Debug for NOS/VE

Usage

60488213

Debug for NOS/VE

Usage

This product is intended for use only as described in this document. Control Data cannot be responsible for the proper functioning of undescribed features and parameters.

Publication Number 60488213

Manual History

Revision	System Version/ PSR Level	Product Version	Date
A	1.2.2/678	1.5	April, 1987
B	1.2.3/688	1.6	September, 1987

This revision:

This is Revision B. It documents the new screen mode features: WIDE and NARROW. Also, minor editorial and technical corrections have been made.

©Copyright 1987 by Control Data Corporation. All rights reserved.
Printed in the United States of America.

Contents

About This Manual 5

- Audience 5
- Organization 5
- Conventions 6
- Submitting Comments 6
- In Case of Trouble 7

Introduction 1-1

- Interactive Debugging 1-3
- Batch Job Debugging 1-6

Getting Started 2-1

- Set Up Your Terminal for Screen Mode Debug 2-2
- Preparing for a Debug Session 2-6

Executing Under Debug Control 3-1

- Beginning a Debug Session 3-1
- Entering Debug Commands 3-6
- Suspending Program Execution 3-7
- Beginning or Resuming Program Execution 3-10
- Displaying and Changing Program Values 3-13
- Getting Help 3-14
- Ending the Debug Session 3-16
- Debug Input and Output Files 3-17
- Automatic Command Execution on Program Failure 3-21
- Recording an Interactive Debug Session 3-26

Screen Mode Debugging 4-1

- Screen Layout 4-2
- Debugging Using Functions 4-4
- Entering Commands on the Home Line 4-5
- The Visible Windows in Screen Mode Debugging 4-6
- Function Descriptions 4-14

Line Mode Debugging 5-1

- Line Mode Command and Function Summary 5-1
- Debug Line Mode Commands 5-3
- Debug Line Mode Functions 5-91

Contents

Comprehensive Debugging 6-1

Addressing 6-1
Interrupt Processing While Debugging 6-6
Debugging Optimized Code 6-7
Optimizing Debug Performance 6-7
Debugging a Terminated Program 6-8
Debugging a CYBIL Runtime Error 6-8
Debugging Condition Handlers 6-9
Debug Rings 6-10
Multi-task Debugging 6-11

Source Language Debug Examples 7-1

Debugging a BASIC Program 7-1
Debugging a C Program 7-14
Debugging a COBOL Program 7-26
Debugging a CYBIL Program 7-40
Debugging a FORTRAN Version 1 or FORTRAN
Version 2 Program 7-55
Debugging a Pascal Program 7-68

Glossary A-1

Related Manuals B-1

ASCII Character Set C-1

Index Index-1

About This Manual

Audience 5

Organization 5

Conventions 6

Submitting Comments 6

In Case of Trouble 7

About This Manual

This manual describes the Debug utility for the Control Data®
Network Operating System/Virtual Environment (NOS/VE).

Audience

This manual assumes that you understand NOS/VE and SCL concepts as presented in the SCL Language Definition manual and the SCL Interface manual, and the programming concepts of the language that you are debugging. You must also be familiar with the use and manipulation of NOS/VE files. All screen examples use the CDC® Viking 721 terminal; knowledge of this terminal is helpful but not essential.

Organization

Chapters 1, 2, and 3 of this manual are organized by topic based on the concepts of the Debug utility. Chapters 5 and 6 describe the line mode commands and the screen mode commands. The commands are described in quick reference format for easy access. The last chapter contains an interactive screen mode Debug demonstration for each of the following languages: BASIC, C, COBOL, CYBIL, FORTRAN Versions 1 and 2, and Pascal.

Conventions

blue	Command parameter names are shown in blue print. Also, within examples of interactive sessions, user input is shown in blue print. System output is shown in black print.
UPPERCASE	In Debug command syntax, uppercase indicates a statement keyword or character that must be written as shown. For purposes of examples, however, lowercase is used.
lowercase	In Debug command syntax, lowercase indicates a name, number, symbol, or entity that you must supply.
numbers	All numbers are assumed to be base 10 unless otherwise noted.
hex	The abbreviation hex indicates a hexadecimal number.
examples	All screen examples use the CDC® Viking 721 terminal unless otherwise noted.
Quick Reference format	The Debug commands and functions are described in Quick Reference format. The purpose, format, and special remarks of each command or function are described. Also, in most cases, an example is demonstrated.

Submitting Comments

The last page of this manual is a comment sheet. Please use it to give us your opinion of the manual's usability, to suggest specific improvements, and to report technical or typographical errors. If the comment sheet has already been used, you can mail your comments to:

Control Data Corporation
Technology and Publications Division
P.O. Box 3492
Sunnyvale, California 94088-3492

Please indicate whether you would like a written reply.

If you have access to SOLVER, the Control Data facility for reporting software problems, you can use it to submit comments about this manual. When SOLVER prompts you for a product identifier, specify DB8.

In Case of Trouble

Control Data's Central Software Support maintains a hotline to assist you if you have trouble using our products. If you need help beyond that provided in the documentation or find that the product does not perform as described, call us at one of the following numbers and a support analyst will work with you.

From the USA and Canada: (800) 345-9903

From other countries: (612) 851-4131

The preceding numbers are for help on product usage. Address questions about the physical packaging and/or distribution of printed manuals to Literature and Distribution Services at the following address:

Control Data Corporation
Literature and Distribution Services
308 North Dale Street
St. Paul, Minnesota 55103

or you can call (612) 292-2101. If you are a Control Data employee, call (612) 292-2100.

Introduction 1

This chapter introduces the Debug utility for NOS/VE.

Interactive Debugging 1-3
Screen Mode 1-3
Line Mode 1-5

Batch Job Debugging.. 1-6

Introduction 1

Debug for NOS/VE is a utility that helps you symbolically debug a BASIC, C, COBOL, CYBIL, FORTRAN Version 1, FORTRAN Version 2, or Pascal program during execution. Using Debug, you can stop execution at selected points, display the values of selected variables and arrays, and resume execution. The format of the displayed values are consistent with the formatted values used in your program. No knowledge of machine addresses is required. However, you can also display locations by specifying machine addresses.

A primary advantage of Debug is that it is easy to use. You don't need to modify your source program or know assembly language to use Debug. Furthermore, using Debug eliminates the need for such conventional debugging techniques as interpreting memory dumps, inserting PRINT or DISPLAY statements within a program, and using a load map.

Other Debug features let you:

- Suspend execution of your program when a selected event occurs.
- Change the values of program variables while execution is suspended.
- Display a subprogram traceback list, beginning with the current procedure and proceeding back through the sequence of called procedures until the main program is reached.
- Display the environment in which you are currently debugging.
- Step through a program by lines or procedures.
- Create a file of Debug commands that are executed only if an execution error occurs in your program. If an execution error does not occur, the program runs as though Debug were not being used.

Because Debug is a command utility, SCL features are available while using Debug. You can:

- Enter SCL commands.
- Temporarily read commands from a file other than the Debug input file using the SCL command INCLUDE_FILE.
- Enter multiple commands, separated by semicolons, on one line.
- Continue a single command on one or more continuation lines.
- Evaluate and display SCL expressions using the SCL command DISPLAY_VALUE.
- Echo Debug commands to one or more files, and write Debug output to several files using the SCL command CREATE_FILE_CONNECTION.
- Include Debug commands in SCL procedures.
- Enter commands for processing by another active command processor, such as an editor, to examine your load map.

Using Debug requires the following steps:

1. Compile your program for use with Debug so that symbolic capabilities can be used.
2. Turn on Debug mode.
3. Begin execution of your program, which initiates the Debug session.
4. Enter Debug commands to debug your program.
5. End the Debug session.
6. Make corrections to your source program, recompile, and, if necessary, conduct additional Debug sessions.

Debug is intended mainly to be used interactively, but can also be used in batch mode.

Interactive Debugging

An interactive Debug session is the sequence of interactions that takes place at a terminal between you and Debug after you begin execution of a program in Debug mode. Each time you enter a command, Debug processes that command, displays any output or informative messages produced by the command, and waits for you to enter another command.

An interactive Debug session can be conducted in either screen mode or line mode.

Screen Mode

Using screen mode debugging, you can display your Debug session on the screen and debug your program using your terminal's function keys. This feature is available on most terminals.

A primary advantage of debugging in screen mode is its ease of use. You do not need to wait to receive a hardcopy listing before debugging your program; the source code is visible as the program executes. Functions and online Help should free you from needing to read Debug documentation before becoming productive. When combined with the Programming Environment or the Professional Programming Environment, you have a set of powerful programming tools with which to work.

In screen mode, you can:

- View the source code as the program executes (an arrow points to the line about to be executed).
- See the output generated by your program and the messages from Debug displayed to the screen.
- See special information about your task, such as historical or tracing information.
- View module components of the task.

Communication with Debug in screen mode is accomplished with functions. Most functions are assigned to function keys displayed at the bottom of your screen. You can also enter the functions at the home line. The functions provide capabilities similar to line mode Debug commands; their names suggest their purpose. In addition to the functions, you can also enter most line mode Debug commands and SCL commands on the home line.

The following screen (displayed on a Viking 721 terminal) typifies an interactive Debug session in screen mode:

```
Debugging FORT
-->   program fort
      real average, total
      total=0.
      call aver(total)
      average=total/5.
      print*,'average = ',average
      end
      subroutine aver(x)
      real scores(5)
      data scores /50., 56., 98., 12., 78./
      do 10 i=1,5
 10   x=scores(i)+x
      end
------------------------------ OUTPUT ------------------------------
              -- Welcome to Full Screen Debugging --

                     Press HELP for Assistance

   StepN     MSpeed    Locate    ChaVal    DelBrk    Deas      ZmOut     Keys
f1 Stepl  f2 MSpeed f3 HSpeed f4 SeeVal f5 SetBrk f6 Quit   f7 Trace  f8 Goto
```

Screen mode debugging is subject to the following restrictions:

- The GOTO function can cause unpredictable results. For example, if GOTO is used to transfer into a loop construct, the control variable controlling the loop may not have been initialized; this could yield unexpected output.

- If your program has user supplied line numbers that do not correspond to the index of the line in the source file, you cannot use screen mode debugging.

- Screen mode Debug cannot reliably determine if the source code being displayed corresponds to the object code being debugged. This occurs if you have made modifications to your source code but did not recompile it. If the source code and the object code are not in agreement, you can continue debugging; however, the source code being displayed shows the latest modifications, but the object code being debugged represents output from a previously compiled source.

- If your program invokes another process that writes to the terminal, such as SCU (Source Code Utility), you cannot use screen mode debugging.

Line Mode

Using line mode debugging, you debug your program by entering commands in response to the Debug prompt

```
DB/
```

After you enter a command, Debug processes that command and issues another DB/ prompt. This process continues until you end the Debug session.

When debugging in line mode, you can use any of the Debug and SCL commands.

Following is a simple example of a Debug session in line mode:

`/execute_task file=lgo debug_mode=on`	Execute the program and begin the Debug session.

`DEBUG 1.5`
`DB/set_break line=5`
`-- Break name DBB$1 assigned to this break`
	Set a break at line 5 of the program.

`DB/run`	Begin execution of the program.
`-- DEBUG: break DBB$1, execution at M=FORT at L=5`	The program runs until the break is encountered.
`DB/display_program_value name=total`	Display the value of variable TOTAL.
`total = 294.`	Debug displays the value of TOTAL.
`DB/run`	Resume execution.

`Average = 58.8`
`-- DEBUG: program terminated by calling exit at M=FLM$BOUND_CORE BO=1332(16)`
`-- DEBUG: The status at termination was: NORMAL.`
	The program terminates.

`DB/quit`
`-- DEBUG: QUIT terminated task`
`/`
	End the Debug session.

Batch Job Debugging

You can also use Debug at the batch level. When debugging in batch mode, Debug reads its commands from a specified file. This file is specified on the DEBUG_INPUT parameter of the SET_PROGRAM_ATTRIBUTES, CREATE_PROGRAM_DESCRIPTION, or EXECUTE_TASK command at execution time. (The DEBUG_INPUT parameter is described in chapter 3.)

Once Debug is accessed, it makes no distinction between interactive line mode and batch mode operations. All Debug and SCL commands and features are available in both; all commands and features are read and processed in the same way.

Getting Started 2

This chapter describes the preparations required for your terminal and your source program before beginning a Debug session.

Set Up Your Terminal for Screen Mode Debug 2-2
Full Screen Terminal Characteristics 2-2
Full Screen Terminal Definition 2-3
Using Function Keys 2-3
User Breaks in Screen Mode Debug 2-4
NOS/VE Dual State With CDCNET or NOS/VE Standalone 2-5
NOS/VE Dual State With 2550s 2-5

Preparing for a Debug Session 2-6
Symbolic Debugging 2-6
Machine-level Debugging 2-6
Optimizing Level for Use With Debug 2-7
Summary of Compilation Command Parameters 2-7
Examples 2-8

Getting Started 2

Debug can be used in screen mode or line mode. To use Debug in screen mode, you must define your terminal for full screen interface. Debug can always be used in line mode. If the terminal is not defined as a full-screen terminal, Debug always uses line mode. Otherwise, if the terminal is defined as a full-screen terminal and Debug is being used in screen mode, you can switch to line mode by pressing the DEAS function. (The DEAS function is described in chapter 4).

Once your terminal is defined correctly, you can prepare your program for a Debug session.

Set Up Your Terminal for Screen Mode Debug

Because screen mode Debug uses a full-screen interface, it can be used only from an interactive terminal that has full-screen capabilities and is defined for full-screen use.

If you have used a full-screen application from your terminal (such as editing in screen mode using the NOS/VE Full-Screen Editor), your terminal is a full-screen terminal and you can prepare your terminal for Debug the same way you prepare it for other screen mode applications.

However, if you have not used a screen mode application from your terminal before, you need to learn how to define your terminal as a full-screen terminal. You may be able to find out from someone else at your site. If not, continue reading. The following paragraphs list the characteristics of a full-screen terminal and describe how you define your terminal as a full-screen terminal.

Full-Screen Terminal Characteristics

A terminal can be defined as a full-screen terminal if it has the following minimum characteristics:

- Uses asynchronous communications.
- Operates in character mode, not block mode.
- Has keys that move the cursor on the screen and transmit characters indicating that the cursor has moved.
- Supports direct cursor addressing.
- Provides a clear-screen operation.

The following terminal characteristics are also desirable:

- Provides a clear-to-end-of-line function.
- Has up to 32 definable function keys, each of which transmits a unique, identifying character sequence. Preferably, the sequence should end with the carriage-return character.
- Allows host definition of tab stops.
- Supports protected screen fields and tabbing between unprotected fields. The tab key must transmit a character sequence to the host indicating that the key was pressed.
- Has graphic characters for drawing lines.

