the number of bytes dumped determined as shown in Table 14.1 on page 215.

Table 14.1 Dump Command: Number of Bytes Dumped

Argument Type	Number of Bytes Dumped
pointer	the size of the pointed-to expression
address of a scalar or aggregate	the size of one item of the array
array	the size of one item of the array
absolute address	1 (treated as a pointer to char)

Format 2: Format 2 dumps the contents of storage pointed to by EXPRESSION up to the number of bytes specified by COUNT. COUNT is an integer that specifies the number of bytes to be dumped.

Format 3: Format 3 dumps the contents of storage associated with EXPRESSION until the null terminator **\ 0** is encountered.

For all formats, the output of the **dump** command shows the contents of the EXPRESSION argument in characters and in hexadecimal format and shows the address of the argument as a hexadecimal number. The keyword relative or absolute selects the type of addressing used to display the address. The default is to display relative addresses.

The output of the **dump** command is affected by the width of the output area. In line mode, the linesize specified by the **auto** command will affect the width of the output. In full-screen mode, the width of the window in which the dump is displayed affects the width of the output. (Output from the **dump** command can be displayed in either the Log or Dump window.)

EXAMPLES

The **dump** command examples are based on the following declarations:

```
char *s;
struct INT2 {int a, b} int2;
```

dump s
dumps one character of storage pointed to by **s**.

dump &int2
dumps the 8 bytes (size of structure **int2**) of structure **int2**.

dump s 10
dumps 10 bytes of storage beginning at the location pointed to by **s**.

dump 0p00234567 20
dumps 20 bytes of storage beginning at the absolute address 0p00234567.

dump s str
dumps the string pointed to by **s**.

SYSTEM DEPENDENCIES
none

COMMAND CAN BE ISSUED FROM

PROFILE	no
configuration file	no
Source window prefix	none

SCOPE
The **dump** command uses command scope to resolve references to all identifiers.

RETURN CODES SET
Successful: 0
Unsuccessful: 1

SEE ALSO
- “monitor” on page 235
- “print” on page 241
- “watch” on page 276

enable

Enable Requests

ABBREVIATION
en{able}

FORMATS

Format 1:	**enable** ACTION-RANGE
Format 2:	**enable** ACTION-RANGE, ACTION-RANGE, . . .
Format 3:	**enable** FUNCTION-NAME/(SECTION-NAME) **all**
Format 4:	**enable all**
Format 5:	**enable last**

DESCRIPTION

The **`enable`** command reenables commands that were disabled previously. (See "disable" on page 212.) Requests are identified by request number as displayed by the **`query`** command. The ACTION-RANGE argument is either ACTION (a single request) or ACTION:ACTION (a range of requests).

Format 1: Format 1 enables one request or one request range specified by ACTION-RANGE.

Format 2: Format 2 enables several single requests or request ranges or both, which are separated by commas.

Format 3: Format 3 enables all requests for a section (SECTION-NAME) or function (FUNCTION-NAME). To use this form of **`enable`**, specify the function name or, in parentheses, the section name, followed by the word all.

Format 4: Format 4 enables all requests (for the entire program). If you specify only the **`all`** keyword, all disabled requests (for the entire program) are enabled.

Format 5: Format 5 enables the last request on the list.

Identical requests: If you issue a request, disable it, and then issue an identical request, the identical request is discarded and the original request is automatically enabled without an indication.

EXAMPLES

`enable 3`
enables request number 3 in the query list that was disabled previously.

`enable 3:6`
enables request numbers 3 through 6 in the query list.

`enable 4, 9:13, 7`
enables request numbers 4, 9 through 13, and 7 in the query list.

`enable func1 all`
enables all the previously disabled requests for the **`func1`** function.

`enable (comp23) all`
enables all the previously disabled requests for the compilation **`comp23`**.

`enable all`
enables all previously disabled requests for the entire program.

SYSTEM DEPENDENCIES

none

COMMAND CAN BE ISSUED FROM

PROFILE	yes
configuration file	no
Source window prefix	e

SCOPE

The **`enable`** command is not affected by changes in scope.

RETURN CODES SET

Successful: a nonzero number

Unsuccessful: 0

SEE ALSO

- “disable” on page 212
- “ignore” on page 226
- “query” on page 244

escape

Transfer Control to the Operating System Debugger

ABBREVIATION

es{cape}

FORMAT

escape

DESCRIPTION

The **escape** command transfers control from the SAS/C Debugger to TEST under TSO or to CP under CMS and lets you return to the program executing under the SAS/C Debugger at the place where you left it. You enter TEST or CP in the context of the program being debugged. You need to perform the following steps before you can use **escape**:

1. When you enter the debugger, press the attention key to transfer control to CP or TSO TEST.
2. Using commands for your operating system, request the breakpoint specified on entry to the SAS/C Debugger. This breakpoint is stated in the header you see when you call the SAS/C Debugger.
3. After you request this breakpoint (using commands for your operating system), reenter the SAS/C Debugger (also using commands for your operating system).

When you issue the **escape** command while executing your program under the SAS/C Debugger, you escape to TSO TEST or CP. The screen is refreshed after completion of execution of the **escape** command.

If you do not request the breakpoint for **escape** according to the instructions and try to use **escape**, nothing happens. The debugger cannot check whether you requested the breakpoint.

In addition, you do not need to request the breakpoint immediately on entry to the debugger. You can note the address and use it later to request the breakpoint when you want to use **escape**.

SYSTEM DEPENDENCIES

TSO: To use **escape**, you must run the SAS/C Debugger under TSO TEST.

1. Invoke the SAS/C Debugger using TSO TEST. (See Chapter 5, “Running the Debugger under TSO,” on page 61.) If you mistakenly invoke the debugger using the TSO CALL command, you cannot run your program under TEST.
2. Issue the TEST command GO to begin SAS/C Debugger execution. Read the ESCAPE message.
3. Press the attention key to return to TEST under TSO.
4. Request the breakpoint using the TEST command AT.
5. Issue the TEST command GO to return to the SAS/C Debugger. See the IBM publication *OS/390 V2R9.0 TSO/E Command Reference SC28–1969* for more information about TSO TEST commands.

CMS: When you are running the SAS/C Debugger under CMS, use the debugger command **system** to issue a PER command to request a breakpoint.

(Under VM/XA, the TRACE command is used instead of PER.) For example, if the breakpoint that you want to request is at 00D174, the following command requests the breakpoint:

```
system CP PER I R D174
```

Then, you can use **escape** to leave the SAS/C Debugger, examine your program under CP PER, and return to the SAS/C Debugger via the CP command BEGIN. See the IBM publication CP Command and Utility Reference SC24–5773 for more information about CP.

Register values: TSO and CMS: When you use **escape** to give control to TEST or PER, the register contents are the same as when the program transfers control to the debugger.

- For a normal line-number hook, register 1 points to the next instruction to be executed.
- For a return hook, register 14 contains the return address and register 15 contains the return value, if any, for integer and pointer type returns. Floating-point register 0 contains double return values.
- When control passes to the debugger due to a program check, register 1 addresses a copy of the EPIE generated by the system for the program check, which includes the register contents and PSW.

Modifying register contents while escaped under TEST or PER may not work and could potentially cause 0CX or other undesired effects. See “return” on page 248 for information about setting RETURN values.

COMMAND CAN BE ISSUED FROM

PROFILE	no
configuration file	no
Source window prefix	none

SCOPE

The **escape** command is not affected by changes in scope.

RETURN CODES SET

none

SEE ALSO

“system (CMS)” on page 267

exec (TSO)

Execute a CLIST or EXEC under TSO

ABBREVIATION

exe{c}

FORMAT

exec CLIST-NAME | EXEC-NAME [’ARGUMENTS’] [**list | nolist**] [**prompt | noprompt**]

DESCRIPTION

The **exec** command executes a CLIST or a REXX EXEC specified by the CLIST-NAME or EXEC-NAME argument. CLIST-NAME or EXEC-NAME is the name of a member in a partitioned data set containing command procedures (CLISTs) or a REXX EXEC.

The data set can contain either CLIST or REXX statements and control variables as well as SAS/C Debugger commands. Debugger commands are passed back to the debugger and performed. TSO commands cannot be executed directly from a CLIST. However, they can be executed using the **system** debugger command.

If you execute a REXX EXEC from the debugger, you can issue either TSO commands or debugger commands with the REXX ADDRESS. ADDRESS CDEBUG is used to issue debugger commands and ADDRESS TSO is used to issue TSO commands.

You can specify values for ARGUMENTS to be passed to your CLIST or EXEC, and you can specify the list, nolist, prompt, or noprompt keyword. See the IBM publication OS/390 V2R9.0 TSO/E Command Reference SC28-1969 for a discussion of the TSO EXEC command and the list, nolist, prompt, and noprompt keywords.

The data set name must follow TSO naming conventions for data sets containing CLISTs or EXECs in a TSO EXEC command. According to these conventions, if the final qualifier of the CLIST is not "clist," the fully qualified data set name must be specified inside single quotation marks. Line numbers (if they exist) in the data set must follow these rules:

fixed block data set
: In each record, the line number is the last eight characters.

variable blocked data set
: In each record, the line number is the first eight characters.

See the IBM publication TSO/E Command Reference SC28-1969 for a discussion of CLISTs. See the IBM publication OS/390 V2R9.0 TSO/E REXX User's Guide SC28-1974 for information about REXX EXECs.

You cannot use another debugger command on the same line following **exec**. If another debugger command occurs on the same line after **exec**, the debugger ignores it with a warning.

Similarly, no other debugger command can be issued on the same line as **exec** when it is used as an argument to **on** (but you can use another command following **exec** on a separate line). Any command following **exec** on the same line is ignored.

If a CLIST or EXEC contains a debugger **go**, **step**, **continue**, **exit**, **abort**, or **resume** command, the command is executed and the CLIST or EXEC is ended. Any debugger commands in the CLIST or EXEC following one of these commands are not performed. In an EXEC, any commands after one of these commands are rejected with a return code of –10.

ADDITIONAL DISCUSSION AND EXAMPLES

See Table 3.2 on page 51.

SYSTEM DEPENDENCIES

The **exec** command followed by a CLIST argument is only valid for TSO. See the CMS version of **exec**.

COMMAND CAN BE ISSUED FROM

PROFILE	yes
configuration file	no
Source window prefix	none

SCOPE
The **exec** command is not affected by changes in scope.

RETURN CODES SET

Successful: code set by the CLIST called

Unsuccessful: parsing error, – 1; otherwise, **execopn()** code

exec (CMS)

Execute an EXEC under CMS

ABBREVIATION
exe{c}

FORMAT
exec EXEC-NAME [ARGUMENTS]

DESCRIPTION
The **exec** command executes the EXEC specified by the argument EXEC-NAME. This file must have file type CDEBUG. The EXEC is executed with a default subcommand environment of CDEBUG. See the IBM publication VM/ESA V2R4.0 REXX/VM Reference SC24-5770 for a detailed explanation of subcommand environments.

You cannot use another debugger command on the same line following **exec**. If another debugger command occurs on the same line as **exec**, the debugger ignores it with a warning. Similarly, no other debugger command can be issued on the same line after **exec** when it is used as an argument to **on**. You can use another command following **exec** on a separate line.

ADDITIONAL DISCUSSION AND EXAMPLES
See Table 3.2 on page 51.

SYSTEM DEPENDENCIES
The **exec** command followed by an EXEC argument is valid only for CMS. See the TSO version of **exec**.

COMMAND CAN BE ISSUED FROM

PROFILE	yes
configuration file	no
Source window prefix	none

SCOPE

The **exec** command is not affected by changes in scope.

RETURN CODES SET

Successful: code returned from the EXEC

Unsuccessful: parsing error, – 1; otherwise, **execopn()** code

exit

Terminate Program Execution

ABBREVIATION

exi{t}

FORMATS

Format 1:	**exit**
Format 2:	**exit** nodrop

DESCRIPTION

Format 1: Format 1 immediately terminates program execution under the debugger, closing both program files and debugger files. Control returns to the operating system. The **exit** command is equivalent to calling the **exit** function from within a program.

Format 2: Format 2 uses the **nodrop** keyword in the debugger **exit** command to retain breakpoints in functions registered with the **atexit** compiler function. Normally, the **exit** command drops any outstanding breakpoints before terminating the program. If the program registers a function via the **atexit** function, any breakpoints in that function are dropped. When the **nodrop** keyword is used, however, outstanding breakpoints are not dropped automatically.

SYSTEM DEPENDENCIES

none

SCOPE

The **exit** command uses command scope to resolve references to all identifiers.

COMMAND CAN BE ISSUED FROM

PROFILE	no
configuration	no
Source window prefix	none

RETURN CODES SET

not applicable

SEE ALSO

“abort” on page 191

go

Start/Resume Program Execution under the Debugger

ABBREVIATION

`g{o}`

FORMAT

`go`

DESCRIPTION

The **`go`** command starts (or resumes) execution of a program under the SAS/C Debugger. The program executes until the first breakpoint or action that is requested is reached or until an incomplete **`step`** or **`continue`** completes. Then, the **`go`** command can be reissued to resume execution.

Note: If you do issue **`go`** as the first command, the debugger regains control if one of the signals trapped by the debugger is raised or if attention/IC is used. See “resume” on page 246 for a list of these signals. △

You can use the **`go`** command to debug a program that fails. For example, suppose that your program fails with an 0C4. To run the program, issue a **`go`** command. When the 0C4 signal occurs and the debugger regains control, you can look at variables.

SYSTEM DEPENDENCIES

none

COMMAND CAN BE ISSUED FROM

PROFILE	no
configuration file	no
Source window prefix	none

SCOPE

The **`go`** command is not affected by changes in scope.

RETURN CODES SET

not applicable

SEE ALSO

- “continue” on page 204
- “goto” on page 223
- “resume” on page 246
- “runto” on page 251
- “step” on page 260

goto

Alias for resume Command; Resume Program Execution at a Specified Location

ABBREVIATION

`got{o}`

FORMATS

Format 1: **`goto`**

Format 2:	**`goto`** LINENO
Format 3:	**`goto`** FUNCTION-NAME
Format 4:	**`goto`** FUNCTION-NAME LINENO

DESCRIPTION

All formats of the **`goto`** command allow you to resume execution at a specified location. Thus, you can reexecute or bypass portions of your program.

The **`goto`** command is an alias for the **`resume`** command. See "resume" on page 246 for a complete description.

SYSTEM DEPENDENCIES

none

COMMAND CAN BE ISSUED FROM

PROFILE	no
configuration file	no
Source window prefix	**`g`**

SCOPE

The **`goto`** command uses command scope to supply default function names.

RETURN CODES SET

not applicable

SEE ALSO

"resume" on page 246

help

Access Debugger Online Help

ABBREVIATION

`h{elp}`

FORMAT

`help` [DEBUGGER-CMD-NAME | WINDOW-NAME | TOPIC]

DESCRIPTION

The **`help`** command invokes the help system. The optional arguments, DEBUGGER-CMD-NAME, WINDOW-NAME, and TOPIC, are used to access information debugger commands, windows, and topics such as how to set up a configuration file. Context-sensitive help is provided by the **`help`** < > command, which is assigned to the PF1 key by default. The key currently assigned to this command is displayed on the left side of the Status window.

Enter the **`help`** command with no arguments to access an index of the help system. The index is used to select help on topics such as prefix-area commands and debugger windows.

The DEBUGGER-CMD-NAME argument is used to specify the command you want help with. DEBUGGER-CMD-NAME can be any debugger command or its abbreviation.

The WINDOW-NAME argument can be used to access help for a specific window. The WINDOW-NAME argument must be one of the following:

Browse	Command	Config
Dump	Find	Help
Keys	Log	Message
Popup	Print	Register
Source	Status	Termin
Termout	Watch	

The TOPIC argument is used to access a specific topic in the help system. Here are some of the topics that are covered:

- □ assigning PF keys
- □ closing windows
- □ configuration file
- □ directing command output
- □ executing EXECs
- □ formats
- □ issuing CMS CP commands
- □ issuing operating system commands
- □ issuing TSO TEST commands
- □ moving windows
- □ opening windows
- □ prefix-area commands
- □ PROFILE
- □ resizing windows
- □ > command prefix
- □ >> command prefix
- □ < > placeholder.

Any of these topics can be entered as a TOPIC argument to the **help** command. For example, entering the following command will access help on closing windows:

```
help closing windows
```

The debugger converts all TOPIC arguments to uppercase before searching the help system for the topic. The search routine used by the help system to find information about your topic is not case sensitive; therefore, the TOPIC argument can be any combination of uppercase and lowercase characters.

The **help** command cannot be followed by another debugger command on the same line. That is, the arguments to the **help** command are assumed to extend to the end of the line, including any semicolons on the line.

EXAMPLES

help
opens the Help window and displays the INDEX.

help break
opens the Help window and displays the card for the **break** command.

`help config window`
opens the Help window and displays the card for the Configuration window.

`help prefix-area commands`
uses the TOPIC argument to display information about prefix-area commands.

SYSTEM DEPENDENCIES
none

COMMAND CAN BE ISSUED FROM

PROFILE	yes
configuration file	no
Source window prefix	none

SCOPE
The **`help`** command is not affected by changes in scope.

RETURN CODES SET
Successful: 0
Unsuccessful: 1

ignore

Ignore Breakpoint or Action Requests, or Signals

ABBREVIATION
`i{gnore}`

FORMATS

Format 1:	**`ignore`** HOOK-TYPE
Format 2:	**`ignore`** signal
Format 3:	**`ignore`** SIGNAL-NAME signal

DESCRIPTION
The **`ignore`** command lets you temporarily suppress action and breakpoint requests that you may want to reinstate later. It also lets you ignore signals.

Note: It is better to drop a request than to ignore it because run-time overhead may be associated with ignored requests, but not with dropped ones. △

Format 1: Format 1 lets you ignore breakpoint or action requests at specified hooks. See Chapter 12, “Using Debugger Commands,” on page 129 for details on the HOOK-TYPE argument, an argument that allows you to specify hooks.

Format 2: Format 2 causes the debugger to ignore all signals.

Format 3: Format 3 causes the specific signal named in SIGNAL-NAME to be ignored when SIGNAL-NAME is used with the signal keyword. SIGNAL-NAME is the name of a signal. Tracing of the signal is turned off. See Chapter 17, “Signal-Handling Functions,” in the SAS/C Library Reference, Volume 1, for a list of signal names.

Ignoring signals: Format 2 and Format 3 allow you to ignore signals. Whenever a signal occurs, it is traced automatically, and the return from signal handling is traced also. The **ignore** command provides a way to turn off this trace. If you choose to ignore a signal, the debugger does not trap or trace the signal, and you cannot recover or resume execution afterward. You must remember this point when you use **ignore** with the signal keyword.

Dropping ignore: You can drop the **ignore** command (using the **drop** command) if you want the breakpoint requests and actions to be in effect again or if you do not want to ignore signals.

Identical requests: If an **ignore** request is made that is identical to an existing one, the identical request is not installed. This is true whether the request to be installed is within an **on** command or typed in at the command line.

If an identical request is issued and the original request is disabled, the identical request is discarded and the original request is automatically enabled without an indication.

EXAMPLES

ignore *
: ignores all breakpoint and action requests at every line-hook for every source file.

ignore entry
: ignores requests at all function entries.

ignore main 45
: ignores any requests at line 45 in the **main** function.

ignore 53
: ignores any requests at line 53 of the current function.

ignore (comp23) entry
: ignores any requests made at entry to the **comp23** compilation.

ignore func1 entry
: ignores any requests made at entry to the **func1** function.

ignore func 23:46
: ignores any requests at lines 23 through 46 in the **func** function.

ignore signal
: ignores all signals.

ignore SIGSEGV signal
: ignores the signal named SIGSEGV.

SYSTEM DEPENDENCIES

none

COMMAND CAN BE ISSUED FROM

PROFILE	yes
configuration file	no
Source window prefix	**i**

SCOPE

The **ignore** command uses command scope to supply default function names and section names.

RETURN CODES SET

Successful: request number from **`query`**

Unsuccessful : 0

SEE ALSO

- □ "disable" on page 212
- □ "drop" on page 213
- □ "enable" on page 216
- □ "query" on page 244

install

Assign or List User-Defined Commands

ABBREVIATION

`in{stall}`

FORMATS

Format 1:	**`install`** COMMAND-NAME as EXEC-NAME 2 [LENGTH]
Format 2:	**`install`** COMMAND-NAME drop
Format 3:	**`install`** *

DESCRIPTION

The **`install`** command enables you to define commands, thus, tailoring the debugger to meet your specific need or to emulate the commands of other debuggers. Your commands execute a CLIST or EXEC, which actually contains the code for your user-defined command. No checking for the existence of the CLIST or EXEC is made at the time the **`install`** command is issued.

Format 1: This format of the **`install`** command is used to define a command. When defining a command, the COMMAND-NAME argument indicates the name of your command and the EXEC-NAME argument is the name of a CLIST or EXEC that is executed when your new command is invoked.

The LENGTH argument is used to specify a short form for your command. If specified, LENGTH is the minimum number of characters in COMMAND-NAME that can be used as a short form of the command. For example, if you installed a command name **`start`**, specifying 2 as the LENGTH argument, your command could be invoked by entering the characters **`st`**, **`sta`**, **`star`**, or **`start`**.

Format 2: This format is used to remove a command from the debugger's list of user-defined commands.

Format 3: This format lists all user-defined commands. The COMMAND-NAME, EXEC-NAME, and LENGTH are displayed.

User-defined commands are placed in a list that is searched before the debugger's command list. If you attempt to use the **`install`** command to define a command that creates a conflict with an existing user-defined command, a message is displayed and the definition is not made.

It is possible for user-defined commands to have the same name or short form as debugger commands. If you create a user-defined command with the same name as a debugger command, you can invoke the native debugger command by prefixing the command with the debugger escape character, a backslash (\).

User-defined commands can invoke other user-defined commands; however, the debugger makes no attempt to identify user-created loops.

When you invoke a user-defined command, you can pass arguments to your CLIST or EXEC by typing them on the same line following your command. For example, suppose that you have defined a command that invokes a CLIST named MYCOMMAND that takes two arguments, ARG1 and ARG2. This user-defined command can be issued from the command line or the Command window as follows:

```
mycommand arg1 arg2
```

As is the case with the **exec** and **%** commands, no other commands can follow a user command on the same line. However, a command can precede the user-defined command. Similarly, no other debugger command can be issued on the same line as the user-defined command when it is used as an argument to the **on** command (but you can use another command following the user-defined command on a separate line). Any commands following the user-defined command on the same line are treated as arguments to the user-defined command.

EXAMPLES

install mycommand as myexec 2
defines a command named **mycommand** that will invoke a CLIST or EXEC named MYEXEC. The **2** establishes **my** as the short form for this command.

install mycommand drop
removes, or drops, **mycommand** from the debugger's list of user-defined commands.

install *
displays a list of all user-defined commands.

SYSTEM DEPENDENCIES

TSO: Your user-defined command invokes a CLIST or REXX EXEC as previously described for the **%** command. When you invoke your user-defined command, it is executed in the same manner as it would be if you had invoked the CLIST or EXEC with the **%** command. All restrictions that apply to CLISTs and EXECs executed by the **%** command also apply to CLISTs and EXECs executed by user-defined commands.

CMS: The user-defined command executes the EXEC specified by the argument EXEC-NAME. This file must have filetype CDEBUG. The EXEC is executed with a default subcommand environment of CDEBUG. See the IBM publication VM/ESA V2R4.0 REXX/VM Reference SC24-5770 for a detailed explanation of subcommand environments.

When you invoke your user-defined command, it is executed in the same manner as it would be if you had invoked the EXEC with the **exec** command. All restrictions that apply to EXECs executed by the **exec** command also apply to EXECs executed by user-defined commands.

COMMAND CAN BE ISSUED FROM

PROFILE	yes
configuration file	no
Source window prefix	none

SCOPE

The **install** command is not affected by changes in scope.

RETURN CODES SET

Successful: 0

Unsuccessful: 1

SEE ALSO

- "%" on page 189
- "exec (TSO)" on page 219

keys

Assign or List PF Key Commands

ABBREVIATION

k{eys}

FORMATS

Format 1:	**keys** [**list** N]
Format 2:	**keys define** N "STRING"
Format 3:	**keys default** N
Format 4:	**keys help** N
Format 5:	**keys ispf** N **on|off**

DESCRIPTION

PF keys can be used to issue debugger commands in full-screen mode. The **keys** command is used to list or modify PF key assignments. You can issue the **keys** command from the command line, the command window, or a configuration file; however, it is not valid in a PROFILE.

Format 1: This format lists the PF key assignments. Issuing the command without any arguments will display all PF key assignments. If issued with the optional list keyword, the **keys** command will display the command assigned to the PF key specified by the N argument. Using an asterisk (*) as the N argument also displays all PF key assignments.

Format 2: In the configuration file, this format changes the definition of the PF key indicated by the N argument to the command specified by the STRING argument. This becomes the session's initial assignment for that PF key, which is copied to a table used by the debugger.

When used outside the configuration file, during a debug session, this format does not change the initial PF key; therefore, you can use format 3 to restore your default PF key assignments.

Format 3: This format sets the PF key indicated by the N argument to its default command assignment. Using an asterisk (*) for the N argument gives all PF keys their initial assignments for the session. When used in a configuration file, this command must be before any **keys define** commands (format 2).

Format 4: This format sets the PF key indicated by the N argument to the **help < >** command. N can be any number from 0 to 24. The key assigned to the **help < >** command is referred to as the help key. If you do not want to have a help key, a 0 is used as the N argument.

If format 4 of the **keys** command is used to reassign the help key during a session, the default command assignment is restored to the old help key. By default, PF1 is assigned the **help < >** command.

If present in the configuration file, the **keys help** command must be before the **keys define** command.

Format 5: This format is used to specify PF keys that are to be handled by ISPF. The N argument is used to select the PF key. Use an asterisk (*) for the N argument to indicate all PF keys. The keyword **on** is used to assign a PF key to

ISPF and the keyword **`off`** causes the debugger to handle the key. This setting is shown in the Keys window. By default all keys are handled by the debugger. The help key cannot be assigned to ISPF.

EXAMPLES

`keys`
displays all PF key assignments.

`keys list 2`
displays the assignment for the PF2 key.

`keys list *`
displays all PF key assignments.

`keys define 7 ''break entry''`
assigns the **`break entry`** command to the PF7 key.

`keys default 7`
restores the PF7 key to the initial assignment for the session.

`keys default *`
restores all PF keys to their initial assignments for the session.

ADDITIONAL DISCUSSION AND EXAMPLES
See "Using PF Keys" on page 27 and "Setting Up a Configuration File" on page 46.

SYSTEM DEPENDENCIES
none

COMMAND CAN BE ISSUED FROM

PROFILE	no
configuration file	yes (**`Keys define`** command only)
Source window prefix	none

SCOPE
The **`keys`** command is not affected by changes in scope.

RETURN CODES SET

Successful: 0

Unsuccessful: 1

SEE ALSO
"config" on page 202

list

Output a Source Line Listing in Line Mode
Move to a Source Line in Full-Screen Mode

ABBREVIATION
`l{ist}`

FORMATS

Format 1:	**`list`**
Format 2:	**`list`** LINENO[:+INTEGER]
Format 3:	**`list`** +INTEGER[:+INTEGER]
Format 4:	**`list`** –INTEGER[:[–]INTEGER]
Format 5:	**`list`** –INTEGER[:+INTEGER]
Format 6:	**`list`** [FUNCTION-NAME | (SECTION-NAME)] *
Format 7:	**`list`** [FUNCTION-NAME | (SECTION-NAME)] LINENO[:LINENO]

DESCRIPTION

The **`list`** command lists source lines in a program executing under the debugger.

Format 1: In full-screen mode, this format of the **`list`** command can be used to return the debugger to the current line in the command scope. In full-screen mode, source code is displayed in the Source window and the current line is highlighted. The Status window displays the command scope as described in "Using the Status Window" on page 20.

In line mode, issuing the **`list`** command with no arguments displays the current line in the command scope.

On entry to a function, the current line is the function header line. (Part 2, "Configuring and Using the Debugger," discusses how the debugger finds the source file or files under the different operating systems.)

Format 2: In full-screen mode, the optional :+INTEGER is meaningless; the number of lines displayed in the Source window is determined by the Source window height. However, this format can be used to move to a specific line in the module that is currently displayed in the Source window; although, it is much easier to move to a line by specifying the module and line number in the **`Module:`** and **`Line:`** fields of the Source window. See "Source Window" on page 173 for information about the **`Module:`** and **`Line:`** fields.

In line mode, this format lists the source line, specified by LINENO, and as many lines after it as you specify in the INTEGER argument. LINENO is a source line number (an integer constant). INTEGER is an integer constant that indicates a number of lines.

Formats 3, 4, and 5: These formats are known as the relative formats. In full-screen mode, **`list`** +INTEGER scrolls the Source window down and **`list`** –INTEGER scrolls the Source window up the number of lines specified by the INTEGER argument. As explained in format 2, the optional :+INTEGER argument is ignored.

In line mode, each time a range of lines is displayed (a single line can also be thought of as a range of lines), the debugger remembers two lines: LS is the start of the range, and LE is the end of the range. Stopping at a new location resets both LS and LE to the line number of the new location. The relative formats then function as shown in Table 14.2 on page 232.

Table 14.2 Results of Using the list Command Relative Formats

Command	Result
`list +N1`[1]	displays LE through LE+N1
`list +N1:+N2`	displays LE +N1 through LE+N2

Command	Result
`list -N`	displays LS-N1 through LS
`list -N1:-N2`	displays LS-N1 through LS-N2
`list -N1:-N2`	displays LS-N1 through LS-N2
`list -N1:+N2`	displays LS-N1 through LE+N2

1 N1 and N2 are INTEGER arguments.

The first time you issue one of the relative formats of the **`list`** command from a location where you are stopped, both LS and LE are equal to your current line number. However, LS and LE can be incremented or decremented as a result of issuing a **`list`** command. Subsequent **`list`** commands result in a range of lines being displayed based on the current value of LS and LE.

Format 6: In line mode, this format lists all lines in the source file or, optionally, all lines in a named section or function. SECTION-NAME is a section name for your program as specified with the **`sname`** compiler option or the default. FUNCTION-NAME is the name of a function. You can specify any function name or section name in the calling sequence.

In full-screen mode, this format can be used to move to the first line in the section or function specified by the SECTION-NAME or FUNCTION-NAME argument.

Format 7: In line mode, this format lists the source line in the section or function indicated by the LINENO argument. To list several lines, specify the line number of the first line that you want to list, a colon, and the line number of the last line that you want to list. You can specify any function name or section name in the calling sequence.

In full-screen mode, format 7 can be used to move to the line number, specified by the LINENO argument, in the section or function specified by the SECTION-NAME or FUNCTION-NAME argument.

For both line mode and full-screen mode, the section or function that is specified in formats 6 and 7 must be in the calling sequence.

The previous description of the **`list`** command during a line mode session assumes that the debugger is running with **`auto list`** set. (You can check the setting with the **`query`** command; **`auto list`** is the default.) For specialized debugging, you can set **`auto nolist`**. In this situation, you should either specify the section or function name as described for formats 6 and 7, or use the **`list`** command without arguments to orient the debugger before using any of the formats that require an INTEGER argument.

EXAMPLES

Table 14.3 on page 237 provides examples of formats 3, 4, and 5. The following are line mode examples of some of the other formats:

`list`
lists the current source line, which is where execution was interrupted last.

`list func1 *`
lists all the source lines in the **`func1`** function.

`list (comp23) 200:220`
lists source lines 200 through 220 of the **`comp23`** compilation.

SYSTEM DEPENDENCIES

The location of the program source file is system-dependent. See Chapter 5, “Running the Debugger under TSO,” on page 61; Chapter 6, “Running the Debugger under OS/390 Batch,” on page 65; Chapter 7, “Running the Debugger under CMS," on page 69; Chapter 8, "Using the Debugger from a Remote System," on page 73; and Chapter 9, "Using the Debugger in a Cross-Development Environment," on page 91 for information about source file locations under different operating systems.

COMMAND CAN BE ISSUED FROM

PROFILE	no
configuration file	no
Source window prefix	none

SCOPE

The **`list`** command uses command scope to supply default function names and section names.

RETURN CODES SET

Successful: number of last line listed

Unsuccessful: 0

log

The log command can be used to log the contents of the Log window to a data set.

ABBREVIATION

`Lo{g}`

FORMATS

Format 1: **`log`** file [FILENAME]

Format 2: **`log`** append FILENAME

Format 3: **`log`** start|stop|capture

DESCRIPTION

Format 1:

The **`file`** keyword specifies the file to which logged output is to be written. The **`log`** command writes over the file. If the **`log file`** command is issued without any arguments, the name of the current log file is displayed.

The FILENAME argument is specified as a **`tso:`** style filename under OS/390, and a **`cms:`** style filename under CMS. Do not, however, specify the **`tso:`** or **`cms:`** prefix in the command; it is assumed.

Format 2:

The **`append`** keyword specifies the file to which logged output is appended.

Format 3:

Logging of the contents is started by issuing a **`log start`** command. Logging is turned off by issuing a **`log stop`** command. The **`log stop`** command does not close the file; it flushes the file to disk. Logging may be resumed at anytime by another **`log start`** command.

Issuing a subsequent **`log file filename`** or **`log append filename`** command closes the current log file and opens the file specified for logging.

The **`log capture`** command is used to log everything in the debugger's Log window buffers since the last **`log stop`**. Some log output may be lost if the Log window buffer is not large enough.

Issuing the **`log`** command with either a **`file`** or an **`append`** keyword and a FILENAME argument specifies the file to be used for logging. However, it does not start the logging process.

EXAMPLES

`log file debugger.log`
use the tso data set **`debugger.log`** specification for log output.

`log file debugger log a`
use the cms file **`debugger log a`** specification for log output

`log file`
display the current log file name specification.

`log append debugger.log`
append output logging to the tso data set **`debugger.log`**

`log start`
start the logging process

`log stop`
stop the logging process

SYSTEM DEPENDENCIES
The **`filename`** argument is specified as a **`tso:`** style filename under OS/390, and a **`cms:`** filename under CMS. However, do not specify the **`tso:`** or **`cms:`** prefix in the command. It is assumed.

COMMAND CAN BE ISSUED FROM

PROFILE	no
configuration file	no
Source command prefix	no

SCOPE
The **`log`** command is not affected by changes in scope.

RETURN CODES SET

- Successful: 0
- Unsuccessful: 1

monitor

Check for Changes Made to an Object

ABBREVIATION
`m{onitor}`

FORMATS

Format 1:	**monitor**	EXPRESSION [LENGTH | CTYPE)] [in FUNCTION-NAME] [**print**] [**where**]
Format 2:	**monitor**	EXPRESSION [LENGTH | (CTYPE)] [in (SECTION-NAME)] [**print**] [**where**]
Format 3:	**monitor**	EXPRESSION [LENGTH | (CTYPE)] [**library**] [**print**] [**where**]

DESCRIPTION

The **monitor** command causes the debugger to test for changes in the value of the monitored object at every line-number hook. If the value changes, the program is interrupted.

The request itself is called a *monitor*. When the **monitor** command is used to request the debugger to check for changes in the value of an object, the debugger is said to be *monitoring* the object. A change in the value of the monitored object causes the monitor to be *triggered*.

EXPRESSION identifies the object (such as a variable) to be monitored.

The following options can be specified in any order:

LENGTH
: is the number of bytes to be monitored. It does not need to be the same as the length of the object.

(CTYPE)
: is either a structure|union tag or a type defined with a **typedef**, or an arithmetic type. If neither LENGTH nor CTYPE is specified, the length used is the length corresponding to the type of the object.

in FUNCTION-NAME
: specifies that the debugger monitor the object only when the named function is executing. You cannot use this argument with the in (SECTION-NAME) option or with library.

in (SECTION-NAME)
: specifies that the debugger monitor the object only when functions in the named section are executing. You cannot use this argument with the in FUNCTION-NAME option or with library.

library
: specifies that the debugger also monitor the object during calls to C library functions. If you do not use this keyword, the object is not monitored during these calls. Use **library** only if you suspect that an object is being modified inadvertently by a **library** function. You cannot use this keyword with the in (FUNCTION-NAME) or in (SECTION-NAME) options.

print
: causes the debugger to issue a **print** command when the monitor is triggered. If print is used, the debugger prints the new value of the object. If the object is no more than 256 bytes in length, the debugger also prints the old value. The values are formatted appropriately for the type of the object. If the object is an aggregate and less than 256 bytes in length, only those fields that have changed are displayed.

`where`
causes the debugger to issue a **`where`** command when the monitor is triggered. The traceback is produced only once, even if more than one monitor is triggered at the same line-number hook.

Monitor requests are maintained in the same way as breakpoints. Each monitor is associated with a number that can be displayed by the **`query`** command. Monitor requests can be disabled, enabled, or dropped like breakpoints.

If the object is identified by the form **`arr[index_expression]`**, where the index expression is not constant, then the current value of the index expression is used to determine the location of the object to be monitored. This means that the monitored location does not change if the value of the index expression changes.

The debugger distinguishes monitor requests by the class and type of the object. In this context, an object can have one of three classes: auto, static, or address.

A simple identifier has class auto if it is an automatic variable or a parameter. The identifier has class static if it is a **`static`** or **`extern`** variable. Expressions used to represent portions of an array or structure (for example, **`arr[index_expression]`**, **`*(arr + index_expression)`**, where **`arr`** is an array, or **`str.mem1`**, where **`str`** is a structure) have the same class as the array or structure. Expressions that involve some form of indirection (for example, **`*(p+5), p-->a`**, where **`p`** is a pointer or **`*0p`** address) have class address.

The class affects the way the **`query`** command displays the monitor request. With one exception, **`query`** displays auto and static monitor requests symbolically (that is, the identifier or expression used in the **`monitor`** command) and class address monitor requests by hexadecimal address. The exception is monitor requests for an element of an array, when the element is distinguished by an expression. In this case, **`query`** displays the monitor request using the value of the expression at the time the **`monitor`** command is issued. This emphasizes that a fixed element of the array is being monitored.

The class also affects the scope of the monitor. If the class is auto, the monitor is enabled only while the function is executing. If the function is recursive, then a separate monitor is used for the current and any future occurrences of the function. Thus, a single monitor request can monitor multiple occurrences of the expression. However, only one request number is associated with the request. Therefore, it is not possible to drop, disable, or enable individual occurrences. Similarly, it is not possible to display or modify any occurrence of the object except the current one.

If a static object is in a subsidiary load module, and the load module is unloaded via the **`unloadm`** function, the debugger automatically drops the monitor and issues a message to that effect. Similarly, if a monitored object of class address is in storage that was allocated via the **`malloc`** function, and the storage is freed by **`free`**, the debugger drops the monitor and issues a message.

The debugger distinguishes monitor requests by the type of the object. Table 14.3 on page 237 shows how the debugger treats monitor requests for objects by type.

Table 14.3 Types of Objects for the monitor Command

Type		Notes
1	arithmetic	The object is monitored. LENGTH is not usually specified because the length can be determined by the C type. If LENGTH is used, the debugger monitors the number of bytes specified starting at &EXPRESSION.
2	pointer	This is the same as type 1, arithmetic.

Type		Notes
3	structure or union	This is the same as type 1, arithmetic.
4	array	This is the same as type 5, address.
5	address	An address cannot be monitored. Storage at an address, however, can be monitored. For example, **`monitor *0p12345678 20`** monitors 20 bytes of storage, starting at the address 0p12345678.
6	**`enum`** constant	An **`enum`** constant cannot be monitored.
7	bitfield	All bytes that contain the bitfield are monitored. However, the debugger does not interrupt the program unless the bits in the bitfield are changed.
8	function	A function cannot be monitored.

Identical requests: If a monitor request is made that is identical to an existing one, the identical request is not installed. This is true whether the request to be installed is within an **`on`** command or typed in at the command line.

If an identical request is issued and the original request is disabled, the identical request is discarded and the original request is automatically enabled without an indication.

EXAMPLES

The following examples illustrate the **`monitor`** command, given the following declarations:

```
int loopcnt;
struct ABC *p;
int arr[10] ;
char buf[100] ;
```

`monitor loopcnt print`
monitors the **`loopcnt`** variable. If the monitor is triggered, both the old and new values are printed.

`monitor *p library`
monitors the structure pointed to by **`p`**. The value of **`p`** when the monitor is installed is used to determine the address being monitored. The debugger checks for changes to the structure, even while library functions are executing.

`monitor *p 20 in (sect1)`
monitors 20 bytes, starting at the location pointed to by **`p`** when any function in **`sect1`** is executing. The value of **`p`** when the monitor is installed is used to determine the address being monitored.

`monitor *arr where`
monitors the first element of **`arr`**. If the monitor is triggered, the debugger produces a traceback.

`monitor *buf 8 in func1`
monitors the first 8 bytes of **`buf`**, only when **`func1`** is executing.

SYSTEM DEPENDENCIES

none

COMMAND CAN BE ISSUED FROM

PROFILE	no
configuration file	no
Source window prefix	none

SCOPE

The **monitor** command uses command scope to resolve references to all identifiers.

RETURN CODES SET

Unsuccessful: 1

Successful: 0

SEE ALSO

- □ "disable" on page 212
- □ "drop" on page 213
- □ "enable" on page 216
- □ "query" on page 244

on

Perform One or More Commands at Specified Locations

ABBREVIATION

o{n}

FORMAT

on HOOK-TYPE [when (EXPRESSION)] [count N] CMD | {CMD-LIST}

DESCRIPTION

The **on** command enables you to perform one or more debugger commands at various locations in your program.

In other words, you can issue a single debugger command (CMD) or a list of debugger commands ({CMD-LIST}) for the HOOK-TYPE argument. Chapter 3 explains the values you can use for the HOOK-TYPE argument, which is an argument that enables you to specify hook locations. The other arguments are explained here:

when

You can optionally use a when clause to give commands conditionally at hooks.

count N

The count N argument is optional. If count N is specified, the first N – 1 times the hook is reached, the count is decremented. The Nth time it is hit, the command is executed. After the Nth time, the command is executed every time the hook is hit.

If a when clause is present, a hit is counted only if the when expression is true.

A **query** command, issued before the count drops to 1, displays the current value of count. The word count can be abbreviated to cou{nt}.

CMD

CMD can contain any debugger command and its arguments. Abbreviations are accepted for the command also. (Note that **on** commands can be nested, as described in the next section.)

CMD-LIST

The CMD-LIST argument contains one or more of the commands and arguments separated by semicolons or new lines. CMD-LIST can contain nested **on** commands. CMD-LIST is enclosed by braces.

Commands that are part of the CMD-LIST argument can be entered on different lines. Use an open brace at the end of the first line, followed by the list of commands, and then a close brace. Note that if you put commands on separate lines, then the end of a line can be the command separator. You do not need a semicolon.

The following is a list of other points you need to know about the CMD-LIST arguments:

- When the **exec** command is used in a CMD-LIST, no other debugger command can occur on the same line following **exec**.
- If CMD-LIST contains the **break** command, then control over the restart of execution is turned over to you only after all the other commands in the list are performed.
- The debugger does not check the syntax of your CMD-LIST arguments until it is time to execute them. If you make a syntax mistake, it is not detected until the debugger tries to execute the command.
    - If a command name is syntactically incorrect, the debugger *flushes the buffer* for that command and any command after it. They are not executed. The debugger tells you about the commands that it did not execute and turns control over to you.
    - If you enter a command name that is syntactically correct, but the argument is invalid, the debugger does not flush the buffer. Instead, the debugger executes valid commands and tells you about the invalid argument via an error message.
    - If you make a mistake in the when clause of an **on** command, no commands from the CMD-LIST are executed. If any other **on** commands apply to this hook, they are not executed. The debugger turns control over to the user.
    - When used as part of a CMD-LIST, the **escape** command behaves like any other command in CMD-LIST. It is executed in order.

EXAMPLES

on entry print
prints the parameters on entry to all functions.

on main 45 print i, pxyz -->a
prints **i** and **pxyz -->a** at line 45 of the **main** function.

on 53 {print i; dump s str}
prints **i** and dumps the string pointed to by **s** at line 53 of the current function.

on (comp23) entry print
prints the parameters on entry to the functions in the **comp23** compilation.

on func1 entry {where; print parm1, parm2};
issues a **where** command and prints **parm1** and **parm2** on entry to the **func1** function.

SYSTEM DEPENDENCIES

Although the **on** command has no system dependencies, the behavior of commands used with **on** can have system dependencies. For example, the behavior of the **escape**, **system,** and **exec** commands used as CMD arguments or as part of CMD-LIST is system-dependent. See "escape" on page 218, "exec (CMS)" on page 221, "on" on page 239, and "system (CMS)" on page 267 for details.

COMMAND CAN BE ISSUED FROM

PROFILE	yes
configuration file	no
Source window prefix	none

SCOPE

The **on** command uses command scope to supply default identifiers, function names, and section names.

RETURN CODES SET

Successful: request number

Unsuccessful: 0

SEE ALSO

- □ "auto" on page 196
- □ "break" on page 199
- □ "disable" on page 212
- □ "drop" on page 213
- □ "enable" on page 216
- □ "ignore" on page 226
- □ "trace" on page 269

print

Print the Value of an Expression

ABBREVIATION

p{rint}

FORMATS

Format 1:	**print**	
Format 2:	**print**	EXPRESSION
Format 3:	**print**	EXPRESSION [%FMT] [COUNT] [,EXPRESSION [%FMT][COUNT]]

DESCRIPTION

The **print** command prints the value of various program elements. The specific elements printed depend on the format of the **print** command that you use.

Format 1: With Format 1, what is printed depends on where execution is interrupted when you issue **print**, as shown in Table 14.4 on page 242.

The left column in the table describes different places in a program where execution can be interrupted. The right column gives the output of Format 1 for each location.

Table 14.4 Output from the print Command with No Arguments

Where Execution Stopped	**What Is Printed**
calls from a function (calling function context)	the parameters as viewed by the called function
entry to a called function (called function context)	the parameters as viewed by the called function
normal return to a function (calling or called function context)	the return value
longjmp from a function (The context is that of the function you are jumping from.)	the value of the argument to **longjmp**
line-number hook	all automatic scalars of the function in scope

Format 2: This format of the **print** command displays the value of the expression identifier. The value is displayed according to its type, as declared in the source code. If the EXPRESSION contains a modulus (**%**) operator, see Format 3.

When you issue Format 2 of the **print** command, the value of EXPRESSION is printed according to its type shown in Table 14.5 on page 242.

Table 14.5 Output Format for Values of Expressions Using the print Command

Type		**Print Output Format**
1	arithmetic	**char** type: the hexadecimal value is always printed. If printable, the value is printed as is. If not printable and if the value is an escape character, the escape sequence is shown. If the value is not an escape character, only the hexadecimal value is shown.
		Types other than **char**: the value is printed in decimal and hexadecimal format. **float** and **double** types are printed in the most appropriate form, as determined by the rules for the **g** format used by the **printf**, **sprintf**, and **fprintf** functions.
		enum variable: The value is printed in decimal. The enumeration constant symbol is printed in parentheses following the value.
2	pointer	The value is printed as eight hexadecimal digits with leading **0p**.

Type		**Print Output Format**
3	structurer, union, or class	All members are printed according to their declared type: types 1, 2, or 7. (Arrays, structures, classes, and unions contained in the structure are fully expanded to show the values of their members.) The address of a structure or union is printed also.
4	array	The address of the array is printed as eight hexadecimal digits with leading **`0p`**.
5	The address is printed as eight hexadecimal digits with leading **`0p`**.	An address cannot be monitored. Storage at an address, however, can be monitored. For example, **`monitor *0p12345678 20`** monitors 20 bytes of storage, starting at the address 0p12345678.
6	**`enum`** constant	The value of the constant is printed in decimal digits.
7	bitfield	The value of a bitfield is printed in both binary and decimal formats. If the number of bits in the bitfield is a multiple of eight, then the value of the bitfield is also printed in hexadecimal format.
8	function	A function cannot be printed. However, indirection through a function pointer is supported and prints the function name, section name, and, if located in a different load module, the load module name.