Full-Screen Terminal Definition

A terminal is defined as a full-screen terminal when you provide NOS/VE with a full-screen terminal definition to use during the interactive session. You do this by specifying a TERMINAL_MODEL parameter value on the SCL command SET_TERMINAL_ATTRIBUTE or CHANGE_TERMINAL_ATTRIBUTE. You can put the SET_TERMINAL_ATTRIBUTE or CHANGE_TERMINAL_ATTRIBUTE command in your prolog file ($USER.PROLOG) so it is executed each time you log in to NOS/VE.

The TERMINAL_MODEL parameter value references a compiled terminal definition. Your site probably has a set of compiled terminal definitions available on file $SYSTEM.TDU.TERMINAL_DEFINITIONS. To see the definitions available on the file, enter the following SCL command:

```
/display_object_library $system.tdu.terminal_definitions
```

The command lists the names of the terminal definition modules. By convention, the names begin with the prefix CSM$ followed by the TERMINAL_MODEL value. For example, one of the modules may be a terminal definition for the Zenith Z29 terminal names CSM$Z29. To use the CSM$Z29 module, you enter the following command:

```
/set_terminal_attribute terminal_model=z29
```

If you cannot find a compiled terminal definition available at your site that is effective for your full-screen terminal, you require creation of a new terminal definition. The process of creating a new terminal definition is described in the Terminal Definition for NOS/VE manual.

Using Function Keys

One of the important features of the full-screen interface is the use of function keys. However, the actual function keys available to you depend on the terminal you use.

You may need to consult the manual for your terminal or the person who wrote the terminal definition to find the function keys on your keyboard. For some terminals, function keys are entered by a combination of keys entered at the same time or in sequence. For example, on the Z29 terminal, unshifted function keys are the top row of keys on the keyboard and shifted function keys are the SHIFT key and a numeric pad key entered at the same time. After entering a Z29 function key, you must press the RETURN key.

As listed under Full-Screen Terminal Characteristics, to use Debug in screen mode, function keys are not required. When the terminal definition does not define function keys, all Debug actions must be specified by commands entered on the home line.

In many cases, a terminal has some function keys, but not the full set. In those cases, some Debug functions are available via function key, while others must be entered by commands on the home line. The function keys defined by the terminal definition are displayed at the bottom of the screen.

For example, the following shows the function key display at the bottom of the Debug screen using the Z29 terminal definition:

	s1	s2	s3	s4	s5	s6	s7	s8
Shifted	First	Last	SetBrk	StepN	DelBrk	Deas	Locate	ChaVal
Unshifted	BWD	FWD	Back	Step1	MSpeed	Quit	HSpeed	SeeVal

Function keys can be entered shifted or unshifted. On the display at the bottom of a Debug screen, the top label is for the shifted key and the bottom label is for the unshifted key.

The function key label is always an alias for the corresponding home line command. Thus, when a description refers to a function key label, the referenced function can be performed either by pressing the function key or entering the command on the home line of the screen.

User Breaks in Screen Mode Debug

To activate pause breaks and terminate breaks in a full screen application such as Debug, you must enter one or two commands before entering the application.

Pause break and terminate break are interactive conditions caused when the user enters a certain key sequence. A terminate break condition terminates the currently executing command.

A pause break condition discards typed-ahead input and suspends the executing command. While the command is suspended, you can interact with the system to get information such as the job's status, consult online manuals and so forth. To resume the suspended command, enter RESUME_COMMAND; to terminate the suspended command, enter TERMINATE_COMMAND.

It may be convenient to include the commands required to activate pause break and terminate break in your user prolog so they are automatically executed when you login. The required commands depend on how you access NOS/VE.

NOS/VE Dual State With CDCNET or NOS/VE Standalone

If you use NOS/VE dual state with CDCNET or NOS/VE standalone, you must enter two commands. The first command defines the attention character value; the second command defines the attention character as the terminate break key and the terminal's break key as the pause break key. The following example assumes that the network control character is the default (%) and the attention character is to be defined as CTRL-T (ASCII DC4), character code 20:

```
/%change_terminal_attribute attention character
/%change_connection_attribute attention_character_action=2 ..
../break_key_action=1
```

For more information on the CHANGE_TERMINAL_ATTRIBUTE and CHANGE_CONNECTION_ATTRIBUTE commands, see the SCL System Interface manual.

NOS/VE Dual State With 2550s

If you use NOS/VE dual state with 2550s, you enter only one command. The command changes the attention character to a non-null value. For example:

```
/set_terminal_attribute attention_character=$char(1)
```

For more information on the SET_TERMINAL_ATTRIBUTE command, see the SCL System Interface manual.

NOTE

If a subsequent CHANGE_TERMINAL_ATTRIBUTES command is entered that does not specify the ATTENTION_CHARACTER parameter, the attention character is reset to null.

Preparing for a Debug Session

As previously noted, using Debug requires no changes to your source program. However, you must specify two additional command parameters when you compile your program, if you intend to use the symbolic capabilities of Debug. These are the Debug parameter and the Optimization parameter. The Debug parameter controls the generation of symbol and line number tables for symbolic debugging and the Optimization parameter modifies the object code to be used with Debug.

Symbolic Debugging

Before you can begin a Debug session using symbolic debugging features, you must compile your program with the Debug parameter option. Symbolic debugging allows you to reference program locations symbolically. Code locations can be referenced by their line number generated by the compiler. Data locations can be referenced using the data identifiers declared in the source program.

Symbolic debugging is available for user programs written in BASIC, C, COBOL, CYBIL, FORTRAN Version 1, FORTRAN Version 2 and Pascal.

To symbolically debug a COBOL, CYBIL, FORTRAN Version 1, FORTRAN Version 2, or Pascal program, the DEBUG_AIDS=DT or DEBUG_AIDS=ALL parameter is required on the compile command. To symbolically debug a C program, the -g option is required on the compile command. If your program is written in BASIC, the symbol tables are automatically generated at compilation; no special compilation is necessary. However, if your BASIC program calls an external subroutine written in COBOL, CYBIL, FORTRAN Version 1, FORTRAN Version 2, or Pascal, the subroutine must be compiled with the Debug parameter option.

Machine-level Debugging

You can also debug your program at the machine-level. Machine-level debugging requires that you reference program locations by their machine address (their NOS/VE process virtual address [PVA]). For example, to display variable X, you would have to specify the machine address of X. At this level, module address tables indicating where program modules are located are also available.

Machine-level debugging does not require the Debug tables and so, does not require the Debug parameter options. Machine-level debugging is more difficult than using symbolic names; however, you can use it to debug an existing object program that would be to costly to recompile.

Optimizing Level for Use With Debug

The Optimization parameter controls the level of optimization performed by the compiler. Optimized code executes faster than unoptimized code, but requires more compilation time. When the optimization level is set to Debug, the minimum optimization is selected and the object code generated is modified for more efficient use with Debug.

To compile a COBOL, CYBIL, FORTRAN Version 1, FORTRAN Version 2, or Pascal program for use with Debug, the OPTIMIZATION_LEVEL=DEBUG parameter is required on the compile command. To compile a C program for use with Debug, the -R option is required on the compile command. If your program is written in BASIC, no special optimization level is necessary.

You can still use Debug if the Optimization parameter is set to a higher level, but some lines may be removed from the program during optimization and some variables may not be available. (For more information on Optimizing Debug, see chapter 6.)

Summary of Compilation Command Parameters

Table 2-1 summarizes the parameters required for symbolic debugging and Debug optimization.

Table 2-1. Compilation Parameters Required for Debug

Compiler	Parameters
BASIC	No compilation parameters required.
CC	-g -R
COBOL	DEBUG_AIDS=DT (or DEBUG_AIDS=ALL) OPTIMIZATION_LEVEL=DEBUG
CYBIL	DEBUG_AIDS=DT (or DEBUG_AIDS=ALL) OPTIMIZATION_LEVEL=DEBUG
FORTRAN	DEBUG_AIDS=DT (or DEBUG_AIDS=ALL) OPTIMIZATION_LEVEL=DEBUG
Pascal	DEBUG_AIDS=DT (or DEBUG_AIDS=ALL) OPTIMIZATION_LEVEL=DEBUG
VECTOR_ FORTRAN	DEBUG_AIDS=DT (or DEBUG_AIDS=ALL) OPTIMIZATION_LEVEL=DEBUG

Examples

In the following example, the FORTRAN program contained in permanent file $USER.FTN_SOURCE is compiled for symbolic debugging. The DEBUG_AIDS parameter, set to ALL, generates the tables required for symbolic debugging; the OPTIMIZATION_LEVEL parameter, set to DEBUG, requests Debug optimization.

```
/fortran input=$user.ftn_source list=list binary_object=lgo ..
../debug_aids=all optimization_level=debug
```

The following example compiles a C program contained in permanent file $USER.SOURCE_C for symbolic debugging. The -g option on the CC command generates the tables required for symbolic debugging; the -R option sets the optimization level for Debug use.

```
/$system.cve.setup
/set_working_catalog $user
/cc -g -R source_c
```

Executing Under Debug Control 3

This chapter describes the Debug features available when executing your program under Debug control.

Beginning a Debug Session 3-1
- Beginning a Debug Session in Screen Mode 3-2
- Beginning a Debug Session in Line Mode 3-4
- Switching to Screen Mode From Line Mode 3-5
- Switching to Line Mode From Screen Mode 3-5

Entering Debug Commands 3-6

Suspending Program Execution 3-7
- Setting Breaks 3-7
- Suspend Execution Automatically on Program Error 3-8
- Suspend Execution at Intervals of Lines or Procedures (Setting Step Mode) 3-9

Beginning or Resuming Program Execution 3-10
- The HSPEED Function (Screen Mode) 3-10
- The MSPEED Function (Screen Mode) 3-10
- The GOTO Function (Screen Mode) 3-11
- The RUN Command (Line Mode) 3-11
- Changing the P Register (Line Mode) 3-11

Displaying and Changing Program Values 3-13

Getting HELP 3-14
- The HELP Function (Screen Mode) 3-14
- The HELP Command (Screen Mode and Line Mode) 3-14
- The DISPLAY_COMMAND_INFORMATION Command (Screen Mode and Line Mode) 3-15
- The DISPLAY_FUNCTION_INFORMATION Command (Screen Mode and Line Mode) 3-15

Ending the Debug Session 3-16
- Ending the Debug Session in Screen Mode 3-16
- Ending the Debug Session in Line Mode 3-16

Debug Input and Output Files 3-17
- Changing the Debug Input File 3-17
- Changing the Debug Output File 3-20

Automatic Command Execution on Program Failure 3-21

Recording an Interactive Debug Session 3-26

Executing Under Debug Control 3

In order to execute a program under Debug control, you must first request a mode of execution called debug mode. Through Debug, you can suspend program execution at selected break points or on program failure, display or change the values of variables and arrays within the program, resume execution, and perform a variety of other tasks.

Beginning a Debug Session

After you compile your program for use with Debug, you are ready to begin a Debug session.

To begin a Debug session, you must first turn on Debug mode, then execute your program while in Debug mode. To turn on Debug mode, you specify the parameter DEBUG_MODE=ON on the SET_PROGRAM_ATTRIBUTES, CREATE_PROGRAM_DESCRIPTION, or EXECUTE_TASK commands.

The SET_PROGRAM_ATTRIBUTES command (described in the SCL Object Code Management manual) turns Debug on at the job level. This means that all subsequent programs (that do not turn Debug mode off at the program level) are executed under control of Debug, unless you specify DEBUG_MODE=OFF on a SET_PROGRAM_ATTRIBUTES command.

The CREATE_PROGRAM_DESCRIPTION command (described in the SCL Object Code Management manual) is a subcommand of the Object Library Generator and can turn Debug on at the program level. This means that only the specified program (or the implied program if the program or file name is not specified) will be executed under control of Debug.

The EXECUTE_TASK command (described in the SCL Object Code Management manual) turns Debug on at the program level. This means that only the specified program will be executed under control of Debug.

You can begin a Debug session in screen mode or line mode.

Beginning a Debug Session in Screen Mode

Using Debug in screen mode requires that your terminal support full screen operation. See chapter 2 for information about terminal definitions that can support full screen interface.

To begin a Debug session in screen mode, perform the following steps:

1. Set the current style for your terminal to SCREEN. This is done with the SCL CHANGE_INTERACTION_STYLE command (described in the SCL System Interface manual). For example:

   ```
   /change_interaction_style style=screen
   ```

   This command enables the Debug session to begin in screen mode when you execute your program while in Debug mode. Once set, the terminal style remains in effect until you change it.

2. Turn on Debug mode. To do this, set the parameter DEBUG_MODE to ON specified on the SET_PROGRAM_ATTRIBUTES, CREATE_PROGRAM_DESCRIPTION, or EXECUTE_TASK commands. For example:

   ```
   /set_program_attributes debug_mode=on
   ```

3. Specify a name call or EXECUTE_TASK command. This begins the Debug session in screen mode. For example:

   ```
   /execute_task file=lgo
   ```

   For programs written in C, COBOL, CYBIL, FORTRAN Version 1, or FORTRAN Version 2, the source listing of the file containing the program is displayed on the screen. This is called the Zoom-in display (described in chapter 4) and it is the display in which you debug your program.

   For programs written in BASIC or Pascal, the module names associated with the program are displayed on the screen. This is called the Zoom-out display (described in chapter 4). In order to show the Zoom-in display, that is, display the source listing of the program, you must select the module you want to debug and then press the ZMIN function (described in chapter 4).

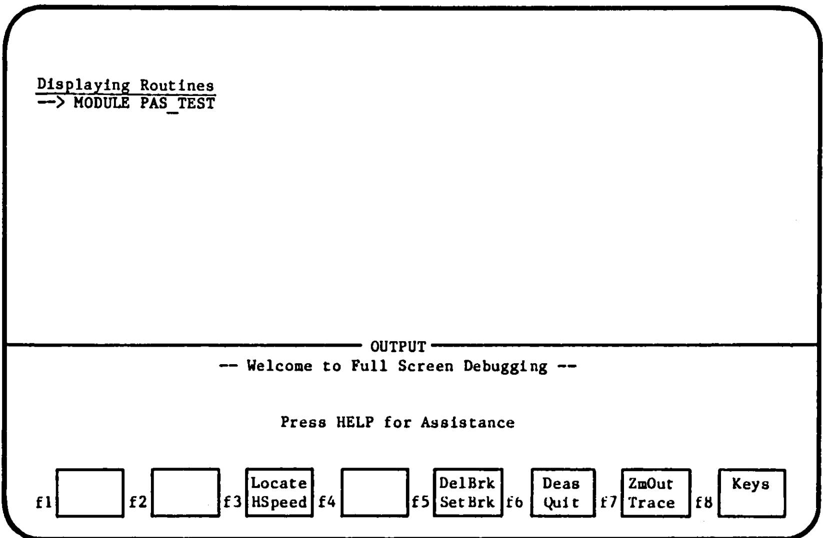
NOTE

Screen Mode Debug can also be accessed through the Programming Environment or the Professional Programming Environment. These utilities are described in the ENVIRONMENT and PPE online manuals, respectively.