Format 3: Format 3 prints the value of the expression (or expressions) identified by EXPRESSION. COUNT specifies the number of values to be printed. If the EXPRESSION argument contains a modulus (%) operator, you must escape the operator with a backslash (\); otherwise, the debugger interprets the modulus operator as a format specifier.

The %FMT argument can be any of the format specifiers that you use with the **`sprintf`** function. These are

`c`	single character
`d`	decimal signed integer
`e`	or **`E`** exponential floating point
`f`	fixed decimal floating point
`g`	or **`G`** f format or e format
`o`	octal integer
`s`	character string
`u`	decimal unsigned integer
`x`	hexadecimal integer (lowercase)
`X`	hexadecimal integer (uppercase)

The **`print`** command supports any `%` format specification that results in the item being formatted in 256 or fewer bytes. If the %FMT argument is not specified, the debugger uses a format determined by the type of the expression (as in Format 2).

COUNT is an integer that specifies the number of items to be printed. If not specified, COUNT defaults to 1.

If more than one EXPRESSION is specified, any (or all) of the %FM and COUNT arguments can be used for each value to be printed. Each EXPRESSION argument and its list of associated %FMT and COUNT arguments (if any) are separated by a comma from the next EXPRESSION argument and its list of %FMT and COUNT arguments.

EXAMPLES

The debugger **print** command examples are based on the following declarations:

```
int i;  double d;
struct XYZ {int a; double b; } xyz;
struct INT2 {int a,b;} int2;
struct XYZ * pxyz;
int arr[10] ;
```

print i
displays the value of **i**.

print i, d
displays the values of **i** and **d**.

print i + d
displays the value of **i** plus **d**.

print xyz.a
displays the value of **xyz.a**.

print pxyz --> b
displays **pxyz--> b**.

print arr[i]
displays the ith element of the **arr** array.

print *(arr+i) + int2.a
displays the value of the sum of the ith element of **arr** plus the value of member **a** in struct **int2**.

print *arr 5
displays first 5 elements in **arr**.

print ::xyz
allows the file scope variable to be accessed in any command that supports expressions by using the C++ unary operator ::.

SYSTEM DEPENDENCIES

none

SCOPE

The **print** command uses command scope to resolve references to all identifiers.

RETURN CODES SET

Successful: 0

Unsuccessful: 1

SEE ALSO

- □ “dump” on page 215
- □ “monitor” on page 235
- □ “watch” on page 276

query

Display Breakpoint/Action Requests

ABBREVIATION

`q{uery}`

FORMATS

Format 1:	**`query`**	
Format 2:	**`query`**	FUNCTION-NAME | (SECTION-NAME)
Format 3:	**`query`**	FUNCTION-NAME | (SECTION-NAME) LINENO
Format 4:	**`query`**	FUNCTION-NAME | (SECTION-NAME) LINENO:LINENO
Format 5:	**`query`**	FUNCTION-NAME | (SECTION-NAME) calls/ entry/return

DESCRIPTION

The **`query`** command shows you a query list (a numbered list of the **`break`**, **`trace`**, **`ignore`**, **`monitor`**, and **`on`** requests currently in effect), as well as a summary of the **`auto`** options.

Format 1: The **`query`** command issued with no arguments produces the numbered list of all requests. The list also shows the current setting of the **`auto`** command options in effect: echo|noecho, id|noid, list|nolist, nullptr|nonullptr, wrap|nowrap, cmacros|nocmacros, and linesize.

Format 2: The **`query`** command used with a section name (SECTION-NAME) or a function name (FUNCTION-NAME) produces only a list of actions in effect for the function or section without a summary of **`auto`** options.

Format 3: Format 3 enables you to issue a **`query`** command for requests in effect at a particular source line (LINENO) of a particular function or section. You must use either a function name or section name with the LINENO argument.

Format 4: Format 4 produces a query list for a particular section (SECTION-NAME) or function (FUNCTION-NAME) of all requests within a line number range (LINENO:LINENO). LINENO is a source line number. The list can include requests that begin or end outside the specified range. For example, if you specify a range of 25:28 for a particular function, **`query`** displays a list of all requests that involve that range, such as a request beginning on line 20 but ending on line 26.

Format 5: Format 5 produces a query list of requests in effect for a particular function or section as follows:

- □ at calls by a function and returns to the function
- □ upon entry into a function
- □ at return from a function.

EXAMPLES

`query`
displays the current settings of the **`auto`** command plus a list of all the actions and monitors in effect for the entire program.

`query func1`
displays the actions and monitors in effect for the **`func1`** function.

`query (comp23)`
displays the actions and monitors in effect for the **`comp23`** compilation.

query func1 e
displays the actions and monitors in effect upon entry into the **func1** function.

query func1 10:50
displays the actions and monitors in effect for lines 10 through 50 of the **func1** function.

SYSTEM DEPENDENCIES
none

COMMAND CAN BE ISSUED FROM

PROFILE	yes
configuration file	no
Source window prefix	**q**

SCOPE
The **query** command is not affected by changes in scope.

RETURN CODES SET:

Successful: 2, number of last request satisfying the 2 arguments of the **query** command

Unsuccessful: 0

SEE ALSO

- □ "auto" on page 196
- □ "disable" on page 212
- □ "drop" on page 213
- □ "enable" on page 216

resume

Resume Program Execution

ABBREVIATION
res{ume}

FORMATS

Format 1:	**resume**	
Format 2:	**resume**	LINENO
Format 3:	**resume**	FUNCTION-NAME
Format 4:	**resume**	FUNCTION-NAME LINENO

DESCRIPTION
The **resume** command enables you to resume execution of a program running under the debugger at any line-number hook in any active function in the calling sequence. The **resume** command, or its alias **goto**, can be issued when stopped at

any line or return hook. While you cannot issue a **`resume`** command when stopped at an entry hook, you may be able to step to the first line hook at the opening brace and then issue **`resume`**.

You cannot use **`resume`** after using the **`attn`** command, but you may be able to issue **`resume`** after a subsequent **`break`**, **`step`**, or **`continue`** command returns control to you.

The **`resume`** command can also be used to attempt recovery from the following types of error conditions:

- an 0C4 or 0C5 (identified by the signal SIGSEGV)
- computational floating-point errors identified by the SIGFPE class of signals: SIGIDIV, SIGFPDIV, SIGFPOFL, and SIGFPUFL
- a call to the function **`abort`**, identified by the SIGABRT signal
- catch of a thrown exception.

When you receive one of the signals listed above with a program running under the SAS/C Debugger, you can examine the values of variables, make changes to variables, and so on. Then, issuing **`resume`** causes the debugger to discard the signal and resume execution.

The location in your program where execution resumes depends on the format of the **`resume`** command that you used and where execution of the program stopped. Note that while you cannot resume a library function or a function compiled with the **`nodebug`** option, you can resume a function that called one of these two types of functions. However, the calling function must be compiled with **`debug`**.

Because the effect of the **`resume`** command is similar to inserting a one-time **`goto`** or **`longjump`** in the program, when bypassing code, you must ensure that variables contain proper values for resumption of execution at the target hook.

Format 1: This format restarts execution at the last line-number hook encountered before the location stopped.

Format 2: This format resumes execution at the first line-number hook encountered in the source line identified by LINENO in the current function.

Format 3: This format resumes execution at the last line-number hook executed in an active function identified by FUNCTION-NAME.

Format 4: This format resumes execution at the first line-number hook in the specified line (LINENO) of the function named by FUNCTION-NAME. The function must be active.

EXAMPLES

`resume`
resumes execution at the last line-number hook executed.

`resume 23`
resumes execution at the first line-number hook encountered in line 23 in the function where it is currently stopped.

`resume prevfunc`
resumes execution at the last line-number hook executed in the function **`prevfunc`**.

`resume prevfunc 34`
resumes execution at the first line-number hook in line 34 of the function **`prevfunc`**.

SYSTEM DEPENDENCIES

none

COMMAND CAN BE ISSUED FROM

PROFILE	no
configuration file	no
Source window prefix	**g**

SCOPE
The **resume** command is affected by changes in scope.

RETURN CODES SET
not applicable

SEE ALSO

- □ "goto" on page 223
- □ "return" on page 248

return

Return Immediately from a Function

ABBREVIATION
ret{urn}

FORMATS

Format 1:	**return**	
Format 2:	**return**	[SCALAR-TYPE-EXPRESSION]
Format 3:	**return**	AGGREGATE-TYPE-EXPRESSION | {VALUE-LIST}

DESCRIPTION
The **return** command performs an immediate return from a function without executing any further code in the function. Return values are supplied by the arguments.

The **return** command can be issued from any hook. However, additional debugger requests that are in effect at the normal return from the function, are processed only if the **return** command is issued from a line hook.

Format 1: This format is used for functions that return **void** or **int**. In the case of functions that return integers, the value in register 15 is returned.

Format 2: This format is used for functions that return a nonbitfield scalar. Except for functions that return an integer, you must always supply the SCALAR-TYPE-EXPRESSION argument if the function returns a scalar. See "SCALAR-TYPE-EXPRESSION Argument" on page 143 for additional information.

Format 3: This format is used for functions that return an AGGREGATE-TYPE-EXPRESSION. An aggregate expression has a type of structure or union. See “AGGREGATE-TYPE-EXPRESSION Argument” on page 143 for additional information.

The VALUE-LIST argument is used to supply a list of values that is returned to the calling function’s structure or union. The VALUE-LIST argument contains any or all of the following items, enclosed by braces:

- □ one or more VALUE arguments separated by commas:

  ```
  {VALUE, VALUE, VALUE, . . .}
  ```

- □ one or more NULL-INITIALIZERS. A NULL-INITIALIZER is an empty pair of braces: {}.
- □ one or more VALUE-LISTs. This means that a VALUE-LIST can contain nested VALUE-LISTs.

The values supplied with a **`return`** command are assigned to the receiving structure or union in the same manner as they are with the **`assign`** command. See Rules for assigning VALUE-LISTs to aggregate objects in “assign” on page 192.

EXAMPLES

`return`
returns from a calling function that returns either a **`void`** or an **`integer`**. If the return type is **`integer`**, the value in register 15 is returned.

`return ptr`
returns the pointer **`ptr`** to the calling function.

`return my_struc`
returns the structure **`my_struc`** to the calling function.

`return {1,2,'A'}`
returns a VALUE-LIST argument of **`{1,2,'A'}`** to the calling function.

SYSTEM DEPENDENCIES
none

SCOPE
The **`return`** command is not affected by changes in scope.

COMMAND CAN BE ISSUED FROM

PROFILE	no
configuration file	no
Source window prefix	none

RETURN CODES SET

Successful: 0

Unsuccessful: 1

SEE ALSO

- □ “goto” on page 223
- □ “resume” on page 246

rsystem

Execute an Operating System Command on a Remote System

Note: The SAS/C Debugger's **`rsystem`** command is intended for use with the remote debugger. It is similar to the **`system`** command described in this chapter, but it allows you to execute an operating system command in the environment of the program being debugged. △

ABBREVIATION

`rs{ystem}`

FORMAT

`rsystem` OPERATING-SYSTEM-COMMAND

DESCRIPTION

The **`rsystem`** command is intended for use with the remote debugger. It sends the command specified in the argument OPERATING-SYSTEM-COMMAND to the environment of the program being debugged. For example, you might use this command to execute a CMS command from TSO when debugging a CMS program from TSO.

The **`rsystem`** command cannot be followed by another debugger command on the same line. That is, the arguments to the **`rsystem`** command are assumed to extend to the end of the line, including any semicolons on the line.

EXAMPLES

```
rsystem alloc fi(iforgot)
 da(to.allocate.this.dataset) shr
```

executes the TSO ALLOCATE command.

```
rsystem access 391 Q
```

executes the CMS ACCESS command.

SYSTEM DEPENDENCIES

See "system (CMS)" on page 267 for more information about the **`system`** command. Command output, if any, generally appears in the remote program's session or log.

The **`rsystem`** command cannot be used when the remote program is executing under UNIX System Services or CICS.

COMMAND CAN BE ISSUED FROM

PROFILE	yes
configuration file	no
Source window prefix	none

SCOPE

The **`rsystem`** command is not affected by changes in scope.

RETURN CODES SET

Successful:	return code from operating system
Unsuccessful:	-101 or less;
	-103 means command not found;

-104 means syntax error;

-110 means the command is not supported in the target environment

SEE ALSO
"system (CMS)" on page 267

runto

Resume Execution and Request a Temporary Breakpoint

ABBREVIATION
`ru{nto}`

FORMAT
`runto` HOOK-TYPE [when(EXPRESSION)] [count N]

DESCRIPTION
The **`runto`** command places a temporary, or one-shot, breakpoint at the location specified by the HOOK-TYPE argument. (See "HOOK-TYPE Argument" on page 135.)

A when clause is used to request **`runto`** breakpoints conditionally; that is, a breakpoint is requested at the specified hook only if the when clause is true when the hook is reached.

The argument count N is optional. If count N is specified, the first N – 1 times the hook is reached, the count is decremented. The Nth time it is hit, the debugger breaks.

If a when clause is present, a hit is counted only if the when expression is true.

A breakpoint set with a **`runto`** command remains installed only until execution is stopped and the debugger returns control to you. In other words, it is temporary and only good for one attempt. The breakpoint is removed the first time the debugger stops, whether it stops for the breakpoint set with the **`runto`** command or for some other reason.

The result of issuing a **`runto`** command is that in most cases the program executes until the temporary breakpoint is hit; however, if some other event occurs that gives you control before the breakpoint is hit, the debugger gives you control and the debugger removes the breakpoint.

Examples of other events that may stop program execution before the **`runto`** breakpoint is hit are

- another breakpoint
- **`step`** or **`continue`** command completion
- monitor requests
- communications signals
- ATTN key signal.

If you receive control because of a **`runto`** breakpoint being hit, the Status window displays Runto as the reason for entry into the debugger.

EXAMPLES

`runto main 52`
sets a temporary breakpoint at line 52 of the **`main`** function and resumes execution.

`runto func1 return`
sets a temporary breakpoint on the return from function **`func1`** and resumes execution.

`runto stats 15 when(i==10)`
sets a temporary breakpoint at line 15 of the **`stats`** function when the value of **`i`** is 10 and resumes execution.

`runto 75`
sets a temporary breakpoint at line 75 of the current function and resumes execution.

`runto 75 count 5`
sets a temporary breakpoint at line 75 of the current function the fifth time the line-number hook at that line is reached and resumes execution.

ADDITIONAL DISCUSSION AND EXAMPLES
See Chapter 11, “Setting Temporary Breakpoints” on page 101

SYSTEM DEPENDENCIES
none

COMMAND CAN BE ISSUED FROM

PROFILE	no
configuration file	no
Source window prefix	**`r`**

SCOPE
The **`runto`** command uses command scope to supply default identifiers, function names, and section names.

RETURN CODES SET

Successful: 0

Unsuccessful: 1

SEE ALSO

- “break” on page 199
- “go” on page 222
- “goto” on page 223

scope

Change Command Scope

ABBREVIATION
`sc{ope}`

FORMATS

Format 1:	**scope**	FUNCTION-NAME
Format 2:	**scope**	+INTEGER
Format 3:	**scope**	–INTEGER
Format 4:	**scope**	

DESCRIPTION

The **scope** command is used to change command scope. Normally, command scope is identical to run scope, which is the term used to described the location where the debugger stops when you are given control. In full-screen mode, the code displayed in the Source window is that around the line where execution stopped. The line where execution stopped is highlighted and its location is displayed as **run scope** in the Status window.

Commands that manipulate expressions refer to the identifiers (variables, structures, and so on) that are visible as you need to look at source code or examine variables in a facility. This process allows you to specify a command scope that is different from the run scope.

Format 1: This format changes the command scope to the function named by the FUNCTION-NAME argument. This argument must name a function in the calling sequence, If there are multiple instances of the function identified by FUNCTION-NAME, command scope is set to the most recent.

Format 2: This format changes command scope to that of a function that is farther up in the calling sequence. The INTEGER argument specifies the number of functions up in the calling sequence to change the command scope.

Format 3: This format changes command scope to that of a function that is farther down in the calling sequence. The INTEGER argument specifies the number of functions down in the calling sequence to change the command scope.

Format 4: This format sets command scope back to run scope.

The following commands use command scope to resolve references to all identifiers:

- **assign**
- **copy**
- **dump**
- **monitor**
- **print**
- **transfer** (in an expression context)
- **watch**
- **whatis.**

The following commands use command scope to supply default function or section names:

- **break**
- **goto** or **resume**
- **ignore**
- **on**
- **runto**
- **trace.**

The **transfer** command in a nonexpression context and all other commands use run scope if a scope is needed for resolution.

In either line mode or full-screen mode, you can use the **where** command to see where you are in the calling sequence. If command scope is different from run scope, it is indicated by an asterisk next to the line number in the calling trace list displayed by the **where** command. Run scope is always the first location in the calling trace list.

EXAMPLES

scope stats
changes command scope to the **stats** function.

scope – 1
changes command scope to the function that is one position up in the calling sequence: the caller. For example, if function **a** called function **b**, and your command scope is in function **b**, then command scope is changed to function **a** by this command.

scope + 1
changes command scope to the function that is one position down in the calling sequence: the callee. For example, if function **a** has already called function **b**, and you have changed your command scope to function **a**, then command scope is changed back to function **b** by this command.

scope
When using the INTEGER argument with the **scope** command, command scope is used to determine to which function in the calling sequence to move.

ADDITIONAL DISCUSSION AND EXAMPLES
See "Using the Status Window" on page 20.

SYSTEM DEPENDENCIES
none

COMMAND CAN BE ISSUED FROM

PROFILE	no
configuration file	no
Source window prefix	none

SCOPE
The **scope** command is used to change command scope.

RETURN CODES SET

Successful: 0

Unsuccessful: 1

SEE ALSO
"where" on page 279

set

Control file access

ABBREVIATION

`se{t}`

FORMAT

set SUBCOMMAND SUBCOMMAND-ARGUMENTS

DESCRIPTION

The SAS/C Debugger's **`set`** command is best used in the debugger PROFILE to specify search lists for source, include, and debugger files, as well as a cache location for your debugger file. However, the **`set`** command may also be issued on the command line. The following reference section describes both the **`set search`** subcommand and the **`set cache`** subcommand

The **`set`** command has two subcommands: **`search`** and **`cache`**. The **`set search`** subcommand is used to control the search templates that are used to access debugger and source files, and the **`set cache`** subcommand is used to specify a cache location for debugger files. The **`set cache`** subcommand also uses a template to specify this location.

The **`set search`** and **`set cache`** subcommands are described in the following paragraphs.

search SUBCOMMAND

The **`search`** subcommand is used to establish a search list, control tracing, and add or remove templates from a search list. The **`search`** subcommand has the following forms:

Table 14.6 search Subcommand Formats

Format	Example
1	**`set search`** FILE-TAG **`=|+|-`** "template1" ["template2" ...]
2	**`set search`** FILE-TAG =
3	**`set search`** FILE-TAG **`|* ?`**
4	**`set search`** FILE-TAG **`|* trace on|trace off`**

The FILE-TAG argument specifies the type of file that a template applies to and can be any of the following:

Table 14.7 FILE-TAG Values

Type of file	Description
`debug`	specifies that the template is for debugger files.
`source`	specifies that the template is for source files.
`altsource`	specifies that the template is for alternate source files. (An alternate source file refers to source code altered by a **`#line`** preprocessor statement that specifies a filename.)

Type of file	Description
`systeminclude`	specifies that the template is for system include files.
`userinclude`	specifies that the template is for user include files.

Format 1: This format of the **`set`** command specifies a search list for the type of files designated by FILE-TAG. Each search list consists of one or more templates that are used by the debugger to locate debugger or source file types.

The =|+|- argument is used as follows:

Table 14.8 set Command Operations

Argument	Description
=	sets the search list equal to the specified templates.
+	appends the specified templates to the search list.
-	removes all occurrences of the specified templates from the search list.

The template arguments define the search list. Each template argument uses one or more of the following conversion specifiers to define a template used by the debugger to generate filenames:

Table 14.9 template Arguments

Value	Description
`%lower` or **`%l`**	causes the replacement text for the conversion specifier following the **`%lower`** to be converted to lowercase. The character after the **`%lower`** or **`%l`** must be the start of another conversion specifier.
`%upper` or **`%u`**	causes the replacement text for the conversion specifier following the **`%upper`** to be converted to uppercase. The character after the **`%upper`** or **`%u`** must be the start of another conversion specifier.
`%sname` or **`%s`**	is replaced by the section name of the program being debugged. (The section name must have been specified when the program was compiled.) The section name is always uppercase. If a lowercase version is required, prefix the **`%sname`** or **`%s`** specification with **`%lower`**.
`%member` or **`%m`**	specifies the member name of a partitioned data set.
`%fullname`	is replaced by the entire filename stored in the object or debugger files. The format of the filename is implementation dependent, and this conversion specifier should not be used unless you have complete knowledge of the filename stored in the object or debugger files. This conversion specifier is most useful for alternate source files, where it will be replaced by the complete filename that appears in the **`#line`** statement.
`%leafname` or **`%lf`**	is replaced by the portion of the filename stored in the object or debugger files after the last slash, if present. If there is no slash, it is the entire filename stored in the object or debugger files.

Value	Description
%basename or **%b**	is replaced by the portion of **%leafname** that is before the last dot. If there is not a dot in **%leafname**, then **%basename** is the same as **%leafname**.
%extension or **%e**	is replaced by the portion of **%leafname** that is after the last dot. If there is not a dot in **%leafname**, then **%extension** is set to a null string.
%m	is replaced by the member name of the original source file if it was a member of PDS.

You can include a percent character (%) in a template by specifying two percent characters successively (%%).

The filenames generated by the application of the conversion specifiers in the template are passed to the **fopen** function, which opens the appropriate file for the debugger to access. If these files are located on a remote host, the SAS/C Connectivity Support Library is used to establish an NFS connection between the local and remote host.

For example, to use the SAS/C Connectivity Support Library to access files on a UNIX workstation, the following template could be specified:

```
"path:dbgfiledir/%leafname"
```

If **%leafname** consists of a base and an extension, a functionally equivalent template could be specified as follows:

```
"path:dbgfiledir/%basename.%extension"
```

A similar template could be specified to access files on OS/390. For example, the following template would access a PDS member that matches **%basename**:

```
"dsn:userid.proj4.h(%basename)"
```

Format 2: The second form of the **set search** subcommand is used to remove all of the search templates associated with a FILE-TAG. It specifies a null search list.

Format 3: The question mark (**?**) character is used to display the search list associated with a FILE-TAG. An asterisk (*) can be used as a wildcard character in place of a specific FILE-TAG argument. Specifying **set search * ?** will display the search lists for all debugger and source files, including the cache location, if it was specified with a **set cache** subcommand.

Format 4: The final form of the **set search** subcommand is used to turn tracing on or off. When tracing is turned on, the debugger displays a message each time it attempts to open a file, possibly using a filename generated by a template. The message displays the name of the file the debugger was looking for and whether or not the search was successful.

An asterisk (*) can be used as a wildcard character in place of a specific FILE-TAG argument. If an asterisk is specified for the FILE-TAG, tracing will be affected (either turned on or turned off) for debug, source, altsource, systeminclude, and userinclude files.

Search Lists

This release of the SAS/C Debugger allows you to create *search lists* for specifying the identity and location of files used by the debugger. Search lists are created with the debugger's **`set search`** subcommand and **`set cache`** subcommand.

You can use search lists in any environment where you can run the SAS/C Debugger. However, they are particularly useful in the following situations:

- □ when you develop applications in a cross-development environment, where compilations occur on a UNIX workstation using the SAS/C Cross-Platform Compiler
- □ when you compile in batch or TSO and debug in the UNIX System Services shell.

For details on using search lists in a cross-development environment, see SAS/C Cross-Platform Compiler and C++ Development System: Usage and Reference, First Edition.

Input file selection and specification

The SAS/C Debugger provides access to information from several different types of files, including

- □ debugger files
- □ source files
- □ alternate source files
- □ system include files
- □ user include files.

When the default search procedure for a file does not meet your needs, it is possible to change this behavior by using the debugger's **`set search`** subcommand.

The **`set search`**sub command is used to specify *filename templates*. Filename templates are used to specify the identity and location of the source, include, or debugger files associated with the load module being debugged. Multiple filename templates can be defined for each type of file. So, when necessary, the debugger can search for a file by more than one name or in multiple locations. Each template is saved in a search list, and each search list is associated with a specific type of file.

Filename templates are character strings, which are patterned after the format argument of the **`printf`** function. Each filename template can contain *conversion specifiers* and characters. A conversion specifier is a character or a string preceded by the percent (%) character. The conversion specifier is either replaced by its associated string or it specifies the format of the conversion specifier that follows it. The resulting string is used as the name of the file to be opened. If this fails, the next filename template is processed until either a file is opened or no more filename templates are in the search list for that type of file.

This is a very powerful technique that allows you to direct the debugger to files that have moved or even changed names or file systems.

Note: If you run the debugger under the UNIX System Services shell, the filename string is interpreted as an OS/390 filename, not as a POSIX filename. If you want to specify searching for a file in the UNIX System Services hierarchical file system, you must begin the filename with the **`hfs:`** filename style prefix. △

Reattempting a set search

If a **`set search`** subcommand is issued, followed by a **`list`** command, the debugger attempts to load any files that were not previously found, using the modified **`set search`** templates. For example, if an attempt to load a source file fails because the source files have been moved to the data set SASC.APPL.SOURCE, issuing the command

```
set search source+"dsn:sasc.appl.source(%basename)"
```

followed by a **list** command causes the debugger to reattempt the search for the source.

The **set search** issued does not have to correlate directly to the failed search. For example, a common problem encountered when debugging, is to forget to allocate the DBGLIB data set definition. When the debugger fails to locate the debugger file, a command such as

```
system alloc fi(dglib) dsn(appl.dbglib)shr
```

could be issued to allocate the Data Definition (DD) statement. A "dummy" **set search** subcommand could then be issued. For example, the following command is followed by a **list** command that will cause the search to be reattempted:

```
set search altsource+""
```

cache SUBCOMMAND

The **set cache** subcommand is used to specify a cache location for the debugger file. (In a cross-development environment, the original debugger file may be located on the host workstation, and the cache location will be on the target mainframe.) A cache location is specified to provide faster access to debugging information.

The format for the **set cache** subcommand is as follows:

Format: **set cache debug =** "**template**"

Notice that **debug** is the only valid type of file for the **set cache** subcommand.

The template argument was described in the previous section and is used to specify the cache location. When debugging a program, the debugger first looks for the debugger file in the cache location. If the debugger finds a current version of the debugger file in the cache location, then the debugger uses the file. If a debugger file is not found in the cache location, or if the debugger file in the cache location is not current, then the current debugger file is copied to the cache location. However, if the cache file is not a valid debugger file, it will not be overwritten by the debugger.

EXAMPLES

set search userinclude = "path:/usr/c/headers/%leafname"
specifies a search list for user include files. When the debugger looks for source code that was included from a user include file located on a host workstation, this template is used to generate a filename and open the file on the workstation.

set search source = "hfs:/home/cxx/src/%leafname"
specifies a search list for source files in the OS/390 UNIX System Services hierarchical file system (HFS). The **hfs:** filename style prefix instructs the debugger to look for the file in the HFS file system and open the file if it is found.

set search userinclude + "dsn:*userid*.c.headers(%basename)"
specifies a template that is appended to the search list for user include files that was established in the previous example. This template generates an OS/390 **dsn:** style filename that is searched if the user include file is not found on the workstation.

`set search userinclude trace on`
turns tracing on for user include files. Whenever the debugger searches for a user include file, a message will be displayed telling you the name of the file searched for and if the search was successful or not.

`set search userinclude ?`
displays the search template list used to generate filenames for user include file searches.

`set search userinclude =`
resets the search template list for user include files to null.

`set cache debug = "dsn:`*`userid`*`.cache.db(%sname)"`
specifies an OS/390 data set used to cache the debugger file on the target mainframe.

`set cache debug = "cms:%sname dbg370"`
specifies the location of a CMS file used to cache the debugger file on a target mainframe.

SYSTEM DEPENDENCIES
The filenames that are generated by the search templates are dependent upon the names the compiler used to open the files originally, which are operating system dependent.

COMMAND CAN BE ISSUED FROM

debugger start-up file	**yes**
command line	yes
configuration file	no
Source window prefix	none

SCOPE
The **`set`** command is not affected by changes in scope.

RETURN CODES SET

Successful: 0

Unsuccessful: 1

step

Restart Execution and Break at the Next Hook

ABBREVIATION
`s{tep}`

FORMAT
`step` [INTEGER]

DESCRIPTION
The **`step`** command resumes program execution and breaks at the next hook, which may be any of the following:

line-number hook
Some source lines, such as **`for`** statements, can have multiple hooks.

call to a function
The break occurs in the context of the calling function.

return to a function
The break occurs in the context of the calling function.

entry to a called function
The break occurs in the context of the called function.

return from a called function
The break occurs in the context of the called function.

INTEGER is a nonnegative integer (including 0). Use the INTEGER argument to specify the number of times you want the **`step`** command to be performed.

If you have issued **`step`** with an INTEGER argument and the debugger breaks before the **`step`** command is completed, you can issue it again with a different value of INTEGER to change the number of times **`step`** is performed. Suppose that the last **`step`** you issue is **`step 4`**, and then you reach a breakpoint after one step is performed. If you decide you only want the step to be performed once more, you can issue **`step 1`** at the breakpoint. (The three pending steps are replaced by one step. The **`step`** command issued with 0 discards the three remaining steps.) However, if you want the three pending steps to be completed, you can issue a **`go`** command.

EXAMPLES

`step`
resumes program execution until the next hook is reached.

`step 20`
performs the **`step`** command 20 times.

SYSTEM DEPENDENCIES
none

SCOPE
The **`step`** command is not affected by changes in scope.

RETURN CODES SET
not applicable

SEE ALSO

- "continue" on page 204
- "go" on page 222
- "resume" on page 246

storage

Display Storage Analysis

ABBREVIATION
`sto{rage}`

FORMAT
`storage` [heap | stack] [check | report] [file | term] [narrow | wide]

DESCRIPTION

The **storage** command prints an analysis of the program's use of both heap and stack storage. *Heap storage* is storage allocated by the **malloc** and **calloc** functions. Once allocated, this type of storage remains allocated until it is freed. *Stack storage* is storage used for automatic variables and register save areas. This storage is allocated when a function is entered and is freed when the function returns.

The **storage** command is executed by the same subsystem as that used when the run-time **=storage** option is used.

The keywords can be specified in any order. If conflicting options are entered, the last (rightmost) option is used.

heap
: specifies that the analysis is to be limited to heap storage only.

free
: specifies that free heap blocks are to be checked. All chains and blocks in free heap storage are inspected for correctness and consistency. If an error is detected, the block will be dumped.

stack
: specifies that the analysis is to be limited to stack storage only. If neither heap nor stack is used, the analysis includes both.

check
: specifies that the storage is to be checked; that is, the control blocks that describe the storage allocated are to be inspected for correctness and consistency. This type of analysis is similar to that performed by the library when heap storage is freed or when the program terminates. If check is used, **storage** attempts to predict storage-related user abends and to indicate the problem area.

report
: specifies that the debugger is to produce a storage use report. If neither check nor report is used, the analysis includes both. Note that a report is not generated until the storage is checked for consistency.

file
: specifies that the analysis is to be written to a file. Under OS/390 batch, the analysis is written to the data set allocated to the DDname DBGSTG. Under TSO, the debugger attempts to use DBGSTG. If the DD statement for the name DBGSTG is not defined, the output is written to 'DSN: *userid.pgmname*DBGSTG'. If NOPREFIX is specified in your TSO profile, 'DSN: *prefix.pgmname.*DBGSTG' or a TSO prefix is specified. In either case, *pgmname* is the name of the program. If the program name cannot be determined, the debugger uses UNKNOWN as the pgmname. Under CMS it is written to the A-disk, using the program name as the filename and a filetype of DGBSTG.

  The output of subsequent **storage** commands is appended to the output of the first. Each part of the analysis is identified by a header similar to the first line of a traceback.

term
: specifies that the analysis is to be displayed on the terminal. Under CMS and TSO, term is the default. Under OS/390 batch, it is written to DBGLOG. User-installed debugger I/O exits have no effect on the output of the **storage** command.

narrow
: specifies that the output is to be limited to 80 columns.

wide

specifies that the output is to be displayed in 132-column format. If neither narrow nor wide is used, the debugger chooses a format based on the choice of term or file. The term output defaults to narrow, and file output defaults to wide.

Interpreting the Reports: The following paragraphs explain how heap and stack allocation is performed by the run-time library. This information is necessary to understand the reports generated by the **`storage`** command. The following explanation is not exhaustive, but it gives enough information for you to understand and use the output from the **`storage`** command.

`Heap storage`: Heap storage is allocated by the **`malloc`** and **`calloc`** functions. Both functions interface to a routine called the heap manager to allocate storage. The heap manager creates control information for the allocated storage block and formats the storage block so that overlays (that is, information written outside the boundaries of the storage block) can be detected. When an overlay is detected (either when the storage is freed or when the program terminates), the heap manager issues a user ABEND. This alerts the programmer to storage overlays.

The heap manager can adjust the size of the allocated storage block in order to reduce fragmentation. Suppose the program allocates a 10-byte block using the following function call:

```
p = malloc(10);
```

The heap manager rounds the request size (10 bytes) to the next multiple of 8, which is 16 bytes. Then it adds 16 bytes of control information to the request, in the form of an 8-byte block header and an 8-byte block trailer.

If the request size is larger than 288 bytes, the heap manager adds the size of the header and trailer information to the request size and then rounds the sum to the next higher multiple of 64. For example, the following request results in a 320-byte (5 * 64) storage block:

```
p = malloc(300);
```

`Validation`: The most common storage overlay is caused by writing into the storage either immediately preceding or immediately following the boundaries of the storage block. For example, a program may allocate a 100-byte I/O buffer and then read 110 bytes of data into it. The heap storage manager formats the storage block so that this type of error is detected easily. Part of the control information is the word HEAP in the 4 bytes immediately preceding and following the original 16-byte block. The block looks like the following:

```
xxxxHEAPzzzzzzzzzzzzzzzzHEAPxxxx
```

In the original 16-byte block, **`x`** is a byte of control information and **`z`** is a byte.

If the program has written over either of the HEAP markers when the storage block is freed, the heap manager detects that the storage was overlaid.

Similarly, when a storage block is freed, the heap manager changes the HEAP markers to FREE, as shown in the following:

```
xxxxFREEzzzzzzzzzzzzzzzzFREExxxx
```

These markers remain in place until the block is reallocated or the program terminates; at which time, if either marker is changed, the heap storage manager detects the overlay.

The heap manager also ensures the validity of its own internal control blocks and performs other consistency checks to determine if a storage overlay occurred.

If the check option is used, the **storage** command performs the same consistency checks as the heap manager. Instead of issuing a user ABEND, **storage** displays a formatted dump of the block in question. The programmer can use this information to trace the error that caused the overlay.

Reporting usage: In order to reduce storage fragmentation, the heap manager keeps track of freed storage blocks. When the program allocates another block, the heap manager checks for a freed block of the same or larger size. If one exists, the heap manager reallocates it instead of allocating a new storage block. If the report keyword is used, the **storage** command counts the number of allocated and freed storage blocks to produce a report on heap storage usage by the program.

Stack storage: Stack storage is storage that is allocated automatically for **auto** variables, register save areas, and other temporary use by a function. Normally, stack storage is allocated from the stack when a function is called and is returned to the stack when the function returns. However, if the stack overflows (that is, runs out of available storage), the stack manager is called to allocate another block of storage for the stack.

Each stack storage block is called a tract. Just as the heap manager formats heap storage blocks, the stack manager formats tracts. The start of the tract is marked with the word AST and the end with the word EAST. The stack manager also maintains control information about the stack, including information on the current size of the stack and the stack top.

The part of the stack that is allocated for each function is called a DSA (dynamic save area). Part of the DSA is a register save area. These save areas are chained to each other.

Validation: As the stack manager operates, the tract markers, save area chains, and the stack control information are checked for consistency. If an overlay is detected for example, caused by storing outside the boundaries of an **auto** array the stack manager issues a user ABEND.

If the check option is used, the **storage** command performs these same checks. If an overlay is detected, the **storage** command produces a dump of the overlaid storage. Because most validation by the stack manager is delayed until program termination, it is difficult to determine which function in the program is causing the problem. The **storage** command is used to localize the bug.

Reporting usage: If the report option is used, the **storage** command produces a report showing how many DSAs are in use. Usually this corresponds to the number of functions in the calling sequence. The report also shows how much of the stack is unused.

Coprocess stacks: If the program uses the coprocessing functions, **storage** checks and reports on each coprocess's stack separately.

EXAMPLES

The following line-mode example shows the output produced by the **storage** command in an uncorrupted situation.

```
CDEBUG:
storage
```

```
At     SUB1(PGM1)     entry    ------
  SIZE: FREE/USED  SIZE: FREE/USED  SIZE: FREE/USED  SIZE: FREE/USED
    24:    0/116     32:    0/118     40:    0/18      48:    0/69
    56:    0/10      64:    0/7       72:    0/59      80:    0/10
    88:    0/9       96:    0/4      104:    0/3      120:    0/2
   144:    0/5      152:    0/1      160:    0/1      280:    0/1
   512:    1/0      792:    1/0     1152:    1/0     1472:    1/1
  1792:    0/2     8192:    0/1
No corruptions found in heap.

  SIZE  NUMBER  SIZE  NUMBER  SIZE  NUMBER  SIZE  NUMBER  SIZE  NUMBER


   152:    1     168:    1     208:    1     248:    1     256:    1
   296:    2     672:    1
 Total unused space in stack (bytes): 1768
No corruptions found in stack.
```

In this example, neither heap nor stack is specified, so **`storage`** produces a report for both. Also, because neither check nor report is specified, **`storage`** both checks the storage and produces a report on its use. Because no output option (term or file) is specified, the report is displayed on the terminal in the narrow format.

Line 1 shows the current function name and section name.

```
At     SUB1(PGM1)     entry    ------
```

The current hook is entry to the function.

Lines 2 through 8 show the heap storage usage report. Each of the six lines following the subtitle is divided into four columns. There is an entry for each size of allocated storage block. For example, the following entry in line 3, column 1 shows that there are 116 allocated blocks of size 24 and no freed blocks:

```
24:     0/116
```

Similarly, the following entry in line 7, column 1 shows that there is one free block of size 512 and no allocated blocks:

```
512:     1/0
```

Note that the block size shown in the entry is the size known to the heap manager. This size includes the 16 bytes of header and trailer information, plus rounding. For example, the 10-byte allocation discussed earlier in Heap storage is shown as an entry for 32-byte blocks.

The large number of 24- and 32-byte blocks (representing allocations of 1 to 8 bytes and 9 to 16 bytes, respectively) indicates that this program can benefit from memory pooling via **`pool`** and its related functions.

The last part of the heap report, line 9, indicates that **`storage`** detected no overlays in heap storage.

```
No corruptions found in heap.
```

The stack report consists of two lines (12 and 13) of output divided into five columns. Each entry in the report shows the number of DSAs allocated by size. For example, the following entry shows that two 296-byte DSAs are in use:

```
296:     2
```

Line 14 of the report shows the amount of unused stack space, 1768 bytes. Again, **storage** reports that no overlays are detected in the stack.

The next example shows the output from **storage** when a heap storage block is overlaid.

```
CDEBUG: }

storage check heap


At     SUB1(PGM1)         2    00005A
LSCD622 Block (size: 00000078 (120)) at address 00007F20 has been
        overwritten.
        Header Address: 00007F20     Trailer Address: 00007F90
 00007F20   00000078  C8C5C1D7  FCFCFCFC  FCFCFCFC *....HEAP........*
 00007F30   FCFCFCFC  FCFCFCFC  FCFCFCFC  FCFCFCFC *................*
                          .
                          .
                          .
 00007F50   FCFCFCFC  FCFCFCFC  FCFCFCFC  FCFCFCFC *................*
 00007F60 TO 00007F90 SAME AS ABOVE .................
 00007F90   C7C5C1D7  00008310  00000078  FFFFFFFF *GEAP..c.........*
```

Because the check keyword is specified, **storage** does not produce a report. (In fact, because heap storage is corrupted, no usage report is created.) The heap keyword limits the analysis to heap storage. Again, the output is displayed on the terminal in 80-column format.

In this example, a 104-byte storage block is allocated, starting at 0x7f28. The heap manager adds header and trailer information at 0x7f20 and 0x7f90, respectively, resulting in a 120-byte storage allocation. This information is summarized in message LSCD622:

```
LSCD622 Block (size: 00000078 (120)) at address 00007F20 has been overwritten.
        Header Address: 00007F20     Trailer Address: 00007F90
```

The program has not yet changed the contents of the storage block at all. It manages, though, to overwrite the byte immediately following the end of the block, at 0x7f90, with the letter G. Because this changes a heap storage marker from HEAP to GEAP, **storage** detects the change and produces a dump.

In this example, **storage** is uncertain about the exact location of the overlay. Either the program overlays the HEAP marker in the trailer information or the length of the block is overwritten (in which case GEAP can be user data). Therefore, **storage** dumped the first 32 bytes of the storage block, including the header information, and then dumped 32 bytes surrounding the probable end of the block.

The next example shows the output produced by **storage** when it detects a stack overlay.

```
CDEBUG:

storage check stack


At     SUB1(PGM1)         2    00005A
LSCD612 Auto storage control block has been overwritten at 00005FB5.
```

```
00005F98   00000FE0  00005FA8  00000000  40C1E2E3    *......^y.... AST*
00005FA8   0000C6E8  00000000  00000000=>00000000    *..FY............*
```

In this example, **storage** analysis is limited to stack storage by use of the **stack** option.

The overlay is not obvious in this example. Note, however, that **storage** adds a => at the location (0x5fb0) where it detects the overlay. The **storage** command puts this mark in the dump when it can show the location of the overlay exactly.

Assuming that an earlier **storage** check shows no corruption, it is likely that one of the functions in the current calling sequence caused the overlay. (The **where** command can be used to produce a traceback.) From this information, it should be easy to check these functions to determine if one is storing a word of 0x00's outside the boundaries of its stack storage.

SYSTEM DEPENDENCIES

The name of the file that is used when the file keyword is used depends on the operating system. (See DESCRIPTION.)

Under OS/390, the debugger tries to open DBGSTG only once, the first time **storage** is used. If it is not defined, all subsequent **storage** output is written to 'DSN: *userid.pgmname.*DBGSTG' if NOPREFIX is specified in your TSO profile, or 'DSN: *prefix.pgmname.*DBGSTG' if a TSO prefix is specified. That is, you cannot use the **system** command to allocate a data set to DBGSTG after the **storage** command is used.

COMMAND CAN BE ISSUED FROM

PROFILE	no
configuration file	no
Source window prefix	none

SCOPE

The **storage** command is not affected by changes in scope.

RETURN CODES SET

Successful: 0

Unsuccessful: 1

system (CMS)

Execute a CMS Command

ABBREVIATION

sy{stem}

FORMAT

system OPERATING-SYSTEM-COMMAND

DESCRIPTION

The **system** command sends the CMS command specified in the argument OPERATING-SYSTEM-COMMAND to the operating system. Using the debugger

system command is equivalent to calling the **system** function from within a program.

The **system** command cannot be followed by another debugger command on the same line. That is, the arguments to the **system** command are assumed to extend to the end of the line, including any semicolons on the line. The screen is refreshed after completion of execution of the **system** command.

EXAMPLE

system filelist
invokes the CMS FILELIST command.

SYSTEM DEPENDENCIES
See the discussion of this command for TSO if you are running the debugger under TSO.

Note: Some CMS commands, such as HELP and XEDIT, can cause storage allocated by the GETMAIN macro to be freed. If the program being debugged allocates storage by means of GETMAIN, do not invoke these commands. C programs do not use GETMAIN to allocate storage, so you can safely invoke these commands if your program is written entirely in C. △

COMMAND CAN BE ISSUED FROM

PROFILE	yes
configuration file	no
Source window prefix	none

SCOPE
The **system** command is not affected by changes in scope.

RETURN CODES SET:

Successful: 0

Unsuccessful: return code from system

SEE ALSO
"exec (CMS)" on page 221 (CMS version)

system (TSO)

Execute a TSO Command

ABBREVIATION
sy{stem}

FORMAT
system OPERATING-SYSTEM-COMMAND

DESCRIPTION
The **system** command sends the TSO command specified in the argument OPERATING-SYSTEM-COMMAND to the operating system. The OPERATING-SYSTEM-COMMAND argument can be a single command or a CLIST containing TSO commands. Note that in contrast to the **exec** command, if

the CLIST contains SAS/C Debugger commands, they are not passed to the debugger. Using the debugger **`system`** command is equivalent to calling the **`system`** function from within a program.

The **`system`** command cannot be followed by another debugger command on the same line. That is, the arguments to the **`system`** command are assumed to extend to the end of the line, including any semicolons on the line.

EXAMPLE

`system listalc status`.
invokes the TSO command **`listalc status`**

SYSTEM DEPENDENCIES
See the discussion of this command for CMS if you are running the debugger under CMS.

COMMAND CAN BE ISSUED FROM

PROFILE	yes
configuration file	no
Source window prefix	none

SCOPE
The **`system`** command is not affected by changes in scope.

RETURN CODES SET

Successful: 0

Unsuccessful: return code from system

SEE ALSO
"exec (TSO)" on page 219

trace

Trace Program Flow

ABBREVIATION
`t{race}`

FORMATS

Format 1:	trace	
Format 2:	trace	HOOK-TYPE [when (EXPRESSION)] [count N

DESCRIPTION

The **trace** command traces program flow at specified hooks.

Format 1: Format 1 is used without arguments only within an **on** command because you specify the hooks as part of the **on** command syntax.

Format 2: Format 2 of the **trace** command traces program flow at the hooks you request with the HOOK-TYPE argument. See Chapter 12, "Using Debugger Commands," on page 129 for details about the HOOK-TYPE argument.

When a hook is reached, program execution proceeds automatically after the trace. You cannot enter other debugger commands.

You determine the type of line that the **trace** command writes by using the **auto** command. The default is a source line that contains, in addition to a listing of the source code, the line number and the location of the line (the section name and function name where the line is located).

The arguments: A **when** clause can be used to trace conditionally, that is, to trace a line when a hook is reached if the when clause is true.

The argument count N is optional. If count N is specified, the first N–1 times the hook is reached, the count is decremented. The Nth time it is hit, the command is executed. After the Nth time, the command is executed every time the hook is hit.

If a **when** clause is present, a *hit* is counted only if the when expression is true.

A **query** command, issued before the count drops to 1, displays the current value of count. The word count can be abbreviated to cou{nt}.

Identical requests: If a **trace** request is made that is identical to an existing one, the identical request is not installed. This is true whether the request to be installed is within an **on** command or typed in at the command line.

If an identical request is issued and the original request is disabled, the identical request is discarded and the original request is automatically enabled.

If count N is used, the count is ignored in identical requests. If an identical request with a different count is entered, the count field of a **query** is updated with the new count, and a message is produced.

EXAMPLES

trace *
traces program flow at all line-number hooks in all functions.

trace entry
traces program flow at entry to all functions.

trace main 45
traces program flow at line 45 in the **main** function.

trace 53
traces program flow at line 53 of the current function.

trace (comp23) entry
traces program flow upon entry into the **comp23** compilation.

trace func1 entry
traces program flow at entry to the **func1** function.

trace func 23:46
traces program flow at lines 23 through 46 in the **func** function.

SYSTEM DEPENDENCIES

none

COMMAND CAN BE ISSUED FROM

PROFILE	yes
configuration file	no
Source window prefix	t

SCOPE

The **`trace`** command uses command scope to supply default identifiers, function names, and section names.