In the following example, the permanent file $USER.PAS_TEST contains a Pascal program. Assuming $USER.PAS_TEST has already been compiled for use with Debug, the commands below set the terminal style to screen and begin the Debug session in screen mode:

```
/change_interaction_style style=screen
/execute_task file=lgo debug_mode=on
```

The following screen is displayed:



To display the source listing of program PAS_TEST, press the ZMIN function.

Beginning a Debug Session in Line Mode

A Debug session in line mode begins when you enter a name call or EXECUTE_TASK command to begin execution of a program while in debug mode. Normally, this begins execution of your program. However, in debug mode, the following Debug prompt is displayed:

```
DB/
```

and you debug your program by entering commands in response to this prompt.

To begin a Debug session in line mode, perform the following steps:

1. Turn on Debug mode. To do this, set the parameter DEBUG_MODE to ON specified on the SET_PROGRAM_ATTRIBUTES, CREATE_PROGRAM_DESCRIPTION, or EXECUTE_TASK commands. For example:

   ```
   /set_program_attributes debug_mode=on
   ```

2. Specify a name call or EXECUTE_TASK command. This begins the Debug session in line mode. Debug immediately gets control and waits for you to enter a command. For example:

   ```
   /execute task file=lgo
   DEBUG 1.5
   DB/
   ```

In the following example, the permanent file $USER.COB_TEST contains a COBOL program. Assuming $USER.COB_TEST has already been compiled for use with Debug, the command below begins a Debug session in line mode:

```
/execute_task file=lgo debug_mode=on
DEBUG 1.5
DB/
```

NOTE

A Debug session in line mode is also initiated when a STOP statement is encountered in a BASIC program. This is true even if the paramenter DEBUG_MODE=OFF is specified. The program resumes execution when a Debug RUN command is entered.

Switching to Screen Mode From Line Mode

If you are debugging your program in line mode, you can switch to screen mode anytime during the debugging session by entering the Debug ACTIVATE_SCREEN command (described in chapter 5). Any existing breaks are deleted and STEP_MODE is turned off when ACTIVATE_SCREEN is entered.

For example, to switch to screen mode from line mode, enter the Debug ACTIVATE_SCREEN command:

```
DB/activate_screen
```

NOTE

If you are debugging a BASIC or Pascal program, you must be sure the FILE_PROCESSOR attribute of the file containing the source specifies the name of the compiler that compiled the program. This is done with the SCL CHANGE_FILE_ATTRIBUTE command (described in the SCL System Interface manual).

For example, assume that permanent file $USER.PAS_TEST contains the source for a Pascal program. The following command defines the FILE_PROCESSOR attribute to be PASCAL:

```
DB/change_file_attributes file=$user.pas_test ..
DB../file_processor=pascal
```

To determine what the FILE_PROCESSOR attribute of a file is, you can enter the SCL DISPLAY_FILE_ATTRIBUTE command (described in the SCL System Interface manual). The CHANGE_FILE_ATTRIBUTE and DISPLAY_FILE_ATTRIBUTE commands can be entered before the Debug session begins or anytime during a Debug session while execution is suspended.

Switching to Line Mode From Screen Mode

If you are debugging your program in screen mode, you can switch to line mode anytime during the Debug session by pressing the DEAS function (described in chapter 4). Any existing breaks remain set, step-mode is turned off, and all options return to their default settings.

Entering Debug Commands

During a Debug session in screen mode, you execute Debug screen mode commands by using function keys instead of typing the command. Debug screen mode commands, Debug line mode commands, and SCL commands can also be entered on the home line. During a Debug session in line mode, you enter Debug line mode commands in response to prompts.

When a function or command is processed, any output and informative messages produced by the function or command are displayed. Using the Debug functions and commands, you can suspend program execution, display or change the values of variables and arrays within the program, resume execution, end the Debug session, and perform a variety of other tasks.

You can also create a file of Debug line mode commands that are executed automatically if an error occurs while your program is executing in line mode. (This feature is described later in this chapter under Automatic Command Execution.)

Suspending Program Execution

At the beginning of a Debug session you will typically want to suspend execution at various points in the program, so that you can display and alter program values, or perform other tasks. You can suspend execution in the following ways:

1. Set Breaks

   Execution is suspended when a specific line, statement, or other event is reached during execution. This requires you to set one or more breaks in your program. When the executing program encounters a break, execution is suspended.

2. Automatically on Program Error

   Execution is suspended automatically when an error occurs during execution (such as division by zero, arithmetic overflow, or a CYBIL range-error). This requires no action on your part. If you simply turn on debug mode and begin execution of your program, execution will stop if a runtime error occurs.

3. Set Step Mode

   Execution is suspended immediately before execution of each source line or procedure in the program. This is known as executing in step mode. With step mode you can step through your program at intervals of lines or procedures.

Setting Breaks

The primary means for suspending program execution during a Debug session is to set breaks at selected points in your program. With a break, you can suspend execution of your program when an event occurs in a specified address range at which point you receive control and Debug commands can be entered. Many events can be specified and address ranges can be specified in many forms.

In screen mode debugging, you set a break and delete a break with the SETBRK and DELBRK functions (described in chapter 4). The line which contains the break is highlighted to inform you that a break is set on the line. In line mode debugging, you set a break, delete a break, and display a break with the SET_BREAK, DELETE_BREAK, and DISPLAY_BREAK commands (described in chapter 5).

Once set, a break stays set until it is explicitly deleted.

PERFORMANCE HINT

Because execution takes longer when there are breaks set, you should delete a break as soon as it is no longer needed.

When a break is encountered during program execution, the following events occur:

1. Execution of the program suspends.
2. Control passes to Debug.
3. Execution stops on the line containing the break. The break occurs before the statement is executed and you can then enter Debug functions or commands.

In line mode debugging, commands can be associated with a break such that when the break is reached during execution, the commands are automatically executed. This can be useful when you want to temporarily compensate for a program bug, automatically display certain critical values each time the break occurs, and test certain conditions (perhaps the number of times the break has occurred).

Suspend Execution Automatically on Program Error

You can also suspend execution during a Debug session simply by beginning execution of the program and allowing it to execute until an error occurs. Debug then gets control, displays a message describing the error, and gives control to you. This is true even if the program has condition handlers established for these exceptions. Debug gets control first. The program's handler is activated when execution is resumed with the RUN command.

Suspending execution automatically during a Debug session when an error occurs is a result of the default error termination breaks provided by Debug. Examples of error termination breaks are the Divide Fault break and the Exponent Overflow break. You do not need to explicitly set these breaks because they are default breaks that are automatically set for every Debug session.

A suggested procedure for debugging a program is to execute the program in Debug mode without setting any breaks, and to allow the program to terminate. If the program terminates with errors, you can run additional Debug sessions using knowledge gained from the first session to help you decide where to set breaks.

Suspend Execution at Intervals of Lines or Procedures (Setting Step Mode)

A third method of suspending execution during a Debug session is to set step mode. Step mode is a mode of execution in which execution is suspended immediately before the execution of each program statement. Execution remains suspended until you enter the command to resume execution. Using step mode, you can step through lines or procedures of your executing program, displaying or changing program values at each step.

In screen mode debugging, you can step through your program using the STEP1 or STEPN functions (described in chapter 4). In line mode debugging, you initiate step mode with the Debug command SET_STEP_MODE set to ON (described in chapter 5).

Beginning or Resuming Program Execution

After beginning a Debug session, there are several ways you can begin execution of your program, or resume execution while in step mode or after the occurrence of a break:

- The HSPEED Function
- The MSPEED Function
- The GOTO Function
- The RUN Command
- Changing the P Register

In addition, if the program has a condition handler for the condition that handler is executed. If there is no such handler, the program resumes execution at the point of interruption. (Condition handlers are described in chapter 6.)

The HSPEED Function (Screen Mode)

Pressing the HSPEED function (also described in chapter 4) begins or resumes execution of your program. Execution continues uninterrupted until your program terminates, a break is encountered, or an unselected event (such as a divide fault, a terminate break, or a pause break) occurs (an arrow points to the source line in your program where execution has stopped).

The MSPEED Function (Screen Mode)

Pressing the MSPEED function (also described in chapter 4) executes your program until the next subroutine or procedure is called. (For COBOL programs, MSPEED executes the program until the next section is called.) MSPEED can be interrupted by a break, program termination, or an unselected event (such as a divide fault, a terminate break, or a pause break) occurs (an arrow points to the source line in your program where execution has stopped.

The GOTO Function (Screen Mode)

The GOTO function (also described in chapter 4) changes the location where execution of your program begins next when the HSPEED, MSPEED, LSPEED, STEP1, or STEPN function is pressed. The GOTO function is intended to be used when a bug has been isolated and can be fixed in the Debug session. You can test the change without recompiling your program by resuming execution of a portion of code with the new data values. The GOTO function can also be used to skip a section of faulty code.

The RUN Command (Line Mode)

You enter a RUN command to begin or resume execution of your program at the point where it was suspended. Execution continues until either another break occurs or the program runs to completion. If you enter a RUN command after your program has run to completion, the Debug session automatically ends and control returns to the activity you were doing before you began the Debug session. (The RUN command is described in chapter 5.)

Changing the P Register (Line Mode)

You can resume execution at another point in your program by changing the P register before entering the RUN command.

Normally, execution begins at the instruction whose address is contained in the P register of the program in which execution is suspended. If the P register points to the instruction that caused the event, the same event will occur immediately after entering the RUN command. In this case, you must change the value in the P register with the CHANGE_REGISTER command or change the value of one of the operands with the CHANGE_PROGRAM_VALUE command before entering the RUN command. (The CHANGE_REGISTER and CHANGE_PROGRAM_VALUE commands are described in chapter 5.)

The value of the P register or the value of the operands must be changed when the following events occur:

- Arithmetic overflow. This is caused when the result of an integer operation exceeds the largest value that can be represented in integer format.

- Arithmetic significance is lost. This is caused when an operation on extremely large integer values caused truncation of the low-order digits of the integer result.

- Divide fault. This is generally caused when the program attempts to divide by zero. (If you are debugging a COBOL program, you can change the value of the variable with the CHANGE_PROGRAM_VALUE command and resume execution of the program without changing the P register.)

- Exponent overflow. This is caused when the result of a mathematical calculation is a floating-point number that exceeds the largest number that can be represented in floating-point format.

- Exponent underflow. This is caused when the result of a mathematical calculation is a floating-point number that exceeds the smallest number that can be represented in floating-point format.

- Floating-point indefinite. This is caused when a floating-point calculation cannot be resolved, such as a division where the dividend and divisor are both zero.

- Floating-point significance is lost. This is caused when an operation on extremely large floating-point values caused truncation of the low-order digits of the floating-point result.

- Invalid business data processing data (described in the Virtual State Hardware Usage manual).

If you do change the P register, be sure its new value is within the procedure that was executing when Debug gained control. Unpredictable results can occur if the P register is changed to a value outside the current procedure.

Displaying and Changing Program Values

After execution of your program is suspended, you can examine and change the contents of variables and arrays. If the value of the variable you displayed is incorrect, you can replace that value with a new value. Then, when you resume execution, the new value is used in subsequent computations.

In screen mode debugging, the functions SEEVAL and CHAVAL are used to display and change values of variables or arrays. (These functions are described in detail in chapter 4.)

In line mode debugging, the commands DISPLAY_PROGRAM_VALUE and CHANGE_PROGRAM_VALUE are used to display and change values of variables or arrays. (These commands are described in detail in chapter 5.)

Getting HELP

During a Debug session, there are several ways to get help information:

- The HELP Function
- The HELP Command
- The DISPLAY_COMMAND_INFORMATION Command
- The DISPLAY_FUNCTION_INFORMATION Command

The HELP Function (Screen Mode)

Pressing the HELP function displays the Help window. The Help window overlays a portion of your screen and prompts you to press the function key for which you need help. When you press a function key, a short description of the function you select is displayed in the Help window. If you press RETURN in response to the prompt, the Help window is removed and your screen is restored to its original contents. (The HELP function is also described in chapter 4.)

The HELP Command (Screen Mode and Line Mode)

The SCL HELP command displays the text of an online manual. You can get detailed help for a particular subject from any online manual during a Debug session. The HELP command can be entered in line mode Debug or on the home line in screen mode Debug.

For example, if you need information about the Debug SET_BREAK command described in the DEBUG online manual, type the HELP command shown below on the home line or in response to the DB/ prompt:

```
help subject=set_break manual=debug
```

This command takes you to the DEBUG online manual for an explanation of the SET_BREAK command. To return to Debug, press QUIT. If you are debugging in line mode, the DB/ prompt returns and debugging continues; if you are debugging in screen mode, the screen is restored to its original contents.

The DISPLAY_COMMAND_INFORMATION Command (Screen Mode and Line Mode)

The SCL DISPLAY_COMMAND_INFORMATION (DISCI) command displays format information about an SCL or Debug command. The DISPLAY_COMMAND_INFORMATION command can be entered in line mode Debug or on the home line in screen mode Debug. This command is useful when you want a quick reminder of the format of a command.

For example, to display the format and required parameters of the Debug DISPLAY_CALLS command, type the DISPLAY_COMMAND_INFORMATION command shown below on the home line or in response to the DB/ prompt:

```
display_command_information display_calls
count, c                   : integer 1..2147483647 or key all =
                                  10000
start, s                   : integer 1..2147483647 = 1
display_option, ..
display_options, do        : list of key user_calls, uc, ..
                                  system_calls, sc, all_calls, ac, ..
                                  variable_values, vv=user_calls
output, o                  : file = $optional
status                     : var of status = $optional
```

The DISPLAY_FUNCTION_INFORMATION Command (Screen Mode and Line Mode)

The SCL DISPLAY_FUNCTION_INFORMATION (DISFI) command displays format information about an SCL or Debug function. The DISPLAY_FUNCTION_INFORMATION command can be entered in line mode Debug or on the home line in screen mode Debug. This command is useful when you want a quick reminder of the format of a function.

For example, to display the format and required parameters of the Debug $REGISTER function, type the DISPLAY_FUNCTION_INFORMATION command shown below on the home line or in response to the DB/ prompt:

```
display_function_information $register
parameter 1 : key p, a, x = $required
parameter 2 : integer 0..15 = $optional
```

NOTE

You can use DISPLAY_COMMAND_INFORMATION and DISPLAY_FUNCTION_INFORMATION to display Debug command and function parameters only while you are in a Debug session. Outside the Debug session, only SCL command and function parameters can be displayed with the DISPLAY_COMMAND_INFORMATION and the DISPLAY_FUNCTION_INFORMATION commands.