RETURN CODES SET

Successful: number of the action from the list produced by **`query`**

Unsuccessful: 0

SEE ALSO

- □ “break” on page 199
- □ “disable” on page 212
- □ “drop” on page 213
- □ “enable” on page 216
- □ “ignore” on page 226
- □ “monitor” on page 235
- □ “on” on page 239

transfer

Transfer Debugger/Program Values to CLIST/EXEC Variables

ABBREVIATION

`tran{sfer}`

FORMATS

Format 1:	**`transfer`**	VAR-NAME [FMT-SPEC] **`auto`** [KEYWORD]
Format 2:	**`transfer`**	VAR-NAME [FMT-SPEC] ENVIRONMENT [- N]
Format 3:	**`transfer`**	VAR-NAME [FMT-SPEC] **`value`** EXPRESSION
Format 4:	**`transfer`**	VAR-NAME **`str`** EXPRESSION [,N]
Format 5:	**`transfer`**	VAR-NAME **`dump`** EXPRESSION [,N]
Format 6:	**`transfer`**	VAR-NAME type of EXPRESSION
Format 7:	**`transfer`**	VAR-NAME **`cause`**
Format 8:	**`transfer`**	VAR-NAME **`scope`**

DESCRIPTION

The **transfer** command enables you to assign different character strings containing debugger information or program values to CLIST/EXEC variables. The **transfer** command facilitates communication between the debugger and CLISTs and EXECs. Note that **transfer** is issued to the debugger by a CLIST or EXEC, not from the Command window.

In all the formats above, VAR-NAME is any valid CLIST or EXEC variable name. VAR-NAME is limited to 64 characters. FMT-SPEC is a valid format, which means any valid **printf** specification, provided it begins with a %.

Format 1: Format 1 transfers one or more settings of the **auto** option. FMT-SPEC is specified only for the linesize keyword. If no keyword is specified, all **auto** settings are transferred in a string formatted as follows (where b=blank):

Columns	String
1–6	bbecho/noecho
7	blank
8–11	bbid/noid
12	blank
13–18	bblist/nolist
19	blank
20–28	bbnullptr/nonullptr
29	blank
30–35	bbwrap/nowrap
36	blank
37–45	bbcmacro/nocmacro
46	blank
47–55	bbdumpabs/nodumpabs
56	blank
57–66	bbexececho/noexececho
67	blank
68–76	bbextname/noextname
77	blank
78–82	bbcxx/nocxx
83	blank
84–95	LINESIZE:nnn

Individual settings are transferred by specifying the keyword.

Format 2: Format 2 transfers various ENVIRONMENT settings. The **–N** option is used to call ENVIRONMENT settings that are earlier in the calling sequence. For example, –1 can be used to transfer the value of a setting that is one function earlier in the calling sequence. The ENVIRONMENT settings that can be specified as arguments to the `transfer` command, include the following:

- □ the current address or any address (using the –`N` option) in the calling sequence. This version of Format 2 is

  transfer VAR-NAME [FMT-SPEC] *address* [–N]

  The formatted string contains the appropriate offset in hexadecimal format with a leading `0p`, for example, `0p00123456`. At the return line-number hook, on the jumper's side of the `longjmp` function, an address is not available, and n/a is returned.
- □ the current section name (sname) or any section name (using the – N option) in the calling sequence. This version of Format 2 is

  transfer VAR-NAME *sname* [–N]
- □ the current function name (fname) or any function name (using the –N option) in the calling sequence. This version of Format 2 is

  transfer VAR-NAME *fname* [–N]

  This is similar to using sname. However, the string may contain leading blanks.
- □ the current line number or any line number (using the –N option). This version of Format 2 is

  transfer VAR-NAME [FMT-SPEC] *lineno* [–N]

  The formatted string contains the appropriate decimal line number at a line-number hook, for example, 234. At the prologue and epilogue hooks, calls, entry, or return is returned.
- □ the current offset or any offset (using the –N option). This version of Format 2 is

  transfer VAR-NAME [FMT-SPEC]*offset* [–N]

  The formatted string contains the specified offset in hex with a leading `0p`, for example, `0p0002BE`. At the return hook, an offset is not available, and n/a is returned.

Format 3: Format 3 assigns the value of an expression to a CLIST/EXEC variable. EXPRESSION is any valid scalar expression supported by the `print` command, and FMT-SPEC is any valid format specification supported by the `print` command.

If an FMT-SPEC is given, the string assigned to VAR-NAME is similar to that created for the `print` command, but without the expression : prefix. Strings can be transferred by specifying %s `printf` format specification, but using the str keyword may be more natural. (See Format 4, which is described next.)

If no FMT-SPEC is given, the default output is as follows:

Type 1
defaults to arithmetic.

`short` and `long`	the decimal value using the %d format for signed and %u for unsigned.
`char`	printable: x form escape character: \ n form otherwise: \ xnn
`float`	and `double` %.6E and %.15E, respectively.

Type 2
defaults to pointer. Eight hexadecimal digits are prefixed with a `0p`.

Type 3
structure or union types are not allowed. Only scalar types are allowed.

Type 4
defaults to array. The address of the array is output using the format of Type 2.

Type 5
defaults to address. The address is output using the format of Type 2.

Type 6
defaults to **enum**. The **enum** constant is printed. If there is no **enum** constant corresponding to the value, the value in decimal is printed.

Type 7
defaults to bitfield. The %d or %u format is printed, depending on whether the bitfield is signed or unsigned.

Format 4: EXPRESSION is any valid expression accepted by the **dump** command, which is any address type expression. If the optional, N is not specified, a null-terminated string is copied to VAR-NAME. Otherwise, N bytes, followed by a null byte, are copied. The following examples illustrate this format. Assume **a=abcdef**.

transfer xyz str a (xyz gets the string “abcdef ”)

transfer xyz str a,3 (xyz gets the string “abc”)

Format 5: EXPRESSION is any valid expression accepted by the **dump** command, which is any address type expression. If the optional, N is not specified, the hex representation of a null-terminated string is copied to VAR-NAME. Otherwise, the hex representation of N bytes is copied. Again assuming **a=abcdef"**, the following examples illustrate this format:

transfer xyz dump a (xyz gets the string & “818283848586”)

transfer xyz dump a,3 (xyz gets the string “818283”)

Format 6: Format 6 assigns the type of an expression to a CLIST/EXEC variable. The string assigned to VAR-NAME is a string that describes the expression in C terms. Assume **i**, **cp**, and **sp** are declared as follows:

```
int i;
char * cp;
struct SSS * sp;
```

Then, this format is illustrated as follows:

transfer xyz typeof i(xyz gets the string "int")

transfer xyz typeof cp(xyz gets the string "char *")

transfer xyz typeof sp (xyz gets the string "struct SSS *")}

Format 7: Format 7 uses the VAR-NAME argument to assign the reason the debugger has control to a CLIST/EXEC variable. VAR-NAME is assigned one of the following strings:

NNNN REASON FNAME (SNAME) LINE-SPEC

NNNN monitor MON-DET

0000 signal SIGNAL-NAME

0000 attn

NNNN is the request number in the query list, and it is right-justified with as few digits as needed. The REASON argument is either **`break`**, **`step`**, or **`continue`**. FNAME and SNAME are the function name and compilation of the routine that was stopped in. LINE-SPEC is a line number (in decimal) or calls/entry/return.

MON-DET is what appears for that monitor if you do a **`query`**.

If more than one of the above causes is present, the following hierarchy (in order of most important to least important) is used to output the more important cause:

TERMIN

TERMOUT

SIGNAL

ATTN

MONITOR

BREAK (also generated for **`on`** commands)

RUNTO

CONT

STEP

Format 8: This format uses the scope keyword to transfer the traceback level from the debugger to a CLIST or EXEC variable specified by the VAR-NAME argument. The output from the command is a string containing the traceback level and starting with a minus sign. For example, if the traceback level is five, the output of the command is –5.

EXAMPLES

The EXEC named DUMPTREE, in "Example REXX EXEC Application" on page 52, contains several **`transfer`** commands that demonstrate how you can transfer debugger variable values to CLIST or EXEC variables.

SYSTEM DEPENDENCIES

none

SCOPE

In an expression context, the **`transfer`** command uses command scope to resolve references to all identifiers.

RETURN CODES SET

- Successful: 0
- Unsuccessful: 1

undef

Undefine a Debugger Macro

ABBREVIATION

`u{ndef}`

FORMATS

Format 1: **`undef`** MACRO-NAME

Format 2: **`undef`** *

DESCRIPTION

The **`undef`** command drops debugger macro definitions, which are defined with the **`define`** command. After **`undef`** removes the definition, any further attempts to use the macro name produce a diagnostic message.

Format 1: Format 1 removes a single definition specified by MACRO-NAME.

Format 2: Format 2 removes all debugger macro definitions.

A debugger macro can be redefined by issuing another **`define`** command for the same macro, but with new substitution text. You do not need to use the **`undef`** command on a macro before redefining it.

EXAMPLES

`undef mac1`
undefines the debugger macro **`mac1`** previously defined with the **`define`** command.

`undef *`
undefines all debugger macros.

SYSTEM DEPENDENCIES

none

COMMAND CAN BE ISSUED FROM

PROFILE	yes
configuration file	no
Source window prefix	none

SCOPE

The **`undef`** command is not affected by changes in scope.

RETURN CODES SET

Successful: 0

Unsuccessful: 1

SEE ALSO

"define" on page 210

watch

Assign Expressions to the Watch Window

ABBREVIATION

`wa{tch}`

FORMATS

Format 1: **`watch`** EXPRESSION [%FMT]

Format 2: **watch** EXPRESSION COUNT

DESCRIPTION

The **watch** command is used to assign expressions to the Watch window, which is used to track the value of expressions or areas of storage. The window is not automatically opened.

Format 1: This format is used to assign an expression to the Watch window where its value will be displayed in a manner similar to the **print** command. The EXPRESSION and %FMT arguments are the same as those specified for the **print** command.

Format 2: This format is used to assign an area of storage to the Watch window where its contents will be displayed in a manner similar to the **dump** command. As with the **dump** command, the EXPRESSION argument must specify an address type. The COUNT argument is an integer that specifies the number of bytes to be watched.

EXAMPLES

watch i
assigns the variable **i** to the Watch window where it is displayed in the default format.

watch i %x
assigns the variable **i** to the Watch window where it is displayed in hexadecimal format.

watch ptr 6
displays 6 bytes of memory starting at the address pointed to by **ptr**.

SYSTEM DEPENDENCIES

none

COMMAND CAN BE ISSUED FROM

PROFILE	no
configuration file	no
Source window prefix	none

SCOPE

The **watch** command uses command scope to resolve references to all identifiers.

RETURN CODES SET

Successful: 0

Unsuccessful: 1

SEE ALSO

- “dump” on page 215
- “monitor” on page 235
- “print” on page 241

whatis

Display Type Information

ABBREVIATION

wha{tis}

FORMATS

Format 1: **whatis** EXPRESSION

Format 2: **whatis** (TAG)

DESCRIPTION

The **whatis** command prints the type information associated with the argument.

Format 1: Format 1 prints the type information for the EXPRESSION identifier. The type information includes the type and the length of the expression. EXPRESSION can be any valid expression or a macro. Unlike the other debugger commands that support EXPRESSIONs, the **whatis** command always substitutes the replacement text of a macro, regardless of the setting of the **auto** cmacros keyword.

Format 2: Format 2 prints the type information for a structure, union, or **enum** tag or type defined with a **typedef**. The type information includes the definition of the structure, union, **enum** or **typedef**, and the length of the expression.

As Format 2 shows, if the operand of the **whatis** begins with a left parenthesis, the debugger assumes that a TAG follows. An expression can begin with a left parenthesis if it is escaped. To escape, precede the expression with a backslash.

EXAMPLES

Assume the following declarations when reading Table 14.10 on page 278:

```
struct XYZ {
   int arr[10] [5] ;
   double d;
   char *buf;
   } my_struct;

struct XYZ *xptr;
int numx;
```

The **whatis** command can be used to examine the types of class members by using the C++ binary operator **::** . For example:

```
whatis my_struct::member_name
```

Table 14.10 Examples and Results Using the whatis Command

Example	Printed Result
whatis numx	**auto at 0p12345678 (4 bytes)** **int**
whatis \(numx)	**auto at 0p12345678 (4 bytes)** **int**
whatis &numx	**int ***

whatis (XYZ)	**struct XYZ {(212 bytes)**
	int arr[10][5];
	double d;
	char *buf;
	};

Example	Printed Result

`whatis xptr`	`auto at 0p12345678`
	`(pointer to 212--byte object)`
	`struct XYZ *`

`whatis my_struct::d`	`double d`

SYSTEM DEPENDENCIES
none

COMMAND CAN BE ISSUED FROM

PROFILE	no
configuration file	no
Source window prefix	none

SCOPE
The **`whatis`** command uses command scope to resolve references to all identifiers.

RETURN CODES SET

Successful: 0

Unsuccessful: 1

where

Produce a Traceback

ABBREVIATION
`w{here}`

FORMATS

Format 1:	**`where`**
Format 2:	**`where`** full

DESCRIPTION
Format 1: Format 1 produces a traceback showing functions active at the point in a program where the **`where`** command is issued. You can use the **`where`** command to determine the sequence of functions that are active or whether you are executing in the appropriate section. If command scope is different from run scope, the command scope is marked by an asterisk in the traceback produced by the **`where`** command.

Because source lines and variables are accessed only for active functions, another use of **`where`** is to display a list of active functions to determine what variables can be accessed. (An external variable, of course, can be accessed from any function whose context contains a declaration for it.)

Format 2: Format 2 causes the debugger to print address and offset information in the traceback as well.

EXAMPLES

The following examples show tracebacks that were produced during a line-mode session. If you are using extended names, your traceback will look different. See "auto" on page 196 for more information about extended names.

```
CDEBUG:
where


Calling trace:
    Function        Line    Context
  READIN(MAIN)        50
    MAIN(MAIN)        29*


CDEBUG:
where full


Calling trace:
    Function        Line    Offset     Address    Context
  READIN(MAIN)        50    00007E    000470F2
    MAIN(MAIN)        29*   000086    0004700E
```

SYSTEM DEPENDENCIES

none

COMMAND CAN BE ISSUED FROM

PROFILE	no
configuration file	no
Source window prefix	none

SCOPE

The **where** command is not affected by changes in scope.

RETURN CODES SET

Successful: 0

Unsuccessful: 1

SEE ALSO

"scope" on page 252

window

Perform Window Management Functions

ABBREVIATION

wi{ndow}

FORMAT

window SUBCOMMAND [WINDOW-FIND] [WINDOW-NAME] [SUBCOMMAND-PARAMETERS]

DESCRIPTION

The **`window`** command is used to perform a wide variety of window management functions. Furthermore, it may be issued three ways in line mode and five ways in full-screen mode:

- □ in line mode
  - □ from a PROFILE
  - □ from a configuration file
  - □ from the command prompt
- □ in full-screen mode
  - □ from a PROFILE
  - □ from a configuration file
  - □ from the Command window
  - □ from the Log window
  - □ with a PF key.

The **`window`** command is always issued with a SUBCOMMAND argument, and sometimes with parameters to the subcommand, SUBCOMMAND-PARAMETERS. The subcommands that are available to you depend on how you issue the command. Table 14.11 on page 281 lists the subcommands and where they are valid.

Table 14.11 window Subcommands

		Configuration	**Command**	**Line**
Subcommand	**PROFILE**	**File**	**Window**	**Mode**
`autopop`	no	yes	yes	no
`border`	no	yes	yes	no
`clear`	no	no	yes	no
`close`	no	no	yes	no
`color`	no	yes	yes	no
`config`	no	yes	no	no
`context`	no	yes	yes	no
`find`	no	no	yes	no
`memory`	no	yes	no	no
`move`	no	no	yes	no
`next`	no	no	yes	no
`off`	yes	no	yes	no
`on`	no	yes	yes	no
`open`	no	yes	yes	no
`previous`	no	no	yes	no
`resize`	no	no	yes	no
`scroll`	no	yes[1]	yes	no

Subcommand	PROFILE	Configuration File	Command Window	Line Mode
top	no	no	yes	no
trace	no	yes	yes	no
zoom	no	no	yes	no

1 Format 2 of the scroll subcommand cannot be specified in the configuration file.

Each of these subcommands is described in detail later in this section.

WINDOW-NAME argument

Some of these subcommands must be used with a WINDOW-NAME argument, which is used to specify the window to which the command applies. If WINDOW-NAME is required, it must be one of the following:

- Command
- Config
- Dump
- Keys
- Log
- Message
- Popup
- Print
- Register
- Source
- Status
- Termin
- Termout
- Watch
- < >.

< > is used as a placeholder to specify the window the cursor is in. As < > is used interactively, it may not be specified in **window** commands present in the configuration file. Any further restrictions are mentioned in the discussions of the various subcommands.

The windows that can be specified with the WINDOW-NAME argument fall into four classes:

In Class 1 these windows are always present:

- Command
- Log
- Source
- Status.

In Class 2 these windows are under debugger control for error correction or reporting:

- Message
- Popup.

In Class 3 you control the presence of these windows; there can be one instance only of each:

- □ Config
- □ Keys
- □ Register
- □ Termin
- □ Termout
- □ Watch.

In Class 4 you control the presence of these windows; there can be multiple instances of each:

- □ Dump
- □ Print.

In **`window`** subcommands that permit a Class 4 name, the name can be specified only if it unambiguously identifies the window. For example, if there are two Dump windows open, the following command is ambiguous:

window close dump

However, the same command would not be ambiguous if only one Dump window was open.

To close a Dump window in the situation where two or more Dump windows are open, you can position the cursor in the window and press the PF15 key, which is assigned the following **`window`** command by default:

window close < >

Or, you could type this command in the Command window and then move the cursor into the window before issuing it with the ENTER key.

WINDOW SUBCOMMANDS

Each of the **`window`** subcommands is discussed in detail in the following paragraphs.

`autopop`

The **`autopop`** subcommand is used to select the autopop status for a window. (This can also be accomplished with the Config window.) Autopop status is either on or off. If the autopop status is set to on, the window is automatically made the top window; that is, the window is popped to the top and the cursor is placed inside the window whenever output is sent to the window, provided the window is at least partially obscured. If the window is completely unobscured, autopop does not cause the window to become the top window. If specified for windows for which there can be multiple instances, the attribute applies to all instances.

The format of the **`autopop`** subcommand is as follows:

autopop WINDOW-NAME on | off

The WINDOW-NAME argument specifies the window that is to have its autopop status set, and the on/off argument sets the status. For example, the following command will set the autopop status of the Dump window to on:

window autopop dump on

In a configuration file, the WINDOW-NAME argument can be any class of window; however, when issued during a session from the Command window or a

PF key, WINDOW-NAME must be Class 1 or 3, or the <> placeholder. If the autopop status of a Class 4 window is changed during your session, the new status applies only to the specific window to which the command was applied. The autopop status in the Config window must be used to change the status for any future instances of this type of window.

By default, autopop is off for all windows except the following:

- Termin
- Termout
- Watch.

border

This subcommand specifies the characters to be used to form the borders of windows with borders. The format of the **border** subcommand is as follows:

border "STRING"

The six characters used to make up the border are specified as the STRING argument in the form of a string literal. For example, the following STRING argument specifies the characters `"++++|-"`:

window border "++++|-"

The first four characters are used for the four corners (top left, top right, bottom right, and bottom left). The fifth is used for the vertical borders, and the last for the horizontal borders. If this command is not present, or if a null-string is present, the debugger uses suitable default characters. Border color is controlled by the first C-A-I-TRIPLET in the **window color** command associated with that window. (See the **color** subcommand later in this section.)

clear

The **clear** subcommand clears the Command, Log, or Termout windows. The format of the **clear** subcommand is as follows:

clear WINDOW-NAME

This command cannot be issued from a configuration file and WINDOW-NAME must be either Command, Log, or Termout. For example, during a session you can issue the following command to clear the Log window:

window clear log

close

The **close** subcommand can be used to close any Class 3 or 4 window. This subcommand is not valid in a configuration file and can only be used during a full-screen session. The format of the **close** subcommand is as follows:

close WINDOW-NAME

For example, to close the Register window, you can issue the following command:

window close register

If this command is used to close a Class 4 window (Dump or Print), there must be only one active instance of the window. Or you must use the <> placeholder character in the command and specify the desired window to close by positioning the cursor within its borders. For example, if you have two Print windows open, you can enter the following command in the Command window, and then, before you press the ENTER key, position the cursor in the Print window you want to close.

window close < >

When you press the ENTER key, the command is issued and the Print window is closed.

`color`

This subcommand is used to set the color, attribute, and intensity of the named window. The Configuration window is normally used to control these settings; however, the **`color`** subcommand can also be entered from the Command window or typed directly into a configuration file. The format of the **`color`** subcommand is as follows:

color WINDOW-NAME C-A-I-TRIPLET C-A-I-TRIPLET . . .

The WINDOW-NAME argument can be any valid window name. The C-A-I-TRIPLET arguments are used to specify the color, attributes, and intensity of each area of a window, including the border, which is customizable. One C-A-I-TRIPLET is required for each area of the window.

The first position of the C-A-I-TRIPLET triplet specifies the color of the area and can be any of the following:

- white
- blue
- magenta
- red
- yellow
- green
- cyan
- none.

The second position of the C-A-I-TRIPLET specifies the display attribute for the area and can be any of the following:

- blink
- reverse (for reverse video)
- underscore
- none.

The third position of the C-A-I-TRIPLET specifies the intensity of the area and can be either of the following:

- low
- high.

C-A-I-TRIPLETs are specified using the first letter of each component. For example, “r b l” specifies a red, blink, low triplet. You can also use a colon to specify that a characteristic in the C-A-I-TRIPLET not be changed. For example, “: r h” specifies current color, reverse video, and high intensity.

Specifying colors, attributes, or intensities on terminals that do not have the support causes the setting to be ignored.

Table 14.12 on page 286 shows how many triplets are required for each window and what they are used for.

Table 14.12 C-A-I-TRIPLET Arguments by Window

Window	Number of Triplets	Areas Affected	Protected?
Command	3	Command window border	yes
		Cdebug: prompt	yes
		command line input area	no
Config	3	Config window border	yes
		titles and protected text	yes
		unprotected text	no
Dump	4	Dump window border	yes
		various prompts	yes
		dump parameters	no
		hex and character dump	yes
Help	1	Help window border	yes
Keys	3	Keys window border	yes
		key names	yes
		ISPF? and key definitions	no
Log	5	Log window border	yes
		debugger output	no
		echoed commands	no
		echoed terminal input	no
		echoed terminal output	no
Message	2	Message window border	yes
		message	yes
Popup	4	Popup window border	yes
		message	yes
		prompt	yes
		input area	no
Print	4	Print window border	yes
		various prompts	yes
		print parameters	no
		value of expression printed	yes
Register	2	Register window border	yes
		register names and values	yes
Source	5	Source window border	yes
		various prompts	yes
		module name and line number	no
		line number and text areas	no
		line stopped at	no

Window	**Number of Triplets**	**Areas Affected**	**Protected?**
Status	5	Status window border	yes
		help information	yes
		reason for break	yes
		run scope	yes
		command scope	yes
Termin	5	Termin window border	yes
		various prompts	yes
		various settings	no
		"Read..."/"Cont..." prompt	yes
		input area	no
Termout	5	Termout window border	yes
		various prompts	yes
		various settings	no
		"More..." prompt	yes
		output area	yes
Watch	5	Watch window border	yes
		various prompts	yes
		watch parameters	no
		drop prefix area	no
		watch name and value	yes

When using the **`color`** subcommand, you must provide the number of C-A-I-TRIPLET arguments shown in the second column of Table 14.12 on page 286. For example, the Keys window requires three C-A-I-TRIPLET arguments to set the characteristics of the border, key name pad, and key definition text pad as follows:

window color keys m r l y n l r n l

When issued from the Command window or the config file, this command sets the following Keys window characteristics:

border
: magenta, reverse video, low intensity

key name pad
: yellow, none, low intensity

key definition pad
: red, none, low intensity

In this example, both the border and key name pad are protected fields, which means that they do not accept user input. However, for the Keys window, the key definition text pad is not protected and does allow input.