Ending the Debug Session

When the Debug session ends, control returns to the activity you were doing before you began the Debug session. For example, if you were using the Programming Environment utility, you return to the Programming Environment when you terminate the Debug session.

Ending the Debug Session in Screen Mode

You can end a Debug session in screen mode at any time by pressing the QUIT function. When you press QUIT, you return to the activity you were doing before you began the Debug session. You can also enter QUIT on the home line. (The QUIT function is also described in chapter 4.)

Ending the Debug Session in Line Mode

You can end a Debug session in line mode at any time by typing the command

```
QUIT
```

in response to the DB/ prompt. When you enter a QUIT command, the following message is displayed:

```
-- DEBUG: QUIT terminated task
```

and you return to the activity you were doing before the Debug session began.

You can also end the Debug session by entering the RUN command after your program runs to completion. In this case, no message is displayed; you return immediately to the previous activity. (The QUIT and RUN commands are described in chapter 5.)

NOTE

Changes made during a Debug session are lost when the session is ended. All variables assume their original values, breaks are removed, and the program is the same as when you compiled it. You can run additional sessions if you want to continue debugging your program.

Debug Input and Output Files

The Debug input file is the file from which Debug reads its commands. The Debug output file is the file to which Debug writes its output (messages and displays). You can, by manipulating these files expand the capabilities of Debug. This feature is available only when debugging in line mode.

The Debug input and output files are initially specified by the DEBUG_INPUT and DEBUG_OUTPUT parameters. You can change the Debug input and output files by specifying the DEBUG_INPUT or DEBUG_OUTPUT parameter on the SET_PROGRAM_ATTRIBUTES, CREATE_PROGRAM_DESCRIPTION, or EXECUTE_TASK command.

Changing the Debug Input File

The DEBUG_INPUT parameter specifies the input file from which Debug reads its commands. The default input file is $LOCAL.COMMAND. For interactive jobs, COMMAND is your terminal. For batch jobs, COMMAND is the normal command stream.

Since the file COMMAND is positioned at its beginning when first accessed by Debug, which is the LOGIN command for the job, this will be read as the first Debug command. Instead, to use Debug in batch mode, use the COLLECT_TEXT command to enter the Debug commands onto another file and then specify that file on the DEBUG_INPUT parameter.

To change the Debug input file, assume that the permanent file $USER.INPUT_FILE contains the following Debug commands:

```
set_break break=bl line=5 module=fort
run
display_program_variable name=total
run
```

The following EXECUTE_TASK command is entered to execute the FORTRAN program illustrated in figure 3-1. Debug mode is turned on and the Debug input file is specified as $USER.INPUT_FILE. When the Debug session begins, the commands in file $USER.INPUT_FILE are immediately executed.

```
/execute_task file=lgo debug_mode=on debug_input=$user.input_file
DEBUG 1.5
 -- DEBUG: break B1, execution at M=FORT L=5
total=294.
Average = 58.8
 -- DEBUG: program terminated by calling exit at
    M=FLM$BOUND_CORE BO=1332(16)
 -- DEBUG: The status at termination was: NORMAL.
DB/
```

When the commands in file $USER.INPUT_FILE have executed, the Debug prompt returns.

```
SOURCE LIST OF FORT

1       program fort
2       real average, total
3       total=0.
4       call aver(total)
5       average=total/5.
6       print*,'average = ',average
7       end

SOURCE LIST OF AVER

1       subroutine aver(x)
2       real scores(5)
3       data scores/50.,56.,98.,12.,78./
4       do 10 i=1,5
5   10  x=scores(i)+x
6       end
```

Figure 3-1. Debug Example: Source Listing of Program FORT

You can also change the Debug input file temporarily by entering an SCL INCLUDE_FILE command during the Debug session. As soon as the INCLUDE_FILE command is entered, commands are read from the file specified on the INCLUDE_FILE command until an end-of-file or a RUN command is encountered.

If an end-of-file is encountered, Debug gains control; the Debug input file is switched to $COMMAND. If a RUN command is encountered, program execution is resumed; any remaining commands in the INCLUDE_FILE are not processed. When Debug again gains control, commands are read from the current Debug input file.

For example, assume that the permanent file $USER.INPUT_FILE2 contains the following Debug commands:

```
set_break break=bl line=5 module=fort
display_program_variable name=total
```

The following EXECUTE_TASK command is entered to turn on Debug mode and to execute the FORTRAN program illustrated in figure 3-1. After the Debug session begins, an INCLUDE_FILE command is entered and the commands in permanent file $USER.INPUT_FILE2 are immediately executed. When the commands in file $USER.INPUT_FILE2 are executed, the Debug prompt returns.

```
/execute_task file=lgo debug_mode=on
DEBUG 1.5
DB/include_file $user.input_file2
total=0.
DB/
```

The Debug command CHANGE_DEFAULTS can also change the Debug input file. The DEBUT_INPUT parameter of the CHANGE_DEFAULTS command can specify a new Debug input file to be used the next time Debug suspends program execution.

If a Debug session is initiated in an SCL procedure, Debug reads its input from the file COMMAND, instead of from the procedure file. To read the commands from the procedure, specify DEBUG_INPUT=$COMMAND.

Changing the Debug Output File

The DEBUG_OUTPUT parameter specifies the output file to which Debug writes its output (messages and displays). The default output file is $OUTPUT. $OUTPUT is the terminal for interactive jobs and the listing file for batch jobs. Initially, $OUTPUT is connected to the actual file OUTPUT. You can connect $OUTPUT to other files by using the SCL command CREATE_FILE_CONNECTION (see Recording an Interactive Debug Session described later in this chapter).

For example, the following EXECUTE_TASK command executes the FORTRAN program illustrated in figure 3-1. Debug mode is turned on and the Debug output file is specified as permanent file $USER.OUTPUT_FILE. During the Debug session, all Debug output is written to file $USER.OUTPUT_FILE.

```
/execute_task file=lgo debug_mode=on ..
../debug_output=$user.output_file
DB/set_break break=bl line=5 module=fort
DB/run
DB/display_program_value name=total
DB/quit
/
```

The contents of permanent file $USER.OUTPUT_FILE are:

```
DEBUG 1.5
 -- DEBUG: break B1, execution at M=FORT L=5
total=294.
 -- DEBUG: QUIT terminated task.
```

The Debug command CHANGE_DEFAULTS can also change the Debug output file. The DEBUT_OUTPUT parameter of the CHANGE_DEFAULTS command can specify a new Debug output file to be used as soon as the command is executed.

The OUTPUT parameter of the Debug display commands can be used to divert display output to another file; the diversion applies only to the command that contains the OUTPUT parameter.

You can also use the SCL DISPLAY_VALUE command (described in the SCL Language Definition Usage manual) to display values of SCL variables to your terminal.

Automatic Command Execution on Program Failure

Debug provides a feature through which you can create a special file, called an abort file, that contains Debug commands. If your program terminates because of an execution error, the commands in the abort file are automatically executed. The abort file is specified by the ABORT_FILE parameter on the SCL SET_PROGRAM_ATTRIBUTES, CREATE_PROGRAM_DESCRIPTION, or EXECUTE_TASK command.

The abort file feature is especially useful for maintaining programs that are in the working stage. It lets you specify a set of commands that is executed automatically when an error occurs. This can eliminate the need to reproduce the error in order to debug the program.

Because Debug is a command utility, you can include SCL commands as well as Debug commands in the abort file. This allows you to write sophisticated error handling procedures. (The SCL Language capabilities are described in the SCL Language Definition manual.)

By default, the abort file is $NULL, meaning that no abort file commands are executed. When executing, Debug mode must be off and the abort file must be a file other than $NULL for the commands in the abort file to be executed when a program fails; if ABORT_FILE=$NULL, the commands in the abort file are not executed.

To use the abort file feature, perform the following steps:

1. Create a file containing the sequence of Debug commands to be executed if the program fails.

2. Specify the file on the ABORT_FILE parameter on the SET_PROGRAM_ATTRIBUTES, CREATE_PROGRAM_DESCRIPTION, or EXECUTE_TASK command.

   If you want the output from your abort file to be written to an output file, specify the DEBUG_OUTPUT parameter also. (See Debug Input and Output in this chapter.)

3. Specify the parameter DEBUG_MODE=OFF when executing your program. (The abort file is not used during a Debug session.)

To demonstrate abort file use, suppose permanent file $USER.SAMPLE contains the COBOL source program listed in figure 3-2 and the permanent file $USER.ABORT_FILE contains the following Debug command:

```
display_calls display_options=(user_calls variable_values)
```

This command traces through the program's calls, and displays all of the variables known at each called location.

The following commands compile and execute the COBOL program listed in figure 3-2 using an abort file:

```
/cobol input=$user.sample list=list binary_object=lgo ..
../debug_aids=all optimization_level=debug
/set_program_attributes abort_file=$user.abort_file ..
../debug_output=$user.abort_file_errors
/execute_task file=lgo debug_mode=off
```

The SET_PROGRAM_ATTRIBUTES command turns off debug mode, specifies the ABORT_FILE, and specifies the DEBUG_OUTPUT. The EXECUTE_TASK command begins execution. When the error occurs, the commands in permanent file $USER.ABORT_FILE are executed.

The following message is displayed:

```
-- FATAL-- divide fault at P=0B 49 98.
```

A DIVIDE FAULT error occurs in program SAMPLE (figure 3-2).

The Debug session terminates after the commands in $USER.ABORT_FILE are executed.

```
SOURCE LIST OF SAMPLE

    1       Identification Division.
    2       Program-Id.  Sample.
    3       Environment Division.
    4       Data Division.
    5       Working-Storage Section.
    6       01  Numeric-Data.
    7           02  Item-A          Pic S999    Value -100.
    8           02  Item-B.
    9               03  Sub-Item-1  Pic 99      Value 10.
   10               03  Sub-Item-2  Pic 99      Value 25.
   11           02  Item-C.
   12               03  Sub-Item-1  Pic 99      Value 0.
   13               03  Sub-Item-2A Pic S9999   Value 0.
   14
   15       01  Character-Data-1.
   16           02  Char1           Pic X(10)   Value "ABCDEFGHIJ".
   17           02  Char2           Pic X(16)   Value "KLMNOPQRSTUVWXYZ".
   18           02  Char3.
   19               03  Char4       Pic X(4)    Occurs 4 times.
   20
   21       01  Numeric-Data-2.
   22           02  Sub1                Pic 99  Occurs 10 times.
   23           02  Sub2                Pic 99  Value 99.
   24
   25       77  Counter1                Pic 99  Value 0.
   26       77  Counter2                Pic 99  Value 0.
   27
   28       Procedure Division.
   29       Test1.
   30           Divide Item-A By Sub-Item-1 Of Item-C
   31               Giving Sub-Item-2A.
   32
   33       Test2.
   34           Move 1 To Counter1.
   35           Move 1 To Counter2.
   36               Perform 4 Times
   37                   Move Char2 (Counter1 : 4) To Char4 (Counter2)
   38                   Add 4 To Counter1
   39                   Add 1 To Counter2
   40               End-Perform.
   41
```

(Continued)

Figure 3-2. Debug Example: Source Listing of SAMPLE

(Continued)

```
      42      Test3.
      43          Move 1 To Counter1.
      44          Move 1 To Counter2.
      45              Perform 10 Times
      46                  Move Counter1 To Sub1 (Counter2)
      47                  Move Counter1 To Sub2
      48                  Call "Sample2" Using Counter1
      49                  Add 2 To Counter1
      50                  Add 1 To Counter2
      51              End-Perform.
      52      Stop Run.
      53      End Program Sample.

 SOURCE LIST OF SAMPLE2

      1       Identification Division.
      2       Program-Id. Sample2.
      3       Environment Division.
      4       Data Division.
      5       Working-Storage Section.
      6       77  Sum1                    Pic 999 Value 0.
      7
      8       Linkage Section.
      9       77  Count1                  Pic 99.
      10
      11      Procedure Division Using Count1.
      12      Start-Sample2.
      13          Add Count1 To Sum1.
      14          Display "SUM1 IS" Sum1.
      15          Exit Program.
```

Figure 3-2. Debug Example: Source Listing of SAMPLE

Figure 3-3 lists the contents of the permanent file $USER.ABORT_FILE_ERRORS (this is the file that contains Debug output). The DIVIDE FAULT error occurs at line 30 of program SAMPLE (figure 3-2). The traceback shows all the variables and their current values when the execution error is discovered. The DIVIDE FAULT on line 30 is caused because variable SUB-ITEM-1 of ITEM-C contains a value of zero.

```
DEBUG 1.5
-- DEBUG:  program terminated by calling
   abort at M=SAMPLE L=30 BO=26
-- DEBUG:  -- FATAL -- divide fault
   at P=0B 49 98.
-- Traceback from procedure SAMPLE module SAMPLE at line 30 byte
   offset 26
-- DISPLAY OF ALL VARIABLES IN SAMPLE

NAME                DATA_TYPE    SIZE_BYTES    VALUE

NUMERIC-DATA        ALPHNUM           13
     ITEM-A         DISPLAY           3        -100
     ITEM-B         ALPHNUM           4
     SUB-ITEM-1     DISPLAY           2        +10
     SUB-ITEM-2     DISPLAY           2        +25
     ITEM-C         ALPHNUM           6
     SUB-ITEM-1     DISPLAY           2        +0
     SUB-ITEM-2A    DISPLAY           4        +0

CHARACTER-DATA-1    ALPHNUM           42
     CHAR1          ALPHNUM           10       ABCDEFGHIJ
     CHAR2          ALPHNUM           16       KLMNOPQRSTUVWXYZ
     CHAR3          ALPHNUM           16
     CHAR4 ( 1)     ALPHNUM           4        ???? ( 4
                                               OCCURRENCES)

NUMERIC-DATA-2      ALPHNUM           22
     SUB1 ( 1)      DISPLAY           2        INVALID_BDP_
                                               DATA ( 10
                                               OCCURRENCES)
     SUB2           DISPLAY           2        +99

COUNTER1            DISPLAY           2        +0

COUNTER2            DISPLAY           2        +0
```

Figure 3-3. Contents of Permanent File $USER.ABORT_FILE_ERRORS

Recording an Interactive Debug Session

In an interactive Debug session, generally you enter input to Debug and receive output from Debug at your terminal. Because input and output are communicated through the terminal, all information about the Debug session is lost when you terminate a Debug session.

You can, however, record the entire session by using the CREATE_FILE_CONNECTION command to connect files with your terminal before you start Debug. This feature is available only when debugging in line mode. The terminal is associated with certain standard system files by default. (A file connection is a connection between one of the system standard files and an actual file. The effect of a file connection is that any data access request against a standard file is passed on to the connected file. See the SCL System Interface Usage manual for detailed descriptions on system standard files and the CREATE_FILE_CONNECTION command.)

For example, by connecting the standard files $OUTPUT, $ECHO, $RESPONSE, and $ERRORS, you can record the entire Debug session on the same file.

```
/create_file_connection $output output_file
/create_file_connection $response output_file
/create_file_connection $echo output_file
```

Screen Mode Debugging 4

This chapter describes the features and functions used in screen mode debugging.

Screen Layout 4-2

Debugging Using Functions 4-4

Entering Commands on the Home Line 4-5

The Visible Windows in Screen Mode Debugging 4-6
- The Source Window 4-6
  - The Zoom-in Display 4-6
  - The Zoom-out Display 4-8
  - The Trace Display 4-10
- The Output Window 4-10
- The Help Window 4-10
- The Keys Window 4-12
- The Options Window 4-13

Function Descriptions 4-14
- Summary of Screen Mode Functions 4-14
- Getting Help (HELP) 4-16
- Changing the Window Displays 4-16
  - Display the Source Code (ZMIN) 4-16
  - Display Source Modules (ZMOUT) 4-17
  - Display All or Specified Source Calls (TRACE) 4-18
  - Display the Screen Mode Debug Function Names (KEYS).. 4-18
- Interrupting Program Execution 4-19
  - Stop Execution at Specified Line (SETBRK) 4-19
  - Delete Break in Program Execution (DELBRK) 4-21
- Beginning or Resuming Program Execution 4-22
  - Execute to Program Termination, Break, or Event (HSPEED) 4-22
- Execute to Next Procedure Call (MSPEED) 4-25
  - Execute to Next Paragraph (COBOL only) (LSPEED) 4-25
  - Execute One Line at a Time (STEP1) 4-26
  - Execute Several Lines at a Time (STEPN) 4-26
  - Change Location of Program Execution (GOTO) 4-27
- Displaying and Changing Program Values 4-30
  - Display Values of Program Variables (SEEVAL) 4-30
  - Change Values of Program Variables (CHAVAL) 4-32

Locating Information in a Window 4-34
Restore Source Window to Previous View (BACK) 4-34
Display the First Screen of Information in the Window (FIRST) 4-34
Display the Last Screen of Information in the Window (LAST) 4-34
Perform Window-Dependent Action (NEXT) 4-35
Move to the Home Line (HOME) 4-35
Move Cursor Position Line to Bottom of Window (DOWN). 4-35
Move Cursor Position Line to Top of Window (UP) 4-35
Move One Page Back From Cursor Position (BKW) 4-36
Move One Page Forward From Cursor Position (FWD) 4-36
Prompt for Text String Then Search Forward (LOCATE).. 4-36

Tailoring a Screen Debug Session 4-38
Refresh the Screen (REFRSH) 4-38
Dividing the Screen Windows (SPLIT) 4-38
Align Source Column With Leftmost Window Column (LEFT) 4-38
Align Leftmost Window Column With Source Column (RIGHT) 4-39
Compress Characters (NARROW) 4-39
Enlarge Compressed Characters (WIDE) 4-39
Viewing Screen Options (OPTS) 4-40
Ending the Screen Mode Debug Session 4-41
Terminate the Debug Session (QUIT) 4-41
End Screen Mode Debug and Switch to Line Mode Debug (DEAS) 4-41

Screen Mode Debugging 4

Screen mode Debug gives you all the Debug features with the ease of a full screen interface. Screen mode Debug is easy to use. You can see your source program displayed on the screen as you debug it and you can execute Debug commands by using function keys instead of typing the command. Screen mode Debug also provides various window displays to further enhance your debugging capabilities. You can learn screen mode Debug as you use it by using the HELP feature.

Using Debug in screen mode requires that your terminal support full screen operation. See chapter 2 for information about terminal definitions that can support full screen interface.

Throughout this chapter, examples and explanations apply to the Viking 721 terminal.

Screen Layout

The screen layout and function key assignments for screen mode Debug may differ slightly depending on the type of terminal you are using. However, in general, the format of a Debug screen is:

```
①
②
   Debugging SAMPLE
   -->     Identification Division.
           Program-Id.      Sample.
           Environment Division.
           Data Division.
③          Working-Storage Section.
           01   numeric-data.
                02  item-a
                02  sub-item-1          pic S999        value -100.
                02  sub-item-2          pic 99          value 0.
           Procedure Division.

----------------------------------- OUTPUT -----------------------------------
                      -- Welcome to Full Screen Debugging --
④

                            Press HELP for Assistance

     | StepN |   | LSpeed |   | Locate |   | ChaVal |   | DelBrk |   | Deas |   | ZmOut |   | Keys |
⑤ f1 | Step1 | f2| MSpeed | f3| HSpeed | f4| SeeVal | f5| SetBrk | f6| Quit | f7| Trace | f8| Goto |
```

The screen divisions are described as follows:

① Home line	The line on which you enter Debug commands and SCL commands. For some terminal types, the home line is at the bottom of the screen.
② Response line	The line on which short responses and advisory messages from Debug are displayed. If the home line is at the bottom, the response line is the line above it.
③ Source window	The area in which the source listing of the program you are debugging is displayed.
④ Output window	The area in which output generated by your program (or output delivered by Debug) is displayed.
⑤ Functions	Screen mode functions currently assigned to function keys. These functions can also be entered on the home line.

The relative size of the divisions of the screen depends on the size of your terminal screen. You can change the relative divisions of the screen; refer to the OPTS and the SPLIT functions discussed later in this chapter.

Debugging Using Functions

Function keys are the primary method for interacting with screen mode Debug. The function keys are displayed at the bottom of the screen and each function key is labeled with the name of the function currently assigned to it.

The top label on each function key names the function performed by the shifted function key; the bottom label names the function performed by the unshifted function key. The action of the function is performed on the line where the cursor is positioned before the function key is pressed. Function names are meant to suggest their purpose. However, by pressing the HELP function (described in chapter 3 and also later in this chapter), you can get a more detailed explanation of the purpose of each function.

For example, the following shows the function key display at the bottom of the Debug screen using the Viking 721 terminal definition:

	f1	f2	f3	f4	f5	f6	f7	f8
Shifted	StepN	LSpeed	Locate	ChaVal	DelBrk	Deas	ZmOut	Keys
Unshifted	Step1	MSpeed	HSpeed	SeeVal	SetBrk	Quit	Trace	Goto

Screen mode Debug requires that your terminal have at least 16 function keys to which functions can be assigned. Debug offers more than 16 functions, therefore, some functions are not assigned to function keys. However, the function label is always an alias for the corresponding home line command. Thus, any function can be entered on the home line by its six character short name.

If function keys are available, the following functions have highest priority and are always assigned to function keys:

HELP (request help)
QUIT (quit)
FWD (move forward)
BKW (move backward)

Other functions which are considered important for debugging are given priority and assigned to the remaining function keys.

The set of functions available and displayed on the screen may change during a debugging session depending upon the language of the program being debugged, the optimization level at which it was compiled, whether or not the program has completed execution, and the displays and windows currently selected. You can select to see zero, one, or two rows of function keys at the bottom of your screen. (Refer to the OPTS function described later in this chapter.) The default number of rows is one.

Entering Commands on the Home Line

Most of the basic Debug functions can be done using function keys. However, there will be times when the function keys provided do not meet your debugging needs. It is then that you will have to enter a command on the home line. Debug functions, Debug commands, and SCL commands can be entered on the home line.

Usually, the home line is the top line of the screen; on some terminals it is the bottom line. Pressing the HOME key (specified as either a function key or as a special key on your keyboard) moves the cursor to the left edge of the home line where you type in a function or command. The function or command is processed when you press the RETURN key.

When you enter a function on the home line, the action of the function is performed on the line at which the cursor was positioned when you pressed the HOME key. The following functions cannot be entered on the home line:

NEXT
HELP

If a Debug command is entered on the home line, the command is processed as if the function key had been pressed. The following Debug commands cannot be entered on the home line:

ACTIVATE_SCREEN or ACTS
CHANGE_DEFAULTS or CHAD
DELETE_BREAK or DELB
RUN
SET_BREAK or SETB
SET_STEP_MODE or SETSM

If an SCL command is entered on the home line, the command is forwarded to the appropriate processor.

The Visible Windows in Screen Mode Debugging

Most of the visible screen is occupied by displays that you have selected for the Source and Output windows. By default, both the Source and Output windows are shown, but you can elect to change the relative size of these two windows. (Refer to the OPTS and the SPLIT functions described later in this chapter.) Several other windows can be selected to temporarily overlay the Source and Output windows to display special information; they are the Help window, Keys window, and Options window.

The Source Window

The Source window shows you information about your program as it executes. By default, the Source window occupies the top three-fourths portion of the screen (except for the home line and the response line). You can select the Zoom-in, Zoom-out, or Trace display for viewing in the Source window.

The Zoom-in Display

The Zoom-in display lets you view the source code of the module in the program being debugged. This is the display in which you debug your program modules and it is titled with the name of the module currently displayed. Modules can be programs, subprograms, procedures, modules, or functions depending on the computer language of the module being debugged. An arrow indicates where execution is to resume and lines containing breaks are highlighted.

You can select the Zoom-in display by pressing the ZMIN function (described later in this chapter). The Zoom-in display is also automatically selected when execution of your program stops and the source code of the module is available.

If the source code for a specified module cannot be found (the file no longer exists or the module was compiled without line number tables), you cannot view the source code. In this case, the Zoom-out display is shown with an arrow pointing to the module.

The following is an example of a Zoom-in display of the source of a FORTRAN module:

```
Debugging FORT
-->   program fort
      real average, total
      total=0.
      call aver(total)
      average=total/5.
      print*,'average = ',average
      end
      subroutine aver(x)
      real scores(5)
      data scores /50., 56., 98., 12., 78./
      do 10 i=1,5
 10   x=scores(i)+x
      end
------------------------------ OUTPUT ------------------------------
             -- Welcome to Full Screen Debugging --

                   Press HELP for Assistance

   StepN          Locate  ChaVal  DelBrk   Deas  ZmOut   Keys
f1 Stepl f2 MSpeed f3 HSpeed f4 SeeVal f5 SetBrk f6 Quit f7 Trace f8 Goto
```

The Zoom-out Display

The Zoom-out display shows an overview of the task being executed, that is, all modules in the program are displayed. For example, Program and Subprogram statements are displayed for FORTRAN programs; Program, Section, and Paragraph names are displayed for COBOL programs; Program, Procedure, and Function names are displayed for Pascal programs; Module, Program, Function, and Procedure names are displayed for CYBIL programs; and functions and local blocks that contain variables are displayed for C programs. An arrow points to the component where execution has stopped.

You can select the Zoom-out display by pressing the ZMOUT function. The Zoom-out display is automatically displayed when execution of your task stops and the source needed is not available. You are prompted to enter the location of the source when it is not found.

In the following example, a Zoom-out display shows a list of modules for a FORTRAN program. Note that the functions that are available depend on the display shown and the computer language in which the modules are written.

```
Displaying Routines
--> PROGRAM FORT
    SUBROUTINE AVER




------------------------------ OUTPUT ------------------------------
              -- Welcome to Full Screen Debugging --

                    Press HELP for Assistance

   StepN            Locate      DelBrk    Deas     ZmIn     Keys
f1 Step1  f2 MSpeed f3 HSpeed f4  f5 SetBrk f6 Quit f7 Trace f8
```

The following example is of a Zoom-out display showing a list of modules of a COBOL program.

```
Displaying Routines
--> PROGRAM-ID  SAMPLE
        TEST1
        TEST2
        TEST3
    PROGRAM-ID  SQUARING-PROCEDURE
        START-SQUARING-PROCEDURE

------------------------------- OUTPUT --------------------------------
              -- Welcome to Full Screen Debugging --

                    Press HELP for Assistance

   StepN     LSpeed    Locate            DelBrk    Deas      ZmIn      Keys
f1 Step1  f2 MSpeed f3 HSpeed f4      f5 SetBrk f6 Quit   f7 Trace  f8
```

The modules, SAMPLE and SQUARING-PROCEDURE, are Program names and the modules, TEST1, TEST2, TEST3, and START-SQUARING-PROCEDURE, are Paragraph names.

The Trace Display

The Trace display shows the routines in the active call chain. The traceback begins with the routine that was executing when Debug gained control, and proceeds through the sequence of called routines until the main program is reached. For each routine in the traceback, TRACE displays the routine name and module name from which the routine was called.

You can select the Trace display by pressing the TRACE function. Once the Trace display is selected, you can press the ZMIN function to see the source line for the call indicated by the cursor.

The Output Window

The Output window displays the output generated by your program and the output delivered by Debug. By default, the Output window occupies the bottom one-fourth portion of the screen (except for the function keys). You can alter the size using the OPTS and SPLIT functions (described later in this chapter).

The Help Window

The Help window temporarily overlays a portion of the visible screen and displays help information for each function. You can select the Help window by pressing the HELP function (described in chapter 3 and later in this chapter). When the Help window is displayed, you are prompted for the function for which you need help. When you press a function key, a short description of the function you select is displayed in the Help window. By pressing the other function keys, information about their use is displayed.

In the following example, the SEEVAL function is described in the Help window:

```
Debugging FORT
-->   program fort
      real average, total
      total=0.
      call aver(total)

                              HELP
       See value will display the current value of a variable
                      in the output window.

                 Press NEXT to return to debugging

------------------------------ OUTPUT ------------------------------
             -- Welcome to Full Screen Debugging --

                    Press HELP for Assistance

    StepN             Locate   ChaVal   DelBrk   Deas     ZmOut    Keys
 f1 Stepl  f2 MSpeed f3 HSpeed f4 SeeVal f5 SetBrk f6 Quit f7 Trace f8 Goto
```

The Keys Window

The Keys window temporarily overlays a portion of the visible screen listing the available functions. You can select the Keys window by pressing the KEYS function (described later in this chapter). The Keys window is useful when your terminal screen does not display the function keys at the bottom of the screen or if your terminal does not have enough function keys for all the available functions.

In the following example, the Keys window is displayed:

```
Debugging FORT
-->   program fort

   Screen functions and Debug commands may be entered on the home line.
                       Available functions are:

          Back      Bkw       ChaVal    Deas      DelBrk
          Down      First     Fwd       Goto      HSpeed
          Help      Home      Keys      Last      Locate
          MSpeed    Narrow    Opts      Quit      Refrsh
          SeeVal    SetBrk    Split     Stepl     StepN
          Trace     Up        ZmOut

------------------------------- OUTPUT -------------------------------
              -- Welcome to Full Screen Debugging --

                     Press HELP for Assistance

   StepN           Locate   ChaVal   DelBrk   Deas     ZmOut     Keys
f1 Stepl  f2 MSpeed f3 HSpeed f4 SeeVal f5 SetBrk f6 Quit f7 Trace f8 Goto
```

The Options Window

The Options window temporarily overlays a portion of the visible screen. The Options window displays a list of options you can select to tailor your debugging session. You can select the Options window by pressing the OPTS function (described later in this chapter).