`config`

This subcommand is used to configure, establish the initial size and position of, the named window. The Config window is normally used to control these settings; however, the **`config`** subcommand can also be typed directly into a configuration file. The format of the **`config`** subcommand is as follows:

config WINDOW-NAME position ROW-POS COL-POS size HEIGHT WIDTH
border | noborder

The WINDOW-NAME argument can be any valid window name. The ROW-POS and COL-POS arguments, which must follow the position keyword, are used to specify the position of the upper-left corner of the window. Window size is controlled by the HEIGHT and WIDTH arguments, which also must follow the appropriate keyword, size. The border or noborder keyword is used to turn the window border either on or off.

For window position arguments, the top left corner of the screen is considered to be the origin (0,0). ROW-POS and COL-POS are integers representing the row and column position of the top left corner of the window. HEIGHT and WIDTH, which are also integers, represent the size of the window in terms of rows and columns. Borders, if any, must be included in the size.

Any of ROW-POS, COL-POS, HEIGHT, or WIDTH arguments can be allowed to assume a default value for the window if you specify a colon instead of an integer. If the values specified cause the window to be placed outside the screen (exceed screen dimensions), the position and size is adjusted accordingly. If either the ROW-POS or HEIGHT argument is specified as an integer, it is not advisable to specify a colon for the other argument. The default assumed by the debugger may cause ROW-POS or HEIGHT to be forcibly adjusted to meet screen dimensions. However, WIDTH and COL-POS defaults will not be affected. Similarly, it is not advisable to specify only one of the COL-POS and WIDTH arguments and use the default for the other.

If the configuration is saved, actual values are written out even if defaults were specified (with a colon) in the original **`window config`** command. The only exceptions are certain parameters of the following windows:

Dump window
Window height is specified at configuration time.

Print window
Window height is specified at configuration time.

Message window
The default colon symbol appears for ROW-POS and COL-POS if they were not specified.

The HEIGHT argument is ignored. If a ROW-POS argument is specified, the window appears on that row; otherwise, it is centered vertically. If a COL-POS argument is specified, the window appears on that column; otherwise, it appears centered horizontally. The height is based on the amount of text in the message.

Popup window
The default colon symbol appears for COL-POS if it was not specified.

The HEIGHT and WIDTH arguments are ignored. If a ROW-POS argument is specified, the window appears on that row; otherwise one of the rows in the middle of the screen is used. If a COL-POS argument is specified, the window appears on that column; otherwise it is centered vertically.

all other windows
Various name-dependent restrictions imposed for the height and width of a window are shown in Table 14.13 on page 289 (terminal height in rows = H, terminal width in columns = W). The maximum height and the maximum width of these windows is screen height and screen width.

Table 14.13 Window Size Defaults and Restrictions

Window	Min. Height	Def. Height	Max. Height	Min. Width	Def. Width[1]	Max. Width	Def. Border
Command	1	1	1	24	W	W	no
Config	3	H	104	12	W	84	yes
Dump	5	H/2	H	51	W	W	yes
Help	4	H/2	H	20	W	W	yes
Keys	3	14	27	12	80	94	yes
Log	3	.45X(th-2)	H	10	W	W	yes
Print	5	H/2	H	49	W	W	yes
Register	3	11	11[2]	8	75	75	yes
Source	4	rest[3]	H	32	W	W	yes
Status	1	1	1	10	W	W	no
Termin	5	5	5[4]	30	W	W	yes
Termout	5	H/2	H	20	W	W	yes
Watch	5	H/3	H	20	W	W	yes

1 If the terminal width is greater than the maximum width, then the default width is the maximum width.
2 This assumes default border.
3 rest = terminal-height - (command-window-height + log-window-height + status-window-height)
4 This assumes default border and scale.

If there is no **`window config`** command for a particular window in the configuration file, or if the command contains a syntax error, default parameters are used.

`context`

This subcommand controls the amount of context information provided by source lines around the highlighted line in the Source window. The format of the **`context`** subcommand is as follows:

context source N M

Except when near the top or near the bottom of a file, the debugger maintains a minimum of N source lines around (above and below) the highlighted line. If the window height is less than 2*N+1, it is not physically possible to do this; in such cases the debugger maintains as much context as possible. N can take any value between 0 and 254. To specify the default value, which is 2, use a colon.

The M argument determines when the highlighted line is centered to maximize context information. If the next line to be highlighted is at least M lines above the top or M lines past the bottom of the window, it is centered. M can take any value between 0 and 0x7fffffff. To specify the default value, which is 10, use a colon.

`find`

The **`window find`** subcommand is used to search for strings and is supported in the following windows:

- Browse
- Log
- Source.

The following format is used with the **window find** subcommand:

window find WINDOW-NAME

The WINDOW-NAME argument can be any of the following:

- □ <>
- □ browse
- □ log
- □ source.

If the **<>** WINDOW-NAME argument is used, the position of the cursor determines the window to which the command is applied. If a window name is specified as the **window-name** argument, the position of the logical cursor is used to determine the starting point of the search, if the search is cursor dependent.

intercepts

This subcommand specifies the status of the input and the output intercepts and the processing of intercepted input or output. The format of the **intercepts** subcommand is as follows:

intercepts "STRING"

The STRING argument is eight characters long; each position controls one intercept. Table 14.14 on page 290 lists the meaning of each position and the valid values for each character.

Table 14.14 intercepts Subcommand STRING Argument

Position	Significance	Value	Meaning	Default
1	input intercept	y	input intercepted	y
		n	not intercepted	
2	input logging	y	input logged	n
		n	not logged	
3	input scale	y	scale present	y
		n	not present	
4	output intercept	y	output intercepted	y
		n	not intercepted	
5	output logging	y	output logged	n
		n	not logged	
6	output display	n	do not display	i
		y	display	
		i	display immediately	
7	output pause	y	pause when screen is full	y
		n	do not pause	
8	output scale	y	scale present	y
		n	not present	

memory

This subcommand allocates memory for the buffers of the Command, Log, and Source windows. The format of the **memory** subcommand is as follows:

memory MEMBRW MEMCMD MEMLOG MEMSRC

The MEMBRW argument specifies the memory allocated to the Browse window, MEMCMD for the Command window, MEMLOG for the Log window, and MEMSRC for the Source window. Table 14.15 on page 291 lists the default and minimum memory sizes in bytes for each of these windows (H=terminal height).

Table 14.15 MEMCMD Argument Values

Window	Minimum Memory	Default Memory
Browse	255 * H	max (255 * H, 6000)
Command	360	1000
Log	1000	12000
Source	255 * H	max (255 * H, 12000)

move

This subcommand is used to move windows during a session. The format of the **move** subcommand is as follows:

move WINDOW-NAME

The WINDOW-NAME argument must be < >. Place the cursor on the window to be moved and execute the command. This puts the debugger in move mode; MOVE appears at the bottom right corner of the window being moved. Move the cursor in the new location for the window, and then press any PF key or ENTER to terminate move mode, reposition the window, and resume normal operation.

next

As explained in "Some Window Basics" on page 12, the debugger uses a stack to keep track of open windows. You can move through this stack of windows in either direction. The **next** subcommand moves the top window to the bottom of the stack. Thus, the next window below the window just moved becomes the top window. The format of the **next** subcommand is as follows:

next

See **previous** subcommand for comparison.

off

This subcommand is used to terminate full-screen mode. The format of the **off** subcommand is as follows:

off

It has no effect if the debugger is in line mode. When this command is processed, all other unprocessed input is discarded with the exception of other commands on the same line as the **window off** command; these are processed in line mode. (See the **on** subcommand.)

on

This subcommand switches to full-screen mode. The format of the **on** subcommand is as follows:

on

The configuration used is determined by the configuration last in effect (default configuration, if entering full-screen mode for the first time), modified by any

config subcommands. If you are re-entering full-screen mode, the state of the following windows will be the same as when the **off** subcommand was specified:

- □ Log
- □ Command.

Only class 1 type windows will reappear.

The **on** subcommand has no effect if you are already in full-screen mode.

Any other command specified on the same line as the **window on** command in line mode is ignored.

open

This subcommand opens the named window using the parameters in effect for it. The format for the **open** subcommand is as follows:

open WINDOW-NAME

WINDOW-NAME can be any Class 3 or 4 window. If it is a Class 3 window, it is automatically made the top window. Window classes are discussed earlier in this section.

previous

As explained in “Some Window Basics” on page 12, the debugger uses a stack to keep track of open windows. You can move through this stack of windows in either direction. The **previous** subcommand positions the bottom window at the top of the stack and makes it the top window. The format of the **previous** subcommand is as follows:

previous

See **next** subcommand for comparison.

resize

This subcommand is used to resize a window during a full-screen session. The format of the **resize** subcommand is as follows:

resize WINDOW-NAME

The WINDOW-NAME argument must be < >. Place the cursor on the window to be moved and execute the command. This puts the debugger in resize mode; RESIZE appears at the bottom right corner of the window being moved. Move the cursor appropriately, and then press any PF key or ENTER to terminate resize mode, reposition the window, and resume normal operation.

The debugger uses an algorithm to determine the new size for the window. The window is divided into equal, or nearly equal, quarters. To expand or contract the window, window edges closest to the original cursor point are moved so that they are the same distance from the terminating cursor point as they were from the original cursor point. If the original cursor point is equidistant from either the right or left edge, or the top or bottom edge, then the edge closest to the terminating point is chosen. However, if the terminating point is still equidistant, then growth or shrinkage occurs in only one dimension.

scroll

The **scroll** subcommand has two formats:

Format 1: **scroll** amount SCROLL-AMOUNT

Format 2: **scroll** WINDOW-NAME up/down/right/left

Format 1 is used to set the scroll amount, and Format 2 is used to scroll a window. Format 2 cannot be used in a configuration file.

Valid values for the SCROLL-AMOUNT argument are

- □ cursor
- □ half
- □ max
- □ page (screen is supported as a synonym for page)

All amounts except for max can be set in a configuration file. However, you can specify any of the values, including max, in the Config window and then issue the appropriate **`window scroll`** command using any of the methods described. The debugger scrolls by max.

Format 2 scrolls the window named by the WINDOW-NAME argument. The amount scrolled is determined by the scroll amount in the Status window. The following windows may be scrolled up and down:

- □ Command
- □ Config
- □ Dump
- □ Keys
- □ Log
- □ Print
- □ Register
- □ Source.

The following windows may be scrolled left and right:

- □ Config
- □ Keys
- □ Log
- □ Print
- □ Register
- □ Source
- □ Status.

Format 2 of the **`scroll`** subcommand can also be used to change command scope. By issuing a **`window scroll < > up`** or a **`window scroll status up`** command, the command scope can be moved up in the calling sequence. Similarly, the down keyword causes command scope to be moved down in the calling sequence. See "Using the Status Window" on page 20 more information.

The WINDOW-NAME argument can be any of these windows or the < > placeholder.

The logical cursor moves with the text being scrolled; cursor movement is naturally limited by window boundaries. The amount scrolled is determined by the scroll amount specified in the Status window. If the scroll amount is cursor, the position of the logical cursor is used to determine the scroll amount.

`top`

This subcommand positions the named window at the top of the stack. The format of the command is as follows:

top WINDOW-NAME

The WINDOW-NAME argument can be any valid window name or < >. The physical cursor appears in this window at the place of the logical cursor when it becomes the top window. (See also the **`next`** and **`previous`** subcommands.)

`trace`

This subcommand controls the production of trace lines in the Log window. The format of the **`trace`** subcommand is as follows:

trace log on | off

The **`window trace log`** subcommand is off by default. If turned on, trace lines are produced in the following cases (similar to when the debugger produces a trace or list line in line mode):

- □ each time the debugger gives you control
- □ at the **`n`** – 1 hooks at which the debugger does not give you control, in the **`step n`** or **`continue n`** case
- □ when an **`on`** is executed provided **`auto id`** is on
- □ when a **`monitor`** request occurs.

This subcommand also controls the production of the "context of ..." message at prologue and epilogue hooks, which appears only when the debugger gives you control.

The **`window trace`** subcommand has no control over trace lines produced by the **`trace`** command; such lines are produced irrespective of the setting of this command.

`zoom`

This subcommand zooms the named window to fill the screen; if already zoomed, it unzooms the window. The format of the **`zoom`** subcommand is as follows:

zoom WINDOW-NAME

PART 5

Appendices

APPENDIX 1 Error Handling

Introduction

The SAS/C Debugger writes messages when it encounters syntax or specification errors and other conditions, such as an invalid debugger file. Some error messages are displayed in the Log window; however, as described in "Pop-Up and Message Windows: Error Processing" on page 35, messages are also displayed in the Popup and Message windows.

For syntax and specification errors, depending on whether the issued command is an argument to an **on** command (either a value of CMD or part of CMD-LIST), the debugger takes various actions that include writing a message to identify the error.

Invalid Commands

If the command name is not recognized, the debugger writes the message **Invalid command name**, and the invalid command is rejected. Type a question mark (**?**) for a list of valid command names.

If you use an incorrect abbreviation for a command, the same message occurs. In both cases, you need to reenter the command and any argument values.

Note that the debugger does not check the syntax of the command list of an **on** command or a **when** clause until it executes the command. If you make a syntax error in the command arguments to the **on** command or in a **when** clause, the debugger does not write an error message until it tries to execute the command.

Valid Commands

If the command name is recognized by the debugger but the number of argument values is not correct or if the values of the arguments are not valid, then the debugger writes a message that identifies the problem.

In some cases, the debugger performs the command and discards the invalid argument or arguments. For example, if you type **where** *xxx*, unless *xxx* is the keyword

`full`, the debugger outputs a traceback, discards *xxx*, and writes a message such as "Input discarded: *xxx*."

In other cases, both the valid command and invalid arguments are discarded, and the debugger writes one or more messages.

Debugger Internal Errors

Debugger internal errors are of two types: recoverable and nonrecoverable. If the debugger detects a nonrecoverable error, it displays the following message:

```
Debugger unrecoverable internal error: <symbol>
```

where *symbol* is an identifying symbol for the error. Then the debugger calls the **`abort`** function in order to produce a traceback and terminate. If the debugger detects a recoverable error, it displays the following message:

```
LSCD998: Debugger recoverable internal error: <symbol>
```

This message is followed by a traceback. Again, *symbol* is an identifying symbol for the error. The debugger does not terminate, but the debugger command that produced this message may not have been executed. These messages are often a result of the debugger's storage being overwritten by the program being debugged. However, if you think that the problem is with the debugger, contact your SAS Installation Representative for C Compiler products. In the problem description, include both the value of *symbol* and the traceback information.

APPENDIX

2

Character Set Defaults for Special Characters

Because C accepts several special characters on input that may not be available on all terminals and printers, alternate representations for these characters can be customized by each compiler site at installation time. If alternate representations are customized for your site, then they are used by the SAS/C Debugger, as well as the compiler and object module disassembler (OMD).

The special characters are braces, brackets, the circumflex, the tilde, the backslash, the vertical bar, the pound sign, and the exclamation point. Determine if your site has customized values for these characters and what the values are. Otherwise, the default representations listed in Table A2.1 on page 299 are in effect.

Keep in mind that these alternate representations for characters apply only to debugger commands. Also, note that you can subsitute (| |) for [ ]and (< >) for { }.

Note: The columns labeled "Primary" and "Alternate" in Table A2.1 on page 299 contain the primary and alternate debugger input representations for the special characters. △

Table A2.1 Default Representations for Special Characters

Character	**Primary**	**Alternate**
left brace	0cx0	0x8b
	{	{
right brace	0xd0	0x9b
	}	}
left bracket	0xad	0xad
	[	[
right bracket	0xbd	0xbd
	]	]
circumflex (exclusive or)	0x5f	0x71[1]
	¬	^
tilde	0xa1	0xa1
	~	~
backslash	0xe0	0xbe
	\	≠
vertical bar (inclusive or)	0x4f	0x6a
	|	|

Character	Primary	Alternate
pound sign	0x7b	0x7b
	#	#
exclamation point	0x5a	0x5a
	!	!

1 The caret (^) is usually associated with this value.

APPENDIX 3

Debugger I/O Exit Routines

Introduction

You can provide exit routines to handle debugger line mode input or output. These exits enable you to use the debugger under circumstances where using its standard terminal input implementation, output implementation, or both is not appropriate or sufficient. These debugger I/O exits must be written in assembler language. Further, they cannot cause the C environment of either the debugger or the program being debugged to be reentered. If they do, the results are unpredictable. Therefore, exit routines must not invoke any C functions.

Note: The interface to debugger exit routines might change in a future release of the SAS/C Compiler. If the interface does change, you might have to modify some or all of your exit routines. However, any future interface will support at least the functionality that is described in the following sections. △

Also note that I/O exit routines cannot be used in full-screen mode. You must switch to line mode by issuing a `window off` command before invoking an I/O exit routine.

Using Debugger Exit Routines

Debugger exit routines provide flexibility to the debugging environment. Output from the debugger can be modified, for example, to produce a different prompt for each OS/390 subtask that is being debugged, thus suffixing the standard prompt (which does not go through the output exit because it is produced by the PROMPT=string amparm). Input and output can be routed to a different virtual machine. An assembler subroutine can pass debugger commands from the program to the debugger via the exit routines, thus providing the functionality of embedding debugger commands in source code.

Invoking Debugger Exit Routines

Each debugger exit routine is a separate load module. The names are L$UDBIN and L$UDBOUT. Under CMS, place these modules in the L$CUSER LOADLIB that you create. Under OS/390, place these modules in SASC.LINKLIB. At debugger initialization time, the debugger looks for each of these load modules. If neither is present, then no further processing is done for that exit routine, and debugger execution proceeds without using that exit routine. Thus, each routine can be used independently. If you do use both routines, they can communicate as described in the following sections.

Each routine is invoked by using standard IBM linkage conventions. Any registers that are used by the routine should be saved (the standard save area that is addressed by register 13 can be used for this) and restored on exit, except as otherwise described below. On entry, register 1 addresses a parameter block. The first parameter in this block is always a call type flag. Other parameters can differ for different call types and these are described in more detail in "Sample Debugger Exit Routines" on page 302. The call type flag can have the values 4 (initialization), 0 (input or output call), or 8 (termination).

The debugger output exit routine can be called to handle attention interrupts that are detected by the debugger. The output exit provides a method for generating simulated attention interrupts. At initialization, the output exit routine can request that the debugger ignore all attention interrupts except those that are generated by the exit routine.

The output exit routine (L$UDBOUT) is invoked first. If you want the two exit routines to communicate, L$UDBOUT is responsible for setting up that communication.

Dummy Debugger Exit Routines (OS/390 Only)

As installed under OS/390, the SAS/C run-time library contains dummy debugger exit routines. These routines do nothing. Your installation can replace these load modules with your own exit routines or delete them. If they are deleted, your operating environment might produce messages when the debugger is invoked (for example, OS/390 message CSV003I), but the debugger will function normally.

Sample Debugger Exit Routines

The installation tape also contains sample debugger exit routines. These are in the sample source libraries LSU MACLIB (CMS) and SASC.SOURCE (OS/390), with member names L$UDBIN and L$UDBOUT. These samples can provide useful information for writing your own exit routines.

Initialization Call: Output Exit Routines

When the output exit routine receives control for initialization, register 1 points to the following parameter block:

0	a fullword with the value 4 (= initialization)
4	the address of an 8-byte area
8	the address of a flag byte

12	the value from the program's CRAB user word 1 (CRABUSR1)
16	the value from the program's CRAB user word 2 (CRABUSR2)
20	the value from the program's CRAB user word 3 (CRABUSR3)
24	the value from the program's CRAB user word 4 (CRABTUSR).

Word 2 addresses an 8-byte area. The first 4 bytes of this area contain the address of a debugger subroutine that is called asynchronously to present an attention interrupt. This subroutine is called the attention simulation routine. The second 4 bytes contain a value that must be passed to the attention simulation routine when a simulated attention is to be generated. See "Attention Handling" on page 303.

Word 3 addresses a flag byte. On entry, this byte is 0. If the output exit sets this byte to a nonzero value, then the debugger recognizes only the attention interrupts that are generated by a call to the attention simulation routine.

The values from the CRAB user words are probably 0 because the debugged program has not yet received control . However, the values may not always be 0.

After initialization, the output exit routine returns four values. These values must be returned in registers 15, 0, 1, and 2. The values are the new values for the CRAB user words. The CRAB user words are used to communicate between the output exit routine and the input exit routine. For example, you may return the address of a communication area for CRABUSR1. Values for CRAB user words that you do not plan to use must be returned unchanged. Do this by loading the appropriate registers from the input parameter block. This indicates that the output does not need to use the word or words and allows some other part of the system to use them. The returned values from registers 15, 0, 1, and 2 are placed in the CRAB user words of the program being debugged.

If the debugger is used in batch mode, the debug log file always opens even if the output exit routine is present, which may suppress all output to the log file. An OPEN failure terminates debugger activity. Therefore, under OS/390 in batch mode, you should provide a JCL statement for DBGLOG, even if the output exit suppresses all output to it. You can use a DUMMY DD statement.

Attention Handling

When the attention simulation routine is called, the following values must be in the general registers:

R1 — contains the value that is passed in the second 4 bytes of the area addressed by word 2 of the initialization call parameter list.

R13 — contains the address of a 72-byte save area.

R14 — contains the return address for the simulated attention routine.

R15 — contains the address of the simulated attention routine. The simulated attention routine returns a *status code* in R15:

- 0 — indicates that the debugger accepted the interrupt for processing.
- 4 — indicates that the debugger ignored the interrupt.

The output exit routine receives control when an attention prompt is needed. The attention prompt is usually issued by the debugger when an attention interrupt is pending, and then the user issues a second attention interrupt. In the usual case (that is, when the I/O exits are not used), the debugger prompts the user to decide whether to continue execution or to abort the program. When the I/O exits are in use, the output exit routine must decide how to handle the second interrupt.

When the output exit routine receives control for an attention prompt call, register 1 points to the following parameter block:

0	a fullword with the value 12 (= attention prompt)
4	0
8	0
12	the value from the program's CRAB user word 1 (CRABUSR1)
16	the value from the program's CRAB user word 2 (CRABUSR2)
20	the value from the program's CRAB user word 3 (CRABUSR3)
24	the value from the program's CRAB user word 4 (CRABTUSR).

The exit returns one of the following return codes:

0	indicates that the debugger should issue the prompt and wait for a response, which is then acted upon normally.
4	indicates that the debugger should ignore this attention. This code should be returned either if the exit routine handles the interrupt or if the attention should be completely ignored.
8	indicates that the debugger should abort the program.

Note that the output exit routine receives an attention prompt call in response to a simulated attention. An output exit routine that supports simulated attentions should be prepared to handle this situation.

Initialization Call: Input Exit Routines

The input exit routine receives control for initialization after the output exit initialization call. The output exit routine may be (and probably is) called multiple times before the input exit routine is invoked.

When the input exit routine receives control for initialization, register 1 points to the following parameter block:

0	a fullword with the value 4 (= initialization)
4	the value from the program's CRAB user word 1 (CRABUSR1)
8	the value from the program's CRAB user word 2 (CRABUSR2)
12	the value from the program's CRAB user word 3 (CRABUSR3)
16	the value from the program's CRAB user word 4 (CRABTUSR).

The values from the CRAB user words are those that are set by output exit initialization, those that are set by an assembler routine in the program itself that is invoked before any debugger input is wanted, or they may be 0 if no output exit routine is present. However, the values may not always be 0 in the last case.

After initialization, the input exit routine returns 0 in register 15 to indicate successful initialization. If a nonzero value returns, the debugger assumes that the input exit routine is not to be invoked again, even for termination.

Note that if the input exit routine does its own terminal I/O, it must also generate any necessary prompt. The `Cdebug:` prompt is only generated when the input exit routine is not used.

Normal Call: Output Exit Routines

When the output exit routine receives control for a normal call, register 1 points to the following parameter block:

0	a fullword with the value 0 (normal call)
4	the address of the current debugger output line
8	the length (**`strlen`**) of the current output line
12	the value from the program's CRAB user word 1 (CRABUSR1)
16	the value from the program's CRAB user word 2 (CRABUSR2)
20	the value from the program's CRAB user word 3 (CRABUSR3)
24	the value from the program's CRAB user word 4 (CRABTUSR).

The output exit routine can choose how to use the current output line. Thus, it may write it to a file, send it to another user ID, save it for later access by the C program proper, discard it, or do any combination of these things. Note that if it does save the current output line for later access by another program, the output exit routine must copy the actual message text and not just save the pointer to it. This is because the storage that contains the message may have been reused or it may be inaccessible when the C program receives control again.