In the example below, the Options Window is displayed:

```
Debugging FORT
-->     program fort              DEBUG OPTIONS WINDOW
        real average, total
        total=0.
        call aver(total)          FEATURE (menu item) | VALUE | AVAILABLE OPTIONS
        average=total/5.
        print*,'average = ',a          MENU ROWS (F1) |   1   | 0, 1, 2
        end                                 MODE (F2) | USER  | USER, SYSTEM
        subroutine aver(x)           SOURCE SIZE (F3) |  18   | 1 .. 23
        real scores(5)               OUTPUT SIZE (F4) |   6   | 1 .. 23
        data scores /50.,56.,
        do 10 i=1,5                Press matching menu item to toggle value.
 10     x=scores(i)+x                  Press NEXT to return to debugging.
        end
------------------------------------ OUTPUT ------------------------------------
                   -- Welcome to Full Screen Debugging --

                         Press HELP for Assistance

   StepN          Locate   ChaVal   DelBrk   Deas    ZmOut    Keys
f1 Stepl  f2 MSpeed f3 HSpeed f4 SeeVal f5 SetBrk f6 Quit f7 Trace f8 Goto
```

Function Descriptions

The functions in this section are grouped by type of function for clarity.

Summary of Screen Mode Functions

Function	Description	Page
BACK	Restores the Source window to the previous view.	4-34
BKW	Moves one page back from the cursor position.	4-36
CHAVAL	Changes the values of program variables.	4-32
DEAS	Switches from screen mode Debug to line mode Debug.	4-40
DELBRK	Deletes a break.	4-21
DOWN	Moves the current line to the bottom of the window.	4-35
FIRST	Displays the first screen of information in the window.	4-34
FWD	Moves one page forward from the cursor position.	4-36
GOTO	Changes the location of program execution.	4-27
HELP	Displays help information.	4-16
HOME	Moves to the home line.	4-35
HSPEED	Executes to program termination, break, or event.	4-22
KEYS	Displays the screen mode Debug functions.	4-18
LAST	Displays the last screen of information in the window.	4-34
LEFT	Aligns the current column to the left column of the window.	4-38
LOCATE	Searches for a text string.	4-37

Function	Description	Page
LSPEED	Executes to the next paragraph (COBOL only).	4-25
MSPEED	Executes to the next procedure call or section.	4-25
NARROW	Displays maximum number of characters per line.	4-39
NEXT	Performs a window-dependent action.	4-35
OPTS	Changes the screen options.	4-39
QUIT	Terminates the Debug session.	4-40
REFRSH	Rewrites the screen.	4-38
RIGHT	Aligns the left column of the window to the current column.	4-39
SEEVAL	Displays values of program variables.	4-30
SETBRK	Stops execution at a specified line.	4-19
SPLIT	Divides the screen windows.	4-38
STEP1	Executes one line at a time.	4-26
STEPN	Executes several lines at a time.	4-26
TRACE	Displays all or specified calls.	4-18
UP	Moves current line to the top of the window.	4-35
WIDE	Displays minimum number of characters per line.	4-39
ZMIN	Displays the source code of your program.	4-16
ZMOUT	Display the modules of your source program.	4-17

Getting Help (HELP)

Purpose: Displays the Help window. The Help window displays the function descriptions.

Function: HELP

Remarks:

- The Help window overlays a portion of your screen and you are prompted to enter the function for which you need help. A short description of the function is displayed in the Help window.
- HELP is a top priority function and is always assigned to a function key, if function keys are available. If, however, you enter the HELP function on the home line, the SCL online manual is displayed to the screen rather than the Help window. To return to debugging, press quit.

Changing the Window Displays

To help you debug your program, various window displays can be selected to show particular information about the debugging session. The ZMIN, ZMOUT, TRACE, and KEYS functions let you temporarily change the viewing displays in the Source and Output windows.

Display the Source Code (ZMIN)

Purpose: Selects the Zoom-in display. The Zoom-in display shows the source code of the module in the program being debugged.

Function: ZMIN

Remarks:

- The Zoom-in display is the display in which you debug your program modules.
- An arrow indicates where execution is currently stopped. Lines containing breaks are highlighted.
- The window is titled with the name of the module currently displayed.

- You access the Zoom-in display by placing the cursor on a module name while in the Zoom-out display or Trace display and press 'ZMIN'. The Zoom-in display is automatically selected when execution of your program stops, unless execution stops on a procedure or function and the Zoom-out or Trace display is selected.

- If the source for a specified module cannot be found (that is, the file no longer exists or the module was compiled without line tables), you cannot view the source. In this case, an informative message is displayed.

Display Source Modules (ZMOUT)

Purpose: Selects the Zoom-out display. The Zoom-out display lists the modules in the program you are debugging.

Function: ZMOUT

Remarks:

- If the source of a module is not needed or not available for the task, only the module name is shown.

- Breaks can be set or deleted on modules if the source for the module has been initialized (that is, the Zoom-in display has been previously selected at least once for the module). Breaks that are set are highlighted (see the SETBRK and DELBRK functions described later in this chapter).

- When the Zoom-out display is selected for the Source window, it is automatically displayed when execution of the task stops or when the source needed is not available. You are prompted to enter the location of the source when it is not found.

- You can move to the Zoom-out display from the Zoom-in display by pressing ZMOUT. You can view and debug the source code for a module by moving the cursor to the desired module and press ZMIN.

Display All or Specified Source Calls (TRACE)

Purpose: Selects the Trace display. The Trace display shows the routines in the active call chain.

Function: TRACE

Remarks:

- The traceback begins with the routine that was executing when Debug gained control, and proceeds through the sequence of called routines until the main program is reached. For each routine in the traceback, TRACE displays the routine name and the module name (if different) from which the routine was called.

- For calls from procedures in languages that support nested procedures, both the module and procedure name are shown in the Trace display. Otherwise, only the module name is shown. If there are no active calls, a message is issued and the current contents of the display are not changed.

- Once the Trace display is selected, you can press the ZMIN function to see the source line for the call indicated by the cursor. If the location of the source is not known (it was not passed to Debug by the compiler), you are prompted to give the file name where the source can be found.

- The speed functions (HSPEED, MSPEED, or LSPEED) or the Step functions (STEP1 or STEPN) are available to resume execution of your program. When execution stops for any reason, the Trace display is replaced by the Zoom-in or Zoom-out display as appropriate, except when MSpeed is selected from the Trace display; the Trace display remains in view (see Zoom-in or Zoom-out displays described in this chapter).

Display the Screen Mode Debug Function Names (KEYS)

Purpose: Displays a window containing a list of available screen mode Debug functions.

Function: KEYS

Remarks:

- The Keys window overlays a portion of the visible screen listing the available functions mnemonically.

- The Keys window is useful when your function keys are not displayed on the screen or if your terminal does not have enough function keys for all the available functions.

- To restore the previously selected window and continue debugging, press the RETURN key.

Interrupting Program Execution

The primary method of suspending program execution during a Debug session is to set breaks at selected points in your program. When execution reaches a statement where a break is set, execution temporarily stops before the statement is executed. You can set breaks before execution begins or whenever execution has stopped. The SETBRK and DELBRK functions set and delete breaks on lines in the source code of your program. Lines with breaks are highlighted to inform you that the break is set.

Stop Execution at Specified Line (SETBRK)

Purpose: Sets a break in a program. The break is set on the line that contains the cursor.

Function: SETBRK

Remarks:

- Execution is interrupted before the line containing the break is executed.
- You can set a break in a program by placing the cursor on the program line where you want the break and then press SETBRK. The line is highlighted, indicating that a break is set.
- Breaks can be set on executable statements only (except for BASIC programs where a break can be set on every line).
- The maximum number of breaks you can set is 32.

For Better Performance

When a break is no longer needed, you should delete it (see DELBRK described in this chapter). Deleting unnecessary breaks improves the performance of Debug; programs execute slower when breaks remain set.

Example: In the display below, the cursor is placed on the line:

```
result := counter * counter
```

```
Debugging SQUARING PROCEDURE in EXAMPLE PAS
-->        result : integer;
        BEGIN
           result := counter * counter;
           writeln(' ',counter:2,' times ',counter:2,' = ',result:4);
        END;
PROCEDURE test1;
   VAR
     counter : integer
   BEGIN test1
     FOR counter := 1 TO 10 DO
        squaring_procedure (counter);
   END; test
------------------------------- OUTPUT -------------------------------
                 -- Welcome to Full Screen Debugging --

                        Press HELP for Assistance
 1 times 1 =       1
    | StepN |  |        |  | Locate |  | ChaVal |  | DelBrk |  | Deas |  | ZmOut |  | Keys |
  f1| Step1 |f2| MSpeed |f3| HSpeed |f4| SeeVal |f5| SetBrk |f6| Quit |f7| Trace |f8| Goto |
```

When the SETBRK function is pressed, the line containing the cursor is highlighted to show that a break has been set on the line:

```
Debugging SQUARING PROCEDURE in EXAMPLE PAS
-->         result : integer;
         BEGIN
            result := counter * counter;
            writeln(' ',counter:2,' times ',counter:2,' = ',result:4);
         END;
PROCEDURE test1;
   VAR
     counter : integer
   BEGIN test1
     FOR counter := 1 TO 10 DO
        squaring_procedure (counter);
   END; test
------------------------------ OUTPUT ------------------------------
                -- Welcome to Full Screen Debugging --

                       Press HELP for Assistance
1 times 1 =       1
   StepN         Locate   ChaVal   DelBrk   Deas    ZmOut    Keys
f1 Step1 f2 MSpeed f3 HSpeed f4 SeeVal f5 SetBrk f6 Quit f7 Trace f8 Goto
```

Delete Break in Program Execution (DELBRK)

Purpose: Deletes a break in a program. The break is deleted from the line that contains the cursor.

Function: DELBRK

Remarks: You can delete a break by placing the cursor on the line with a break and then press DELBRK. The highlight is removed from the line.

For Better Performance

Deleting unnecessary breaks improves the performance of Debug; programs execute slower when breaks remain set.

Beginning or Resuming Program Execution

The HSPEED, MSPEED, LSPEED, STEP1, and STEPN functions begin execution of your program or resume execution after encountering a break. Execution continues until either another break occurs, an execution error occurs, or the program runs to completion. Also, you can change the location of where execution will resume next with the GOTO function. These functions are available regardless of the displays visible in the Source and Output windows.

Execute to Program Termination, Break, or Event (HSPEED)

Purpose: Resumes execution of your program.

Function: HSPEED

Remarks:

- Execution continues uninterrupted until your program terminates, a break is encountered, or an unselected event (such as a divide fault, a terminate break, or a pause break) occurs.
- An arrow points to the source line in your program where execution has stopped and is to resume next time.
- The message "Program Terminated" is issued when your program terminates.

Example: Suppose the screen below is displayed. The highlighted line indicates that a break is set on that line. The arrow points to the line where execution has currently stopped and is to resume next.

```
Debugging FORT
-->     program fort
        real average, total
        total=0.
        call aver(total)
        average=total/5.
        print*,'average = ',average
        end
        subroutine aver(x)
        real scores(5)
        data scores /50., 56., 98., 12., 78./
        do 10 i=1,5
   10   x=scores(i)+x
        end
--------------------------- OUTPUT ---------------------------
            -- Welcome to Full Screen Debugging --


                 Press HELP for Assistance

   |StepN|    |      |   |Locate|   |ChaVal|   |DelBrk|   |Deas|   |ZmOut|   |Keys|
 f1|Stepl| f2|MSpeed| f3|HSpeed| f4|SeeVal| f5|SetBrk| f6|Quit| f7|Trace| f8|Goto|
```

Beginning or Resuming Program Execution

When the HSPEED function is pressed, execution begins with the line containing the arrow. The message, hspeed, is displayed in the top right-hand corner of the screen. When execution reaches the line containing the break, execution stops (the arrow now points to this line).

```
Debugging FORT
          program fort
          real average, total
          total=0.
          call aver(total)
          average=total/5.
          print*,'average = ',average
          end
          subroutine aver(x)
          real scores(5)
          data scores /50., 56., 98., 12., 78./
          do 10 i=1,5
   -->    x=scores(i)+x
          end
------------------------ OUTPUT ------------------------
              -- Welcome to Full Screen Debugging --

                     Press HELP for Assistance

   StepN        Locate  ChaVal  DelBrk   Deas    ZmOut   Keys
f1 Stepl f2 MSpeed f3 HSpeed f4 SeeVal f5 SetBrk f6 Quit f7 Trace f8 Goto
```

Example: Suppose the screen below is displayed. The highlighted line indicates that a break is set on that line. The arrow points to the line where execution has currently stopped and is to resume next.

```
Debugging FORT
-->     program fort
        real average, total
        total=0.
        call aver(total)
        average=total/5.
        print*,'average = ',average
        end
        subroutine aver(x)
        real scores(5)
        data scores /50., 56., 98., 12., 78./
        do 10 i=1,5
   10   x=scores(i)+x
        end
---------------------------- OUTPUT ----------------------------
               -- Welcome to Full Screen Debugging --

                    Press HELP for Assistance

   StepN         Locate   ChaVal   DelBrk   Deas     ZmOut    Keys
f1 Step1  f2 MSpeed f3 HSpeed f4 SeeVal f5 SetBrk f6 Quit f7 Trace f8 Goto
```

When the HSPEED function is pressed, execution begins with the line containing the arrow. The message, hspeed, is displayed in the top right-hand corner of the screen. When execution reaches the line containing the break, execution stops (the arrow now points to this line).

```
Debugging FORT
          program fort
          real average, total
          total=0.
          call aver(total)
          average=total/5.
          print*,'average = ',average
          end
          subroutine aver(x)
          real scores(5)
          data scores /50., 56., 98., 12., 78./
          do 10 i=1,5
    -->   x=scores(i)+x
          end
------------------------------ OUTPUT ------------------------------
              -- Welcome to Full Screen Debugging --

                   Press HELP for Assistance

   StepN           Locate   ChaVal   DelBrk   Deas    ZmOut    Keys
f1 Stepl f2 MSpeed f3 HSpeed f4 SeeVal f5 SetBrk f6 Quit f7 Trace f8 Goto
```

Execute to Next Procedure Call (MSPEED)

Purpose: Executes your program until the next subroutine, procedure, or section is called.

Function: MSPEED

Remarks:

- An arrow points to the source line in your program where execution has stopped.
- MSPEED can be interrupted by a break, program termination, or an unselected event (such as a divide fault, a terminate break, or a pause break).
- MSPEED is available only for modules compiled with optimization level set to Debug. (See Chapter 2 for more information about optimization levels.)
- MSPEED is not supported by the C language.

Execute to Next Paragraph (COBOL only) (LSPEED)

Purpose: Executes your program until the next paragraph is called.

Function: LSPEED

Remarks:

- An arrow points to the source line in your program where execution has stopped.
- LSPEED can be interrupted by a break, program termination, or an unselected event (such as a divide fault, a terminate break, or a pause break).
- LSPEED is available only for modules compiled with optimization level set to Debug. (See Chapter 2 for more information about optimization levels.)
- LSPEED is supported by the COBOL language only.

Execute One Line at a Time (STEP1)

Purpose: Executes the current line of the program. The line indicator is then increased by 1 and points to the next line in the program.

Function: STEP1

Remarks:

- STEP1 is available only for modules compiled with optimization level set to Debug. (See Chapter 2 for more information about optimization levels.)
- If executing the line produces any output, the output appears in the Output window.
- STEP1 is not supported by the C language.

Execute Several Lines at a Time (STEPN)

Purpose: Beginning with the current line, STEPN executes N number of lines.

Function: STEPN

Remarks:

- You are prompted for the value of N when STEPN is pressed.
- STEPN is available only for modules compiled with optimization level set to Debug. (See Chapter 2 for more information about optimization levels.)
- If executing the lines produces any output, the output appears in the Output window.
- STEPN is not supported by the C language.

Change Location of Program Execution (GOTO)

Purpose: GOTO changes the location where execution of your program begins next when another of the Speed functions (HSPEED, MSPEED, or LSPEED) or the Step functions (STEP1 or STEPN) is pressed.

Function: GOTO

Remarks:

- Before pressing GOTO, you position the cursor on the line where you want execution to resume, then press GOTO. This line must be in the same procedure where execution is currently stopped.
- GOTO is available in the Zoom-in display only.
- GOTO is a very powerful function. Use it with care. Unexpected results can occur since code that is skipped may have a significant effect on the execution of the entire program. If, for example, you use GOTO to transfer into a loop construct, the control variable for the loop may not be initialized and unexpected output could result. GOTO is intended to be used when a bug has been isolated and can be fixed with CHAVAL. You can test the change without recompiling your program by resuming execution of a portion of code with the new data values. GOTO can also be used to skip a section of faulty code.

Example: In the example below, execution has stopped on line:

divide dividend by divisor giving quotient.

because the value of DIVISOR is zero resulting in a DIVIDE FAULT error. The message, divide fault, is displayed in the top right hand corner of the screen.

```
                                                          divide fault
Debugging EXAMPLE COB
       Paragraph3.
-->        divide dividend by divisor giving quotient.
           display "ANSWER IS:  " ANSWER.
           stop run.
       end program example-cob.

-------------------------------- OUTPUT --------------------------------
                 -- Welcome to Full Screen Debugging --

                        Press HELP for Assistance

  |StepN|   |LSpeed|   |Locate|   |ChaVal|   |DelBrk|   |Deas|   |ZmOut|   |Keys|
f1|Step1| f2|MSpeed| f3|HSpeed| f4|SeeVal| f5|SetBrk| f6|Quit| f7|Trace| f8|Goto|
```

Using the CHAVAL function, you can change the value of DIVISOR temporarily and then resume execution using the new value for DIVISOR. Once the value of DIVISOR is changed, using the up-arrow key, the cursor is moved to the line

    Paragraph3.

and then GOTO is pressed. The arrow now points to this line and program execution will resume with this statement using the new value given to DIVISOR. In this example, the GOTO and CHAVAL functions are used in concert to recover from execution errors. However, to correct execution errors permanently, you must exit Debug, edit the program, and recompile it.

```
Debugging EXAMPLE COB
-->     Paragraph3.
            divide dividend by divisor giving quotient.
            display "ANSWER IS:  " ANSWER.
            stop run.
        end program example-cob.

------------------------------- OUTPUT -------------------------------
              -- Welcome to Full Screen Debugging --

                     Press HELP for Assistance

   StepN     LSpeed   Locate    ChaVal    DelBrk    Deas     ZmOut     Keys
f1 Step1  f2 MSpeed f3 HSpeed f4 SeeVal f5 SetBrk f6 Quit  f7 Trace  f8 Goto
```

Displaying and Changing Program Values

With Debug, you can examine and change the contents of variables. Using the SEEVAL and CHAVAL functions, you can view and change the values of program variables and arrays while execution of your program is suspended. The values are displayed as they are written in your source program. Replacement values for variables are entered in the same format as they are defined in your program.

Display Values of Program Variables (SEEVAL)

Purpose: Displays the current values of program variables.

Function: SEEVAL

Remarks:

- When you press SEEVAL, you are prompted to enter the variable name or names whose values you wish to see displayed.
- You can display a list of variables by separating the variable names with commas or spaces.
- Variables are assumed to exist in the module (and procedure if appropriate) in which the cursor is located.
- When prompted for a variable name, you can enter $ALL to see all of the variables in the current module or procedure.
- See the description of the DISPLAY_PROGRAM_VALUE line mode command in chapter 5 for a list of variable types that can be displayed with the SEEVAL function.

Example: In the example below, the SEEVAL function is used to view the value of variable X. When the prompt

    Enter name(s) to see

is displayed in the top right hand corner, the variable X is entered on the home line. The value of variable X is displayed in the Output window.

```
x
                                                        Enter name(s) to see
 Debugging FORT
        program fort
        real average, total
        total=0.
        call aver(total)
        average=total/5.
        print*,'average = ',average
        end
        subroutine aver(x)
        real scores(5)
        data scores /50., 56., 98., 12., 78./
        do 10 i=1,5
 -->    x=scores(i)+x
        end
---------------------------------- OUTPUT ----------------------------------
                  -- Welcome to Full Screen Debugging --


                         Press HELP for Assistance
x = 204.
   StepN         Locate  ChaVal  DelBrk  Deas     ZmOut   Keys
f1 Stepl f2 MSpeed f3 HSpeed f4 SeeVal f5 SetBrk f6 Quit f7 Trace f8 Goto
```

Change Values of Program Variables (CHAVAL)

Purpose: Changes the value of program variables.

Function: CHAVAL

Remarks:

- When you press CHAVAL, you are prompted to enter the variable name and its new value (variable=value).
- The variable must be known in the module or procedure in which the cursor is located.
- See the description of the CHANGE_PROGRAM_VALUE line mode command in chapter 5 for a list of variable types that can be changed with the CHAVAL function.

Example: In the example below, the CHAVAL function is used to change the contents of array element SCORES(i). When the prompt

```
Enter name = value
```

is displayed in the top right hand corner, the following is entered on the home line:

```
scores(i)=88.
```

The value of array element SCORES(i) is changed to 88 and will be used in subsequent computations. To see the new value of SCORES(i), you can use the SEEVAL function.

```
scores(i)=88.
                                                        Enter name=value
 Debugging FORT
       program fort
       real average, total
       total=0.
       call aver(total)
       average=total/5.
       print*,'average = ',average
       end
       subroutine aver(x)
       real scores(5)
       data scores /50., 56., 98., 12., 78./
       do 10 i=1,5
 -->   x=scores(i)+x
       end
------------------------------ OUTPUT ------------------------------
                 -- Welcome to Full Screen Debugging --

                       Press HELP for Assistance

   StepN        Locate    ChaVal    DelBrk    Deas    ZmOut    Keys
f1 Step1 f2 MSpeed f3 HSpeed f4 SeeVal f5 SetBrk f6 Quit f7 Trace f8 Goto
```

Locating Information in a Window

The BACK, FIRST, LAST, NEXT, HOME, DOWN, UP, BKW, FWD, and LOCATE functions are used to locate information on the screen by navigating through the windows.

Restore Source Window to Previous View (BACK)

Purpose: Restores the Source window to what was previously displayed. For example, if you press ZMOUT from the Zoom-in display, pressing BACK returns to the Zoom-in display.

Function: BACK

Remarks: If BACK is pressed after a function that requires additional input (such as SEEVAL), BACK cancels the function.

Display the First Screen of Information in the Window (FIRST)

Purpose: Displays the first screen of information in the current window.

Function: FIRST

Remarks:

- The current window is the window which contains the cursor before FIRST is pressed.
- If you are in the Source window, FIRST takes you to the first line of the current source file.

Display the Last Screen of Information in the Window (LAST)

Purpose: Displays the last screen of information in the current window.

Function: LAST

Remarks: The current window is the window which contains the cursor before LAST is pressed.

Perform Window-Dependent Action (NEXT)

Purpose: NEXT is the carriage-return key for a terminal.

Function: NEXT (or RETURN key)

Remarks:

- The action performed by NEXT is context-dependent. NEXT removes the Help or Keys windows if they are displayed, and restores the previously visible window. For other windows, NEXT advances the cursor to the line containing the arrow.
- NEXT cannot be entered on the home line.

Move to the Home Line (HOME)

Purpose: Moves the cursor to the left edge of the home line.

Function: HOME

Remarks:

- Screen mode functions, Debug commands, and SCL commands can be entered on the home line.
- Usually, the home line is the top line of the screen; on some terminals it is the bottom line.
- The HOME key is either a function key or a special key on your keyboard.

Move Cursor Position Line to Bottom of Window (DOWN)

Purpose: Moves the current line to the bottom of its window.

Function: DOWN

Remarks: The current line is the line where the cursor is positioned before DOWN is pressed.

Move Cursor Position Line to Top of Window (UP)

Purpose: Moves the current line to the top of its window.

Function: UP

Remarks: The current line is the line where the cursor is position before UP is pressed.

Move One Page Back From Cursor Position (BKW)

Purpose: Scrolls the current window backward to the previous screen of text.

Function: BKW

Remarks: The current window is the window which contains the cursor before BKW is pressed.

Move One Page Forward From Cursor Position (FWD)

Purpose: Scrolls the current window forward to the next screen of text.

Function: FWD

Remarks: The current window is the window which contains the cursor before FWD is pressed.

Prompt for Text String Then Search Forward (LOCATE)

Purpose: Prompts you for a text string and searches for it.

Function: LOCATE

Remarks:

- A forward search for the string is made in the current window. If found, the window is updated to show the string. The current window is the window which contains the cursor before LOCATE is pressed.
- If RETURN is pressed in response to the LOCATE prompt and no text string is entered, the next occurrence of the previously located string is found.

Example: In the following example, the LOCATE function is used to locate a specific line. When LOCATE is pressed, the prompt

    Enter text to locate

is displayed in the top right hand corner of the screen. On the home line, the text

    FOR COUNTER

is entered. This is the text to be searched in the source program. When the text is found, the cursor is moved to the line containing the text.

```
FOR COUNTER
                                                  Enter text to locate
  Debugging $MAIN
  --> MONTHTABLE$(16)
  LET DIVIDEND = -100
  LET DIVISOR = 0

  LET MONTHCOLUMN = 1
  LET MONTHLIST$ = "JANFEBMARAPRMAYJUN"

  FOR COUNTER = 1 TO 10
      CALL SQUAREPROCEDURE (COUNTER)

------------------------------ OUTPUT ------------------------------
                 -- Welcome to Full Screen Debugging --

                       Press HELP for Assistance

    StepN          Locate  ChaVal  DelBrk  Deas    ZmOut    Keys
 f1 Step1  f2 MSpeed f3 HSpeed f4 SeeVal f5 SetBrk f6 Quit f7 Trace f8 Goto
```

Tailoring a Screen Debug Session

The REFRSH, SPLIT, LEFT, RIGHT, and OPTS functions modify the characteristics of your terminal screen so that it displays a debugging session suited to your needs.

Refresh the Screen (REFRSH)

Purpose: Clears the entire screen then rewrites it. If you suspect the screen does not look right, rewrite it by entering REFRSH.

Function: REFRSH

Dividing the Screen Windows (SPLIT)

Purpose: Adjusts the relative sizes of the upper and lower windows.

Function: SPLIT

Remarks:

- The division between the windows is determined by the current line. The current line is the line where the cursor is positioned before SPLIT is pressed.
- You cannot eliminate either the Source window or the Output window. Each must contain at least one line.

Align Source Column With Leftmost Window Column (LEFT)

Purpose: Aligns the column containing the cursor position to the leftmost column of the window.

Function: LEFT

Remarks: The LEFT function appears only when the terminal screen is not wide enough to show an entire source line.

Align Leftmost Window Column With Source Column (RIGHT)

Purpose: Aligns the leftmost column of the window to the column containing the cursor position.

Function: RIGHT

Remarks: The RIGHT function appears only when the terminal screen is not wide enough to show an entire source line.

Compress Characters (NARROW)

Purpose: Displays the maximum number of characters per line that your terminal can support.

Function: NARROW

Remarks:

- The NARROW function is available only for terminals that support compressed characters.
- The NARROW function toggles with the WIDE function.

Enlarge Compressed Characters (WIDE)

Purpose: Displays the minimum number of characters per line that your terminal can support.

Function: WIDE

Remarks:

- The minimum number of characters that can be displayed on any terminal is 80.
- The WIDE function is available only for terminals that support compressed characters.
- The WIDE function toggles with the NARROW function.

Viewing Screen Options (OPTS)

Purpose: Displays the Options window. The Options window displays a list of options you can modify to tailor the debugging session.

Function: OPTS

Remarks:

- The following options can be modified:
  - The number of function key rows displayed at the bottom of your screen. The default is to display one row of function keys.
  - The type of modules you can debug (USER or SYSTEM). When in USER mode, information about the SYSTEM modules is suppressed. The default is USER mode.
  - The size of the Source window. This is the number of lines used by the Source window not including the home line, response line, or function key rows. The default is 3/4 of the available area on the screen.
  - The size of the Output window. This is the number of lines used by the Output window not including the home line, response line, or function key rows. The default is 1/4 of the available area on the screen.
- To change an option, you select a new value for the option by pressing the function key associated with the function.
- The NEXT function removes the Options window and restores the previously selected window(s).

Ending the Screen Mode Debug Session

The QUIT and DEAS functions end the screen mode Debug session. All the changes made during the Debug session are lost when the session is ended. All variables assume their original values, breaks are removed, and the program is the same as when you compiled it. You can run additional sessions if you want to continue debugging your program.

Terminate the Debug Session (QUIT)

Purpose: Ends the Debug Session and returns you to the utility you were using before you began the Debug session.

Function: QUIT

End Screen Mode Debug and Switch to Line Mode Debug (DEAS)

Purpose: Ends the Debug session in screen mode and switches you to line mode debugging.