In addition to its own use of the line, the output exit routine controls the debugger's use of the line. It has the following three choices:

- □ Tell the debugger to write the original line to its standard output file.
- □ Tell the debugger to write a modified line.
- □ Tell the debugger to write no line at all.

The output exit routine indicates its choice by the values that it returns in register 15 and register 0.

The return value in register 15 is the address of the output line to be written or 0 (C NULL) when no line is to be written. If the original line is to be written, register 15 should be set to the second input parameter. (Recall that the first parameter is call type.) If a new or modified line is to be written, register 15 should be set to its address. Any nonaddress bits in register 15 must be set to 0. The instruction LA 15,0(,15) accomplishes this. Note that if the output exit routine decides to edit the original line, it must copy the line to a work area. The original line may be a constant string in read-only storage. Attempting to modify the original line directly has unpredictable consequences. Any new or modified line must be delimited by the C string delimiter of hex 00. Also note that the original line may not be null terminated. The length of the output line should always be obtained from the parameter list. Finally, note that the maximum length of a new line is 255 bytes. Longer lines are truncated at this length.

In addition to the value it returns in register 15, the output exit routine must always return a second value in register 0. This second value is for the convenience of assembler callers, which otherwise need a translate table or loop to determine the length of a new or modified line. C callers can use **`strlen`**. This value must be the length of the output line (excluding the delimiter) or 0 when no line is to be written. If the original line is to be written, register 0 is set to the third input parameter. If a new or modified line is to be written, register 0 is set to its length.

Note that the output exit routine is not called for terminal output that is caused by the operation of a CMS EXEC or TSO CLIST that is invoked via the debugger EXEC interface (for instance, REXX SAY statements).

Normal Call: Input Exit Routines

When the input exit routine receives control for a normal call, register 1 points to the following parameter block:

0	a fullword with the value 0 (normal call)

4	the value from the program's CRAB user word 1 (CRABUSR1)
8	the value from the program's CRAB user word 2 (CRABUSR2)
12	the value from the program's CRAB user word 3 (CRABUSR3)
16	the value from the program's CRAB user word 4 (CRABTUSR).

Whenever it is called for a normal call, the input exit routine should either provide a command line or tell the debugger to read a command line from its standard input file (usually your terminal).

To tell the debugger to read a command line from its standard input file, the input exit routine returns values of 0 in both register 15 and register 0. If the input exit routine provides a command, it must set register 15 to the address of the command. Any nonaddress bits that are in register 15 must be set to 0; the instruction LA 15,0(,15) accomplishes this. The command must end with a C new-line character (hex 15), followed by the standard C string delimiter (hex 00). If the command includes characters that a user cannot type from the terminal (except for the new-line character and delimiter described above), the results are unpredictable. Register 0 must be set to the length of the command, including the new-line character but excluding the string delimiter.

Any debugger commands can be provided. They are treated as if they were read from the standard debugger input file.

Note that the input exit routine is not called for terminal input caused by the operation of a CMS EXEC or TSO CLIST invoked via the debugger EXEC interface (for instance, CLIST READ statements).

Termination Call: Output Exit Routines

When the output exit receives control for termination, register 1 points to the following parameter block:

0	a fullword with the value 8 (= termination)
4	0
8	0
12	the value from the program's CRAB user word 1 (CRABUSR1)
16	the value from the program's CRAB user word 2 (CRABUSR2)
20	the value from the program's CRAB user word 3 (CRABUSR3)
24	the value from the program's CRAB user word 4 (CRABTUSR).

The output exit routine should perform any necessary cleanup at this time and set a return code of 0 to indicate successful termination. If problems are encountered, recall that an abend can terminate the debugger and the user program. A message, followed by return code 0, may be preferable.

The input and output exit routines can be terminated in either order. Termination can occur because of termination of the user program or termination of the debugger. If the debugger or the program that is being debugged terminates abnormally, no termination calls are made to the exit routines. However, remember that a program check in the user program is usually trapped by the debugger and does not cause termination of your program.

Termination Call: Input Exit Routines

When the input exit routine receives control for termination, register 1 points to the following parameter block:

0	a fullword with the value 8 (= termination)
4	the value from the program's CRAB user word 1 (CRABUSR1)
8	the value from the program's CRAB user word 2 (CRABUSR2)
12	the value from the program's CRAB user word 3 (CRABUSR3)
16	the value from the program's CRAB user word 4 (CRABTUSR).

The input exit routine should do any necessary cleanup at this time and set a return code of 0 to indicate successful termination. If problems are encountered, recall that an abend can terminate the debugger and the user program. A message, followed by return code 0, may be preferable.

The input and output exit routines can be terminated in either order. Termination can occur because of termination of the user program or termination of the debugger.

APPENDIX 4

Installing and Administering the NFS Client

Introduction

In a cross-development environment, the Network File System (NFS) client support that is provided by the SAS/C Connectivity Support Library (CSL) enables the SAS/C Debugger to communicate with the host workstation. This appendix provides the basic information that is necessary to administer this NFS support. Additional information is contained in Appendix 5, "Using the NFS Client," on page 321.

As an administrator for the SAS/C CSL NFS client, you install the software, establish access controls for remote file security, and diagnose problems. You may also develop file-system mount configurations. This support is provided in a distributed file systems environment that uses the Sun NFS protocol for network communication between computer systems.

Distributed File Systems

As networking protocols and applications have become more sophisticated, file sharing among computers has evolved from simple file transfer to the construction of distributed file systems. In a distributed file system, programs and users can access (open, read, write, and so forth) file systems from a remote machine directly, as if they were attached to the local system.

Although numerous designs for distributed file systems have been implemented experimentally, only a few have achieved commercial success. Of these, the Sun Microsystems Network File System (NFS) protocol is by far the most widely used. Although not as full-featured as some other file systems (most notably the Andrew File System) in areas such as file caching and integrated security administration, its design has made it easy to implement on a wide variety of systems. NFS software is currently available for almost every computer and operating environment.

NFS Design

NFS is implemented by using a protocol that is composed of Sun Remote Procedure Call (RPC) function calls. As with most RPC applications, the protocol supports a dialog among servers and clients. The NFS servers are the machines that provide remote access to their file systems. NFS clients are programs that access the files on another system. The use of RPC applications enhance interoperability among diverse machines.

The NFS protocol views all file systems as conforming to the hierarchical directory organization that has been popularized by the UNIX operating environments. That file system was subsequently codified by the IEEE POSIX standard. The NFS protocol not only allows reading and writing of files, but it also supports manipulation of directories.

Each NFS client system builds and maintains its own file system view. This view results from a hierarchical combination of its own file systems and the file systems of servers that it accesses. At any given directory of this view, the client system may attach a new subtree of directories from an NFS server. This process of attaching a new subtree of directories is called *mounting a remote file system*. The directory to which the remote file system is mounted is called the *mount point*.

An important effect of mounting a remote file system is that the files in the mount-point directory are no longer visible to the client. The newly mounted files in the remote file system are visible instead.

Another important principle is that NFS mounts that are made by a server, when that server acts as a client to another system, are not visible to its clients. The clients see only the files that are physically located on the server.

For users of OS/390 and CMS, perhaps the most important aspect of the NFS design is its orientation as a network service instead of as the file system component of a distributed operating environment. This orientation is critical in enabling the use of NFS on operating environments that are dissimilar to the UNIX environments in which NFS was originally implemented. The primary requirements for an operating environment to participate in NFS are the ability to interpret a hierarchical file system structure and the ability to share UNIX format user identification numbers. Other similarities to UNIX are not required. The SAS/C CSL NFS implementation is able to effect support for directories and UNIX user identification on OS/390 and CMS.

SAS/C NFS Client Overview

When you are working in a distributed environment without file sharing, the barrier between systems can become problematic. Files that are needed on one system often reside on another. The solution of transferring the entire file, using File Transfer Protocol (FTP) for example, is practical if the file is small and seldom changes, but becomes much more laborious when this is not the case.

Traditionally, programs running under OS/390 and CMS have had little or no access to files that are located on PCs, workstations, and other nonmainframe computers. The SAS/C CSL NFS client support changes this situation. For example, in the cross-development environment you can run the SAS/C Debugger on the mainframe while your source and debugger files reside on a workstation.

Accessing Files

The SAS/C CSL NFS client transient libraries enable a new filename style prefix, **`path:`**, in SAS/C filenames. In the same way that an OS/390 program can use the **`dsn:`** prefix to open a file by data set name, the program can now open an NFS file with the **`path:`** prefix. Thus, files that are accessed by using NFS are placed in a separate name space from traditional OS/390 or CMS files. This separation is due to differences in file system organizations, such as directories versus partitioned data sets, rather than the fact that one group is local and the other is remote.

Mounting Directories

SAS/C CSL functions and configuration files are available to mount directories in the mainframe environment. As multiple mounts are established from one or more remote machines, the CSL NFS client library maintains a unified hierarchical view of the resultant directory structure. With the CSL NFS client, mounts are the responsibility of the individual user, not of a system administrator.

For example, a configuration file with the following line can be used if the user wants to access a UNIX root directory / on a machine that is named **`acct.langdev.abc.com`**.

```
acct.langdev.abc.com:/ / nfs
```

This indicates that the root of the **`acct.langdev.abc.com`** machine should be mounted as the root directory on the mainframe, thus enabling a debugger user to specify **`set search`** commands relative to the mount point. See “Using the Debugger’s set Command” on page 93 for information about the SAS/C Debugger’s **`set search`** command.

To continue the example, suppose the user now invokes the debugger on the mainframe and issues the following **`set search`** command:

```
set search userinclude =
   "path:/usr/name/project/headers/%leafname"
```

The debugger now looks for user include files in the **`/usr/name/project/headers`** directory on the remote workstation that is named **`acct.langdev.abc.com`**.

In a more complicated setup, many different UNIX workstation file systems can be mounted together. The overall organization is the responsibility of the mainframe user, and the pathname for a particular file will often differ from what would be used on any of the systems individually.

File Security

The CSL NFS client enforces security controls that prevent unauthorized access to files on the server. Before the user can access an NFS file, the user identification must be authorized by the local Resource Access Control Facility (RACF) compatible security system, if one is available, and by a login server that is running on a UNIX system. If a local security system is available, this login process can be invoked automatically by the CSL library. If not, the user must supply a UNIX (or other NFS server operating environment) username and password.

In either case, the NFS client software maintains the standard UNIX or POSIX User Identification (UID) and Group Identification (GID) numbers for the duration of the user's session. The NFS client software controls access to remote files that are based on the user identification and the file's permissions.

Installation Considerations

The NFS client software depends on the SAS/C transient library, the SAS/C CSL transient library, and the TCP/IP software that are provided by your TCP/IP vendor. These must all be installed properly for the NFS client software to function correctly. See SAS/C Library Reference, Volume 2 for additional information.

The NFS client commands must be accessible to users. Under CMS, this involves accessing the disk. Under OS/390, the commands can be found if the commands are placed in linklist or LPALIB, or if they are in a data set that is allocated to the DDname CPLIB (provided that the optional SAS/C TSO command support is installed). Alternatively, OS/390 sites with REXX support can use REXX EXECs that invoke the commands. This avoids any need to install the SAS/C TSO command support.

In addition to mainframe installation considerations, you must coordinate NFS usage with the administrators of the NFS servers. They must grant the mainframe access in their configuration files. Additionally, they must install a login server for mainframe users to contact.

SAS/C CSL comes with distribution kits (in UNIX **`tar`** format) for two login servers. The first is the standard PCNFSD version 2 server from Sun Microsystems. The second is the CSL's **`sascuidd`** server, which is used for login without a password. If the NFS network is already running a PCNFSD version 1 server, it can be used instead of the PCNFSD version 2 server. The distribution kits include README and Makefile files to explain the process of building the programs under your login server operating environment.

PCNFSD may be difficult to port to some systems, particularly systems that are not UNIX systems. There are a number of alternative approaches to solve this problem. If there is a secure UNIX system available in the network that is already running PCNFSD, then that system can be used. If no such system is available, sites with mainframe security systems can rely exclusively on **`sascuidd`** (which is much easier to port). **`sascuidd`** runs on any POSIX system that also supports RPC applications. It is also possible to use an abbreviated version of PCNFSD. Only the authorization and null procedures are needed for CSL NFS. The other proceedures (mostly related to printing) are not needed.

Whatever server is installed and used, it must be running whenever mainframe users might need access to NFS files.

NFS Security Administration

The installation of NFS client software on any system should be a security concern for administrators of NFS servers. The availability of client software might enable file access by users for whom the access was not previously possible.

All security on NFS servers is administered via UNIX (or POSIX) UIDs and GIDs. The UID is a number that represents a user. The GID represents a group of users. The NFS design ensures that all participating machines share the same UID and GID assignments. OS/390 users are identified by a security system such as RACF or ACF2. CMS users are identified by entries in a CP directory. UNIX UIDs and GIDs are not normally associated with mainframe users.

Administrators of NFS servers can usually control, on a file system basis, which client machines can access files via NFS. When security is a concern, the ability of the client machine to allow only authorized UID and GID associations is the most important factor. The CSL NFS client software derives its UID and GID associations from a combination of mainframe security system and UNIX servers. The exact source authorization depends on site configuration.

Because of differences between UNIX and mainframe operating environments, and because of a lack of reserved port controls in current mainframe TCP/IP implementations, the CSL NFS client software is generally less secure (in authorizing UID and GID associations) than most UNIX NFS client implementations. Methods of access are briefly described in the SAS/C CSL Installation Instructions. Note that it is server file security that is of concern. An NFS client implementation can pose no additional security threat to files on the client (in this case the mainframe) unless it gives unauthorized access to files that contain passwords. Note also that most UNIX NFS servers allow controls for which file systems can be accessed, thus limiting exposure to unauthorized UID associations.

Because it can use authorizations that are provided by a mainframe security system, CSL NFS client software is generally more secure than NFS client implementations on PC operating environments.

UID/GID Acquisition

The SAS/C CSL NFS client software will always retrieve the UNIX UID and GID information from a UNIX server. The retrieval of the UID information is based on a UNIX username. The association of UNIX username to UID/GID is always performed by a UNIX server.

One of two methods is used to associate a UNIX username with a mainframe user. If there is a (RACF-compatible) security system installed, profiles can be established to associate mainframe users with UNIX usernames. For a user whose mainframe login ID is the same as the UNIX username, a single profile can be used. There is considerable flexibility in this arrangement. For example, the association between mainframe userids and UNIX usernames need not be one-to-one. When this method is used, the **`sascuidd`** login server is used to provide UID information for that username. No UNIX password is required.

A second method is for the mainframe user to supply the desired username and password to the UNIX login server PCNFSD (version 2 if available, otherwise version 1), which authorizes the username (based on the password) and supplies UID information in one step.

The first method is preferred because it makes login easier and removes the requirement that UNIX passwords be present on the mainframe.

For TSO or CMS users, the UID information is stored in environment variables. The UID is stored in **`NFS_UID`**. The primary GID is stored in **`NFS_GID`**. The list of supplementary GIDs (**`sascuidd`** and PCNFSD V2 only) is stored in **`NFS_GIDLIST`**. **`NFS_LOGINDATE`** is set to the date of the login.

The environment variable NFS_LOGINKEY receives an encrypted value that is used by subsequent NFS calls to determine that these environment variables have not been tampered with.

NFS logins must be reissued each time the user logs in to TSO or CMS. Authorization is also lost after about 48 hours, even if the user does not log off. This prevents users from retaining their authorization indefinitely, even after they have had their UNIX authorizations removed.

At most 16 additional GIDs are allowed. This is the maximum supported by the PCNFSD protocol. A user who can login as UID 0 (root) will probably not be given full authority by the NFS server system. Most NFS servers remap UID 0 to a UID value of **`(unsigned short) -2`**.

SAS/C NFS client capabilities cannot be used until a successful login has occurred. Successive calls to NFSLOGIN can be made in order to access a different server or to use a different login ID. If a security system is present to allow login without a password, the actual login may be performed automatically when an NFS operation is requested. No corresponding logout is required.

When a mainframe security system is present, it can also control which login server a user is allowed to access. This prevents users from rerouting their login authorization request to a less trusted machine. It also reduces the risk of a user sending a UNIX password to a Trojan horse program that is running on an unauthorized system.

Use of a mainframe security system requires the definition of a generalized resource named LSNUID. The mainframe security administrator can then supply profiles that give mainframe users access to particular login servers and equate mainframe userids with UNIX usernames. The next section describes this in detail.

RACF Definitions for NFS Clients

The SAS/C CSL NFS Client mainframe security system interface is based on profiles that are defined for a generalized resource named LSNUID. Using this resource, you grant specific mainframe users access to specific UNIX userids and login servers in the same way that DATASET profiles enable you to grant users access to mainframe files.

Until this resource is defined and activated, the NFS client code behaves as it does when no security system is installed.

Here is a description of the macro parameters that are needed to define LSNUID (in a RACF environment).

```
ICHERDCE    CLASS=LSNUID
            ID=nn
            MAXLNTH=39
            FIRST=ANY
            OTHER=ANY
            POSIT= (prevented when RACF unavailable,
                    auditing if you want it,
                    statistics if you want them,
                    generic profile checking on,
                    generic command processing on
                    global access checking off)
ICHRFRTB    CLASS=LSNUID
            ACTION=RACF
```

In RACF, once this resource is defined, it must also be activated via the command

```
SETROPTS CLASSACT(LSNUID)
```

The NFS client libraries make authorization inquiries about the following profile names (all requests are for read permission):

LOCAL_ *userid*
: Users who are permitted to this profile are authorized to use their mainframe userid (lowercased) as the UNIX username without specifying a UNIX password.

USER_ *name*
: Users who are permitted to this profile are authorized to use the string name (lowercased) as their UNIX username without specifying a UNIX password. For example, if a mainframe user is permitted to the profile **USER_BILL**, then he is allowed to assume the UNIX username of **bill**.

P*ddd.ddd.ddd.ddd*
This specifies the network address (dotted decimal) of a PCNFSD server that the user can access to obtain a UID and GID. Requiring permission for access to servers prevents users from setting up unauthorized versions of PCNFSD on a less trusted machine and then directing their login queries to it. For example, if mainframe user BILL is permitted to P149.133.175.68, he can use the server at that IP address when logging in. Leading zeros are not allowed in these names. That is, the previous profile could not have been for P149.133.175.068.

S*ddd.ddd.ddd.ddd*
This is similar to the above, but permits access to a **`sascuidd`** server.

Configuring a Default Login Server

In most cases, it is better for users to reach a default login server. Having a correct default reduces user effort and confusion. But most important, the correct default must be set if the NFS client library is to perform logins automatically.

You can control the login server in three ways. One way is to set the NFSLOGIN_SERVER environment variable in the user's PROFILE EXEC or TSO startup CLIST. Another way is to apply the default login server configuration zap that is supplied in the installation instructions. The best method is to accept the default name **`nfsloginhost`** and to configure your nameserver or **`/etc/hosts`** format file accordingly.

Developing Standardized File-System Configurations

You may want to set up the file system configuration for users. If so, you can create a system-wide **`fstab`** file to perform their mounts. The search rules for the **`fstab`** file include a provision for a system-wide name. Users who do not set up **`fstab`** files of their own use the system-wide file. If you want users to save file system context between programs, you can define the ETC_MNTTAB environment variable in the PROFILE EXEC or TSO startup CLIST.

Diagnosing Problems

The first step in identifying problems is to look carefully at the diagnostics that are produced by the debugger at the point where the failure occurred. Depending on whether the messages are generated by the debugger or by the library, the messages may be printed in the log window, or they may be printed in line mode after erasing the debugger screen.

Many user problems are caused by incorrect installation of system software. These problems can often be diagnosed by understanding what is missing. Sometimes a configuration file is missing. Other times an environment variable definition is needed, or a REXX EXEC is not placed where it will be accessed.

In other cases, problems are caused by network and server failures. For server problems and failures on remote systems, the RPCINFO and SHOWMNT commands are useful. Both SHOWMNT and RPCINFO are compatible with the equivalent commands under UNIX.

If you are having problems during the NFS login or the remote mounting process, set the _SASC_NFS_VERBOSE evironment variable to 1. This produces additional diagnostic information during the NFS login process and the remote directory mount process.

Beginning with SAS/C Release 7.00 the SAS/C CSL NFS client library function NFSLOGIN has been changed to allow communication with NFS login servers that use port 2049 as well as one of the reserved ports. Port 2049 has been registered by Sun Microsystems for use by NFS.

The NFSLOGIN function was also enhanced to allow the installation to specify a specific port to be used for communication with the NFS login server. After customization, the NFSLOGIN function will communicate with the NFS login server that uses the specified port. No other port will be accepted.

Note: See the installation instructions for information on customizing the desired port. △

To provide more diagnostic information, the NFS login process and the mount remote directory process have been enhanced to display informative messages about the activity of each process. To enable the display of these messages, define the environment variable _SASC_NFS_VERBOSE. When _SASC_NFS_VERBOSE is defined as 1, the informative message will be produced.

Output similar to the following will be produced by the NFSLOGIN function when the NFSLOGIN command is executed:

```
nfslogin solgnu
nfslogin -s nfsloginhost.mysite -u myuserid -p mypassword
NFSLOGIN - NFSLOGINHOST IP ADDRESS: 10.23.149.16
NFSLOGIN - RACF Resource          : P10.23.149.16
NFSLOGIN - RACF access to Resource: allowed
NFSLOGIN - NFSLOGIN Server        : PCNFSD
NFSLOGIN - Username               : myuserid
NFSLOGIN - PCNFSD Program Number  : 150001
NFSLOGIN - Port returned          : 924
NFSLOGIN - Port checking          : reserved or 2049
NFSLOGIN - PCNFSD Version         : 2
NFSLOGIN - NFS_LOGINKEY : f75cf6f3f66ff4f45cf1f0.........
NFSLOGIN - NFS_LOGINDATE: 12/3/1999
NFSLOGIN - NFS_UID      : 2447
NFSLOGIN - NFS_GID      : 105
NFSLOGIN - NFS_GID_LIST :
NFSLOGIN -   GID_LIST item 0 value: 44
Login succeeded.
```

Output similar to the following is produced by the **ls** command during the mount process:

```
ls
MOUNT    - mount - ftstab used : //DDN:ETCFSTAB
MOUNT    - mount - device      :
MOUNT    -   >>>>krups.unx:/vol/vol0/u/userid
MOUNT    - mount - mountpoint  :/
MOUNT    -        Host(krups.unx), MOUNTPROG(100005), MOUNTVERS(1)
MOUNT    - mount: successful
MOUNT    - mount - device      :
MOUNT    -   >>>>krups.unx:/vol/vol0/u/sasctg/playpen_700_mvs
MOUNT    - mount - mountpoint  :/unix_mount_point1
MOUNT    -        Host(krups.unx), MOUNTPROG(100005), MOUNTVERS(1)
MOUNT    - mount: successful
MOUNT    - mount - device      :
MOUNT    -   >>>>d5412.us:c:/C++_Samples
MOUNT    - mount - mountpoint  :/pc_mount_point1
MOUNT    -       Host(d5412.us), MOUNTPROG(100005), MOUNTVERS(1)
MOUNT    - mount: successful
MOUNT    - mount - device      :
MOUNT    -   >>>>krups.unx:/vol/vol0/u/sasctg/
MOUNT    - mount - mountpoint  :/unix_mount_point2
MOUNT    -       Host(krups.unx), MOUNTPROG(100005), MOUNTVERS(1)
MOUNT    - mount: successful
```

Typically, NFS login servers use a reserved port (<=1023) or port 2049 for communication with a client. If your NFS login server uses some other port you may specify the correct port by using the zap provided in Usage Note 1900. The RPCINFO command can be used to determine which port is being used by the NFS login server.

For true network problems, SNMP or other network diagnostic facilities are most useful.

Recommended Reading

Many of the concepts and topics discussed in the following book may also help you administer mainframe NFS client software: Stern, H. (1991), Managing NFS and NIS, Sebastopol, CA: O'Reilly & Associates, Inc.

NFS Administrator Commands

In addition to the commands described in Appendix 5, "Using the NFS Client," on page 321, as an NFS administrator you should be familiar with the SHOWMNT and RPCINFO commands as they are described in the following section.

SHOWMNT

queries an NFS server for file system information

SYNOPSIS

```
SHOWMNT [-e] [-d] [-a] [host]
```

DESCRIPTION

The SHOWMNT command queries an NFS server for information about file systems that may be mounted by NFS.

host is the hostname of the NFS server. If you omit this parameter, SHOWMNT returns information about the NFS server on the local machine (if one is installed).

SHOWMNT handles two basic types of lists. The first is an exports list. The exports list tells you which file systems can be mounted. The second is a list describing which mounts have actually taken place. The form of the second list depends on the **-d** and **-a** options. The **-e** option requests the exports list. This includes information about which hosts are authorized to mount the listed file systems. This information may either be everyone, or a list of group names that represent a set of hosts. If it is authorized, a host may mount any of the listed file systems.

You can use the following command when you are trying to determine the name of a file system to mount:

```
SHOWMNT -e
```

Note: You can often mount subdirectories of the listed file systems. Whether you can do this depends on whether the subdirectory is in the same physical file system on the server. Contact the server administrator or examine server configuration files to determine this. △

If the **-e** option is used in conjunction with other options, this exports list will be printed first, followed by the list describing actual mounts.

If you don't specify any options, SHOWMNT prints the list of actual mounts, showing only the names of the hosts that have a mount. The list is sorted by host name.

If you specify the **-d** option, SHOWMNT prints the list of actual mounts, showing only the names of directories that have been mounted. The list is sorted by directory name.

The **-a** option gives the most verbose format for the list of actual mounts. It indicates that the list should be printed as host:directory pairs. If you do not use the **-d** option, SHOWMNT sorts the list by host. If you do use the **-d** option, SHOWMNT sorts the list by directory.

INVOCATION SYNTAX

The syntax is generally identical to that shown above. Under OS/390, system administration considerations may require use of the TSO CALL command or other techniques.

EXAMPLES

```
showmnt -e byrd.unx
```

Show mountable file systems on the byrd.unx NFS server.

```
showmnt byrd.unx
```

Show the list of other hosts that have mounted the NFS file system from byrd.unx.

RPCINFO

queries the Portmapper

SYNOPSIS

- □ Format 1: RPCINFO -p [**host**]
- □ Format 2: RPCINFO [-n **port**] -u **host program** [**version**]
- □ Format 3: RPCINFO [-n **port**] -t **host program** [**version**]
- □ Format 4: RPCINFO -b **program version**

DESCRIPTION

The RPCINFO command queries the portmapper on the designated host to determine the status of programs that are available as RPC servers. It can list all programs that are known to the portmapper, and it can test the program itself (as opposed to asking only the portmapper) for availability. To test for program availability, RPCINFO calls the null proceedure (procedure 0) of the program and waits for a response.

The **-p** option of RPCINFO is used to query the portmapper on the specific host for a list of all registered programs.

The **-u** and **-t** options test the programs themselves for availability. Use **-u** for UDP based services and **-t** for TCP based services. You can also use **-n** to specify a TCP or UDP port in the rare case that you want to query a program at a port different from the one known to the portmapper.

The **-b** option checks all hosts that can be reached with a broadcast from your local machine. This variant is often of limited utility because the set of hosts queried is dependent on the physical configuration of your network.

INVOCATION SYNTAX

The syntax is generally identical to that shown in the SYNOPSIS. On OS/390, system administration considerations may require use of the TSO CALL command or other techniques.

EXAMPLES

rpcinfo -p byrd.unx
Show all RPC servers that are registered on the host named byrd.unx.

rpcinfo -u byrd.unx nfs
Check the status of the NFS server on the host byrd.unx.

rpcinfo -u byrd.unx 100005
Check the status of the NFS mount server (program number 100005) on byrd.unx.

APPENDIX

5

Using the NFS Client

Introduction

In a cross-development environment, the Network File System (NFS) client support provided by the SAS/C Connectivity Support Library enables the SAS/C Debugger to communicate with the host workstation. This appendix provides the basic information that is necessary to use this NFS support. Additional information is contained in Appendix 4, “Installing and Administering the NFS Client,” on page 309.

The NFS client feature provides flexibility in configuring NFS for each user. The degree of effort that is required to set up your configuration depends on the amount of support that is given by the system administration staff at your site.

For example, minimal user effort is required when the system administrators provide a centralized mount-configuration file and when they set up security-system definitions to enable automatic login. In this situation, users can begin specifying NFS filenames to application programs immediately. On the other hand, some sites may leave mounting files to the individual user. Lack of a Resource Access Control Facility (RACF) compatible security system might require that users issue an NFSLOGIN command at the beginning of each session. Even at sites where a centralized configuration has been set up, individual users with specialized access requirements may still develop their own configurations.

Logging on to the NFS Network

NFS servers use a UNIX, or POSIX, file-permission system. This system gives each user a user identification number (UID), a group identification number (GID), and possibly several additional supplementary GIDs. Each file is assigned ownership by

UID and by GID. Permissions for the file are set based on whether the user who wants access is the owner (has the same UID as the file), is in the file's group (has a GID that matches the GID of the file), or is some other user. For each of these three categories (owner, group, and other) read, write, and execute permissions can be assigned.

To access files that use NFS, your session on OS/390 or CMS must acquire UID and GID numbers that correspond to some user on the NFS server network. You acquire these numbers by contacting a login server on the NFS network to ask permission to access files according to a username that is known to that server. In many cases, contact with the NFS login server can be automatic the first time that you access an NFS file. In other cases, you must issue the NFSLOGIN command to effect the login.

The function of the login server is to check your identification and grant you access to the network. Once you are logged on, the login server functions as an NFS server and provides access to the files that are located on the machine on which it resides. At this point you may also use the network to access files that are controlled by other NFS servers on other machines.

If you have a RACF-compatible security system running on your mainframe and your site administration has given you access to your NFS login server username, then the security system suffices and no password is required. Note that the login server username is not necessarily the same as your OS/390 or CMS userid. If you do not have a security system, then you will need to type your password during the login process.

In summary, the login process can involve three pieces of information:

- host name of the login server. For example, the host name of a workstation running UNIX that acts as an NFS server.
- login server username. For example, your username on UNIX.
- login server password for that username.

The requirement for a password depends on whether a mainframe security system can provide authentication for login server usernames. If the NFS client software can determine the other two pieces of information, either by default or by environment variables, then automatic login is possible. Otherwise, the NFSLOGIN command must be used.

For example, if your NFS network is composed of UNIX machines, your UNIX username is **`comkzz`**, and your login server is a UNIX machine called **`byrd.unx`**, then the CSL NFS client software must contact **`byrd.unx`** and provide **`comkzz`** as the user name. If your OS/390 username is also COMKZZ (the same except that it is uppercase), the mainframe security administrator has authorized you to use the **`comkzz`** username for NFS, and if **`byrd.unx`** has been configured as the default login server at your site, then the NFS client library will log you in automatically the first time you try to use NFS.

If, on the other hand, your site does not have RACF, a password is required. In this case, you need to issue the NFSLOGIN command to type your password. See "NFSLOGIN" on page 327 for details.

After the login processing has succeeded, your session receives a UID and one or more GIDs. These control your subsequent accesses to NFS files.

Accessing Remote File Systems

Logging on establishes UID and GID information. The next step is to mount the remote file systems that you want to access.

Because the SAS/C CSL NFS client feature runs totally within your user address space under OS/390, or on a virtual machine under CMS, you must mount remote file systems before accessing NFS files. A number of facilities are provided to make this process as transparent as possible. Mounts can occur in three ways:

- □ The configuration file, **`fstab`**, specifies a mount that occurs at session or program startup.
- □ You issue the MOUNT command.
- □ An application program performs a mount as part of its own processing logic.

At sites with standardized configurations, a series of mounts may be provided automatically. In this case, you do not need to do additional work unless you want a different configuration.

Saving File-System Context

Assuming that you are doing the configuration yourself, one of the first things to decide is the duration of your mounts. That is, do you want mounts and directory changes from one program to be preserved for the next program that is run? Mounts and directory changes form a file system context that may be restricted to the execution of a particular program or may be shared serially by programs under TSO or CMS.

The serial sharing of file system context is accomplished by using the **`mnttab`** file. Not sharing context can be easier. When only a few file systems are mounted, reissuing the mounts in each program can be faster than reading and writing the **`mnttab`** file. NFS mounts are very fast and involve minimal processing on the server.

Unfortunately, processing the **`mnttab`** file at program startup and shutdown adds noticeable delays to otherwise fast commands and programs. The NFS sample programs **`cd`**, **`pwd`**, and **`ls`** illustrate this. Overall NFS performance is much better when a single program does many operations. Sharing is required, however, if working directory changes are to be preserved from one program to the next. You should always save the file-system context when you are working with the SAS/C Debugger in a cross-development environment.

You specify serial sharing of file system context by setting the ETC_MNTTAB environment variable to the name of a file to contain the context. For example, under TSO, you might use the value TSO:ETC.MNTTAB. This creates a file tsoprefix.ETC.MNTTAB. Under TSO you set the value by using the PUTENV command. Allocating a DDname of ETCMNTTB has the same effect under OS/390 batch and may be more convenient. Under CMS, you can set the value by using GLOBALV commands with the CENV group. See SAS/C Compiler and Library User's Guide and SAS/C Library Reference, Volume 1 for more information about using environment variables with the SAS/C Compiler.

You do not need to create the **`mnttab`** file yourself. The NFS client library will create it automatically. It will also be deleted each time you log on to the NFS server. Note that, unlike the conceptually similar UNIX **`/etc/mnttab`** file, this file has a binary format. It also contains information, notably the current working directory, that is held by the kernel in UNIX.

Finally, the **`mnttab`** file cannot be shared simultaneously by many programs. If you are managing multiple programs that use NFS concurrently, either set up multiple **`mnttab`** files or set them up not to save context at all.

To avoid serial sharing, do not set the environment variable. In this case, the MOUNT command and the **`sample cd`** command appear to have no effect, because the changes that they request are not saved when they end. When not sharing file system context, you invoke all your mounts with the **`fstab`** configuration file.

Setting Up an fstab Configuration File

When NFS starts with no **`mnttab`** file available, either because there is no serial sharing of file system context or because NFS has not yet been used, the NFS client library searches for an **fstab** configuration file that specifies which initial mounts to perform. The **fstab** file removes the need to issue mount commands manually each time NFS is used.

The **fstab** configuration file format is identical to that used in most UNIX systems. It should have a series of lines that specify mount points that use the following format:

```
server : directory mount-point type options
```

Fields are separated by white space, and any fields that follow the options parameter are ignored. You can also include comments in the **fstab** configuration file. The pound (#) character that appears at the beginning of a line or that is preceded by white space indicates that the rest of the line is a comment.

For NFS file systems, the device is specified as a server name that is followed by a colon (:), which is followed by the name of the directory to mount. This name must be a physical file on the server.

Note: It must not be a name that was created by NFS client features of the server. This is a common source of confusion. Users of the NFS server are often accustomed to specifying directory names that are not physical directories on their system. As discussed earlier, the design of NFS does not cause these names to be propagated automatically to NFS clients of that server. △

The mount-point parameter must be a pathname in the directory hierarchy that is being created on the mainframe. In order for the first directory to be mounted, the mount point must be a slash (/), which indicates the root directory. Following NFS conventions, later mount points must be actual directories in a file system that have already been mounted. The directories that are being mounted then obscure the contents of the directory that they are mounted on.

The type parameter must be **nfs**. As in UNIX, the table definition is generalized to accommodate multiple types of file systems; however, at present only NFS file systems are supported.

Mount options, which are described in "Mount Options" on page 325, generally are not needed.

Output A5.1 on page 324 shows a typical **fstab** configuration file:

Output A5.1 Example fstab configuration file

```
# My NFS setup
byrd.unx:/local/u/bill  /       nfs     #No mount options
server.unx:/tools       /tools  nfs ro  # Mount tools read-only
elgar.langdev:c:/       /lang   nfs     # Mount from OS/2
```

This example assumes that the **/local/u/bill** directory on **byrd.unx** contains subdirectories that are called **tools** and **lang**. Presumably these are empty directories that were set up to serve as mount points for the second and third mounts. If they are not empty, any contents that they have are obscured to the mainframe user by the second and third mounts. Instead of seeing the contents of the local directories, the corresponding directory trees from the **/tools** directory on server.unx and the **c:/** directory on **elgar.langdev** are seen by the mainframe user at those locations.

The **fstab** data set is located in the following manner:

1 If there is an environment variable that is named **ETC_FSTAB**, its value is used. Note that the default style is **ddn:**. Remember to include the style at the beginning of the name if you want a different one, such as in **tso:etc.fstab**.
2 Under OS/390, if there is a DDname of ETCFSTAB, it will be used.
3 The next data set in the sequence depends on the operating environment that you are working under.

- □ Under TSO, *tsoprefix*.ETC.FSTAB is used.
- □ Under OS/390 (other than TSO), if the userid can be determined, *userid*.ETC.FSTAB is used.
- □ Under CMS, ETC FSTAB is used.

4 If you are working under OS/390, *zappedprefix*.ETC.FSTAB is used. The *zappedprefix* defaults to NFS if it is not zapped, and it can be overridden by the **`NFS_PREFIX`** environment variable.

The **`fstab`** data set cannot itself be accessed with the **`path:`** prefix. See "Accessing Files" on page 311 for information about the **`path:`** prefix.

Mount Options

Mount options control the operation of mounting the file system, as well as the file system's characteristics for subsequent use. The options must be separated by commas, with no intervening spaces. They can be specified in either uppercase or lowercase. Mount options are not usually needed; the defaults are generally adequate.

Table A5.1 on page 325 contains the options that you can specify.

Table A5.1 Mount Options

Option	Description
RW	Indicates that the file system is read/write. This is the default setting.
RO	Indicates that the file system is read-only.
DELTAMIN	Indicates the time adjustment in minutes to be applied to time stamps on the given file system. This can be useful when file systems are set to operate in different time zones. This value can be either positive or negative.
RETRY=n	Number of retries for mount failures. The default is 1. The parameter affects only mount attempts. It does not affect other operations such as read and write. (See RETRANS for other operations.)
RWSIZE=nnK	Reads and writes buffer size. The default is 4K. The maximum allowed is 1024K.
TIMEO=n	Controls the timeout interval in tenths of a second used between retransmission attempts. The actual timeout interval begins at n tenths of a second and is doubled for each retransmission. The default TIMEO value is 7. (See also RETRANS.)
RETRANS=n	Specifies the number of NFS retransmissions. The default is 4. The timeout is multiplied by 2 for each successive retransmission.
SOFT	Specifies that a transmission attempt should be abandoned after a complete set of retransmissions fails. This is the default.
HARD	Specifies that a transmission attempt should not be abandoned after a complete set of retransmissions fails. If HARD is specified, the retransmission process is started over again after each set of transmissions is completed.
TEXT	Performs ASCII or EBCDIC translation on all files. An ASCII-to-EBCDIC translation is performed when the file is read from the server, and an EBCDIC-to-ASCII translation is performed when the file is written to the server.

Option	Description
BINARY	Always leaves data in untranslated, binary form.
XLATE	Gives the name of a loadable translate table to be used for ASCII and EBCDIC translation in this file system. This translation affects data that are read and written. By default, NFS data are translated using the IBM code page 1047 standard. The table is built in much the same manner as SAS/C CSL RPC translate tables. (See the description of the **`xdr_string`** function in SAS Technical Report C-113, SAS/C Connectivity Support Library, Release 1.00.) The only difference is that you may choose any load module name and then specify it here. If you have created an L$NAEXDR table for RPC, you may specify it to get the same translations for NFS data as for RPC strings. The XLATE option does not affect pathnames, which are controlled by the RPC L$NAEXDR translate table if present. If the translate table is not present, use the code page 1047 standard.

The TEXT and BINARY mount options enable you to override the defaults, which are determined by the debugger when it accesses a file on the workstation. However, we recommend using them only in unusual situations. When using the SAS/C Debugger, the settings defined by the debugger are generally appropriate.

Mounting and Unmounting Manually

When you are saving your file system context between programs, you can manipulate your file system organization by using the MOUNT and UMOUNT commands. These commands are described later in this appendix.

Manipulating Files and Directories

Once you are logged on and have the remote file systems mounted into the directory structure that you want, you can begin to access files. In many cases you can do this through SAS/C programs that are not aware of NFS by specifying **`path:`** where you previously specified a local filename. This will work if the particular program that you are using enabled you to specify the style prefix. For example, CMS programs that enable you to access CMS Shared File System files by using the **`sf:`** prefix enable you to access NFS file by using the **`path:`** prefix. If the program uses the correct setting for text or binary processing when it opens files, text files will be translated from ASCII to EBCDIC automatically. If it does not, you can use the TEXT and BINARY mount options to override the program's decision.

Existing SAS/C programs can also remove, rename, and check accessibility of NFS files.

If you are not saving file system context (or if you are but have not run a program to change the initial directory), you must use the full pathname (from the mainframe point of view) in order to access a file.

Programs that were developed by using SAS/C CSL can access and manipulate the remote file systems more completely. They can create, delete, and list directories. They can work with hard and symbolic links. They can change or check the current working directory, and they can retrieve and change UNIX or POSIX file-status information.

The SAS/C CSL product contains many sample programs that can also be used as simple utilities. For example, the **`ls`** command lists the files in a directory. The **`ncp`** command can copy files between mainframe file systems and NFS file systems (and can be much quicker than using FTP). These sample programs do not have the full features of their UNIX equivalents, but they are useful.

The following examples are distributed with the CSL run-time transients that are provided with the SAS/C Cross-Platform Compiler:

Table A5.2 Sample Programs

Example	Description
`cd`	Changes the directory (requires an ETC_MNTTAB setting)
`ls`	Lists a directory (no wildcards)
`ncp`	Copies files between mainframe and NFS file systems
`pwd`	Prints the working directory

NFS User Commands

The following commands are used primarily by users who are running NFS client applications:

Table A5.3 NFS User Commands

Command	Description
NFSLOGIN	Authorizes TSO or CMS users to access files via NFS
MOUNT	Mounts remote NFS file systems into the NFS client file system structure
UMOUNT	Removes a previously established mount

The format that is used to invoke the NFSLOGIN, MOUNT, and UMOUNT commands is generally identical to that shown in the following reference information. Under OS/390, system administration considerations may require use of the TSO CALL command or other techniques. See your system administrator for details. See “NFSLOGIN” on page 327, “UMOUNT” on page 329, and “MOUNT” on page 328 for information.

NFSLOGIN

Authorizes TSO or CMS users to access files via NFS

SYNOPSIS

Format 1: NFSLOGIN [**`-s`** *server*] [**`-u`** *username*] [**`-p`** *password*] [**`-n`**]

Format 2: NFSLOGIN **`-f`**

DESCRIPTION

The NFSLOGIN command authorizes TSO or CMS users to access files via NFS. In some cases the NFS client software can determine the correct server and username without your specifying them. If a RACF-compatible security system is installed, the site can define particular mainframe users as having access to specified UNIX userids without requiring a password. If no password is required, and if the other values are correct by default, you do not need to use this command. The login will occur automatically when you access the first NFS file or directory.

The NFSLOGIN command is provided for sites and situations where either a password is needed or the default server or username values must be overridden.

See “Logging on to the NFS Network” on page 321 for discussion of NFS login considerations. Also see “NFS Security Administration” on page 312 for more information.

The **-f** option requests a full-screen display. This display has fields for specifying the same information that can be specified on the command line. The full-screen option provides nondisplay password entry.

The server parameter is the host name of the login server that you want to contact. This may differ from the servers on which files are being accessed. The specified host must be running the appropriate login server software. See Appendix 4, “Installing and Administering the NFS Client,” on page 309 for details. You can usually omit this option because the site can set up a default host server at installation time. Note also that, when a security system is installed, the mainframe security administrator controls your access to login servers. Using an unauthorized server causes a RACF violation.

For username, specify your username on the NFS login server. This is often different from your OS/390 or CMS login ID. You do not need to specify a username if the USER environment variable is set to the desired name, or if your login server username is the same as your mainframe userid but converted to lowercase.

If you do not have a RACF-compatible security system, or if you want to login as a username that is not associated with your RACF profile, use the **-p** option or the password field to specify your password on the login server. The mainframe security system (if present) can also control whether a password will be allowed on your NFS login.

Note that the **-p** option requires a value. The **-n** option is required for the special case in which the UNIX (or other login server operating environment) system account has a null password. The **-p** and **-n** options are mutually exclusive. Not specifying either **-p** or **-n** indicates that the user expects the mainframe security system to authorize access to the login server username. The full-screen display also allows for the special case of a null password.

If the login attempt fails, NFSLOGIN prints a message that describes the reason. Otherwise it prints a message that indicates success. The login fails if the login server is not running on the NFS network.

Note that you need not log out from the login server; your UID and GID permissions expire after you log off TSO or CMS. If you want to access files under a different username, you can issue the NFSLOGIN command again. A login expires after two days. See “Diagnosing Problems” on page 315 for more information.

EXAMPLES

```
nfslogin -f
```

Invokes the full-screen login panel.

```
nfslogin -u bbritten -p ocean
```

Logs in to the default login server with username **bbritten** and password **ocean**.

MOUNT

Mounts remote NFS file systems into the NFS client file system structure.

SYNOPSIS

Format 1: MOUNT **`server`** :*directory mount-point* [*options*]

DESCRIPTION

The MOUNT command is one method of mounting remote NFS file systems into the NFS client file system structure on the mainframe. This command is useful only when you have configured your session to save file system context. Otherwise, the MOUNT command has no effect when it completes.

The server parameter specifies the name of the NFS server on which the files are physically located. The directory is the name of the directory for the directory tree that you want to mount. It must be a physical filename on that server (it cannot be created by the server's NFS client software).

The mount-point parameter specifies the name of the mainframe NFS client directory on which the remote file system is to be mounted. For the first mount, this must be a slash (/). For subsequent mounts, it must be a valid pathname in the directory structure that was established by existing mounts.

The options string is not required. It specifies mount options for the file system. See "Mount Options" on page 325. The string of options must be separated by commas, with no intervening spaces.

You cannot mount a file system on a directory that is already being used as a mount point. You must first unmount the existing file system with the UMOUNT command.

Be aware that mounts made by this command are preceded by mounts from any **`fstab`** file.

EXAMPLES

These examples assume that there is no **`fstab`** file and that file system context is being saved.

```
mount byrd.unx:/local/u/bill /
```

Mounts **`bill's`** home directory on **`byrd.unx`** as the root directory on the mainframe.

```
mount server.unx:/tools /tools ro
```

Adds the **`/tools`** directory from **`server.unx`** as a subdirectory and treats it as read-only.

UMOUNT

Removes a previously established mount

SYNOPSIS

Format 1: UMOUNT *mount-point*

DESCRIPTION

The UMOUNT command removes a previously established mount. This command is useful only when you have configured your session to save file system context. Otherwise, the MOUNT command has no effect when it completes.

The mount-point parameter specifies a mainframe pathname to a directory from which a remote file system will be unmounted. The directory must have been used in a previous mount operation.

You cannot unmount the root directory. If you want to mount a different root directory, delete the **mnttab** file and then mount the new root directory. The NFSLOGIN command also deletes the **mnttab** file.

You cannot unmount a file system that has other directories mounted over it, or a file system that contains your current directory. Attempting to do so results in the following message:

```
UMOUNT failed: file or record in use.
```

EXAMPLE

This example assumes that file system context is being saved.

```
umount /tools
```

Removes the file system that was previously mounted at **/tools**. If the file system mounted at / had any files in its **tools** subdirectory, these now become visible.

Index

Your Turn

If you have comments or suggestions about SAS/C® Debugger User's Guide and Reference, Release 7.00, please send them to us on a photocopy of this page, or send us electronic mail.

For comments about this book, please return the photocopy to

SAS Publishing
SAS Campus Drive
Cary, NC 27513
email: yourturn@sas.com

For suggestions about the software, please return the photocopy to

SAS Institute Inc.
Technical Support Division
SAS Campus Drive
Cary, NC 27513
email: suggest@sas.com