Function: DEAS

Remarks: Breaks remain set, but step-mode is deactivated. All other options return to their default settings.

Line Mode Debugging 5

This chapter describes the commands and functions used in line mode debugging.

Line Mode Command and Function Summary 5-1

Debug Line Mode Commands 5-3
ACTIVATE_SCREEN (ACTS) 5-4
CHANGE_DEFAULT (CHAD) 5-6
CHANGE_MEMORY (CHAM) 5-11
CHANGE_PROGRAM_VALUE (CHAPV) 5-15
CHANGE_REGISTER (CHAR) 5-24
DELETE_BREAK (DELB) 5-28
DISPLAY_BREAK (DISB) 5-30
DISPLAY_CALL (DISC) 5-33
DISPLAY_DEBUGGING_ENVIRONMENT (DISDE) 5-36
DISPLAY_MEMORY (DISM) 5-39
DISPLAY_PROGRAM VALUE (DISPV) 5-46
DISPLAY_REGISTER (DISR) 5-58
DISPLAY_STACK_FRAME (DISSF) 5-61
QUIT (QUI) 5-65
RUN 5-66
SET_BREAK (SETB) 5-67
SET_STEP_MODE (SETSM) 5-85

Debug Line Mode Functions 5-91
$CURRENT_LINE ($CL) 5-92
$CURRENT_MODULE ($SCM) 5-93
$CURRENT_PROCEDURE ($CP) 5-94
$CURRENT_PVA ($CPVA) 5-95
$MEMORY ($MEM) 5-96
$PROGRAM_VALUE ($PV) 5-97
$REGISTER ($REG) 5-100

Line Mode Debugging 5

This chapter describes the Debug commands and functions that are available when using line mode debugging.

Line Mode Command and Function Summary

Command or Function	Description Page	Page
ACTIVATE_SCREEN or ACTS	Switches to screen mode debugging.	5-4
CHANGE_DEFAULTS or CHAD	Changes default Debug input/ output files and module and procedure names.	5-6
CHANGE_MEMORY or CHAM	Changes the contents of memory.	5-11
CHANGE_PROGRAM_VALUE or CHAPV	Changes the value a program variable.	5-15
CHANGE_REGISTER or CHAR	Changes the contents of a register.	5-24
DELETE_BREAK or DELB	Deletes break definitions.	5-28
DISPLAY_BREAK or DISB	Displays break definitions.	5-30
DISPLAY_CALLS or DISC	Displays information about the dynamic call chain.	5-33
DISPLAY_DEBUGGING_ENVIRONMENT or DISDE	Displays the environment of your session.	5-36
DISPLAY_MEMORY or DISM	Displays the content of memory.	5-39
DISPLAY_PROGRAM_VALUE or DISPV	Displays the value of a program variable.	5-46

Continued

Continued

Command or Function	Description Page	Page
DISPLAY_REGISTER or DISR	Displays the contents of a register.	5-58
DISPLAY_STACK_FRAME or DISF	Displays the contents of one or more stack frames.	5-61
QUIT	Ends the Debug session.	5-65
RUN	Begins or resumes program execution.	5-66
SET_BREAK or SETB	Defines a named break.	5-67
SET_STEP_MODE or SETSM	Turns step mode on or off.	5-85
$CURRENT_LINE or $CL	Returns value of current line.	5-92
$CURRENT_MODULE or $CM	Returns name of current module.	5-93
$CURRENT_PROCEDURE or $CP	Returns name of current procedure.	5-94
$CURRENT_PVA or $CPVA	Returns value of PVA.	5-95
$MEMORY or $MEM	Returns contents of memory.	5-96
$PROGRAM_VALUE or $PV	Returns displayable value of a program variable.	5-97
$REGISTER or $REG	Returns contents of a register.	5-100

Debug Line Mode Commands

The Debug commands follow the syntax and conventions for SCL commands, as described in the SCL Language Definition Usage manual.

If the command parameters are specified positionally and a parameter is omitted, its position must be indicated by a coma.

To allow for future development of Debug, any command sequences you intend to save should contain named parameters so that the command interpretations are independent of parameter position.

The source language in which your program is written determines the use of some of the command parameters. Source language dependencies are identified in the applicable parameter description.

NOTE

FORTRAN source language dependencies apply to both FORTRAN Version 1 and FORTRAN Version 2 programs, unless otherwise specified.

ACTIVATE_SCREEN (ACTS)

Purpose: Switches to screen mode Debug from line mode Debug.

Format:
```
ACTIVATE_SCREEN or
ACTS
    SOURCE_FILES = list of files                  (optional)
    STATUS = status variable                      (optional)
```

Parameters: SOURCE_FILES or SF

Specifies the files containing the source statements of the program to be debugged. Options are:

Omitted

Initiates screen mode Debug with the Zoom-in display for programs written in C, COBOL, CYBIL, and FORTRAN. Initiates screen mode Debug with the Zoom-out display for programs written in BASIC and Pascal. To switch to the Zoom-in display, you must press the ZMIN function. (See the description of the Zoom-in and Zoom-out displays in chapter 4).

List of files

Specifies the file or files containing your source programs to be debugged in screen mode. This parameter is provided mainly for languages which do not initiate screen mode Debug with the Zoom-in display. This allows you to start screen mode with the Zoom-in display rather than the Zoom-out display. The FILE_PROCESSOR attribute must be set for any files specified.

STATUS

Specifies a variable to receive the return status of the command. Options are:

Omitted

The next command is processed if an error does not occur. The return status is output to $RESPONSE (and to the debug_output file if $RESPONSE is connected to that file) if an error does occur. $RESPONSE is normally connected during interactive debugging.

Status variable

The named variable receives the return status.

Remarks:

- To use the SOURCE_FILES parameter of the ACTIVATE_SCREEN command, the FILE_PROCESSOR attribute of each file must contain the name of the compiler that compiled the file. This is done with the CHANGE_FILE_ATTRIBUTE (descibed in the SCL System Interface manual) command before you begin the Debug session.
- You can enter ACTIVATE_SCREEN anytime during an interactive line mode Debug session.
- If the ACTIVATE_SCREEN command is included in a Debug input file, the Debug session switches to screen mode.
- When ACTIVATE_SCREEN is entered, any existing breaks are deleted and STEP_MODE is turned off. The Debug input and output files are changed to specific files used by Debug for screen mode.
- During a screen mode Debug session, screen mode functions, certain line mode commands, and SCL commands are available. The DEAS function can be used to return to line mode.

Examples: In the following example, the Debug session is switched from line mode to screen mode. If the source program to be debugged is written in C, COBOL, CYBIL, FORTRAN Version 1, or FORTRAN Version 2, the screen mode session immediately begins with the Zoom-in display. If the source program to be debugged is written in BASIC or Pascal, the screen mode session begins with the Zoom-out display. To move to the Zoom-in display, you must press the ZMIN function key.

```
DB/activate_screen
```

CHANGE_DEFAULT (CHAD)

Purpose: Changes the default module, default procedure, default Debug input file, and default Debug output file.

Format:

```
CHANGE_DEFAULT or
CHANGE_DEFAULTS or
CHAD
    MODULE = name or $CURRENT              (optional)
    PROCEDURE = name or $CURRENT           (optional)
    DEBUG_INPUT = file                     (optional)
    DEBUG_OUTPUT = file                    (optional)
    STATUS = status variable               (optional)
```

Parameters: MODULE or M

Specifies the name that is used by default when the MODULE parameter is not specified in Debug commands that refer to a module. Options are:

Omitted

The current default module remains unchanged.

Name

The named module is used as the default module.

$CURRENT

The default module is reset to the module executing when Debug gains control.

Source	Parameter Dependency
BASIC	MODULE is the main program referred to as $MAIN. $MAIN can reference internal subroutines and internal functions which are procedures within the module $MAIN. MODULE can also name an external subroutine or external function. An external subroutine and an external function can reference internal subroutines and internal functions which are procedures within that module.
C	MODULE names a C compilation unit (a C source file). The initial module is EM, a startup module. The EM module has no Debug tables so the default, $CURRENT, is not useful until program execution reaches a module with Debug tables. Global variables are in the module c_globals.
COBOL	MODULE names the program specified on the Program-id statement. Since MODULE and PROCEDURE parameters name the same program, both parameters should be assigned the same name or only one parameter should be specified.
CYBIL	MODULE names a module which may contain a program, procedure, or function.
FORTRAN	MODULE names a program, subroutine, function, or block data subprogram. Since MODULE and PROCEDURE parameters name the same program unit, both parameters should be assigned the same name or only one parameter should be specified.
Pascal	MODULE is the program name.

PROCEDURE or P

Specifies the name that is used by default when the PROCEDURE parameter is not specified in Debug commands that refer to a procedure. Options are:

Omitted

The current default procedure remains unchanged.

Name

The named procedure is used as the default procedure.

$CURRENT

The default procedure is reset to the procedure executing when Debug gained control.

Source	Parameter Dependency
BASIC	PROCEDURE names internal subroutines and internal functions within a module.
C	PROCEDURE names a function or a block within a function.
COBOL	PROCEDURE names a program specified on the Program-id statement. Since MODULE and PROCEDURE parameters name the same program, both parameters should be assigned the same name or only one parameter should be specified.
CYBIL	PROCEDURE names a program, procedure, or function in the module specified by the MODULE parameter.
FORTRAN	PROCEDURE names a program, subroutine, function, or block data subprogram. Since MODULE and PROCEDURE parameters name the same program unit, both parameters should be assigned the same name or only one parameter should be specified.
Pascal	PROCEDURE names a program, procedure or function name.

DEBUG_INPUT or DI

Specifies the default file from which Debug commands are read when Debug next gains control. Options are:

Omitted

The current DEBUG_INPUT file remains unchanged. Unless otherwise specified, the initial DEBUG_INPUT file is COMMAND.

File

The named file is used as the DEBUG_INPUT file.

DEBUG_OUTPUT or DO

Specifies the default file on which Debug output is to be written. The change takes effect immediately. Both break report messages and command output are written to this file. Options are:

Omitted

The current DEBUG_OUTPUT file remains unchanged. Unless otherwise specified, the initial DEBUG_OUTPUT file is $OUTPUT.

File

The named file is used as the DEBUG_OUTPUT file.

STATUS

Specifies a variable to receive the return status of the command. Options are:

Omitted

The next command is processed if an error does not occur. The return status is output to $RESPONSE (or to the debug_output file if $RESPONSE is connected to that file) if an error does occur. $RESPONSE is normally connected during interactive debugging.

Status variable

The specified variable receives the return status.

Remarks:

- The effect of the CHANGE_DEFAULT command remains until altered by another CHANGE_DEFAULT command.

- If a DEBUG_INPUT or DEBUG_OUTPUT file is specified, the file is positioned at the beginning-of-information the first time it is used. The file is not repositioned the next time it is used. Commands are read from the file sequentially. If an end-of-information is reached on the input file during an interactive session, the input file is switched to COMMAND. If an end-of-information is reached on the input file during a batch session, program execution resumes.

Examples:

Assuming that the working catalog has been set to $USER, the following command specifies that Debug is to read its commands from file $USER.DBIN when program execution is suspended next. It also specifies that Debug is to write its output to file $USER.DBOUT, beginning immediately.

```
DB/change_default debug_input=dbin ..
DB../debug_output=dbout
```

The following command specifies that output from the Debug session is to be written to file $LIST:

```
DB/change_default debug_output=$list
```

The following command specifies the default module to be MAIN:

```
DB/chad m=main
```

The C_GLOBALS module is specified as the default module:

```
DB/change_default module=c_globals
```

CHANGE_MEMORY (CHAM)

Purpose: Changes the contents of memory starting at a specific address. You can change the value of any memory location for which you have write permission.

Format:

```
CHANGE_MEMORY or
CHAM
    ADDRESS = rsssoooooooo or ?svar or
              function                          (required)
    VALUE = integer or string                   (required)
    TYPE = keyword                              (optional)
    REPEAT_COUNT = positive integer or ALL      (optional)
    STATUS = status variable                    (optional)
```

Parameters: ADDRESS or A

Specifies the address of the first byte of memory to change. Options are:

rsssoooooooo

where r is the ring number, sss is the segment number, and oooooooo is the offset from the beginning of the segment. The specified value cannot contain any blanks.

?svar

Specifies the address indicated by the value of the SCL variable.

function

Specifies the address indicated by a Debug or SCL function.

VALUE or V

Specifies the new memory value. Options are:

String

A string value can be interpreted as a hexadecimal or ASCII string, depending on the TYPE parameter.

A hexadecimal string consists of the hexadecimal digits 0 through 9, A through F, and blanks. Blanks are ignored, but they can be used to improve readability. Each hexadecimal digit corresponds to 4 bits of memory. The first two digits replace the first byte of memory at the specified address, the second two digits replace the second byte, and so on. If there is an odd number of hexadecimal digits, only the first half of the last byte is changed.

An ASCII string consists of a string of ASCII characters. Each ASCII character corresponds to one byte of memory. The first character replaces the first byte of memory at the specified address, the second character replaces the second byte, and so on.

Integer

An integer value completely replaces the contents of eight bytes. A diagnostic message is issued if the integer exceeds eight bytes.

TYPE or T

Specifies the type of data defined by the VALUE parameter. Options are:

Omitted

Type is HEX for string values and INTEGER for numeric values.

ASCII or A

Data is ASCII string values.

HEX or H

Data is hexadecimal string values.

INTEGER or I

Data is integer values.

REPEAT_COUNT or RC

Specifies the number of times the VALUE parameter is placed in memory. Options are:

Omitted

A value of 1 is used.

Positive integer

Defines the number of times the VALUE parameter is repeated in memory. The address is incremented by the value size each time the value is repeated. The memory change is limited to the end of the data segment containing the specified address. Specifying an integer that is too large changes all the memory that can be changed. The memory change is limited to the end of the data segment containing the specified address.

ALL

Changes all the memory that can be changed. The memory change is limited to the end of the data segment containing the specified address.

STATUS

Specifies a variable to receive the return status of the command. Options are:

Omitted

The next command is processed if an error does not occur. The return status is output to $RESPONSE (and to the Debug_output file if $RESPONSE is connected to that file) if an error does occur. $RESPONSE is normally connected during interactive debugging.

Status variable

The named variable receives the return status.

Examples: The following example replaces four bytes of memory beginning at location B04700000A18(16) with the hexadecimal string '1010aaab':

```
DB/change_memory address=b04700000A18 ..
DB../value='1010aaab'
```

or abbreviated,

```
DB/cham a=b04700000A18 v='1010aab'
```

The following command displays the new value of address location B04700000A18:

```
DB/dism a=b04700000a18
STARTING ADDRESS:  B 047 00000A18
00000000 1010 AAAB 0000 50D0 ??????P?
```

The following example replaces six bytes of memory beginning at location 0B04700000598(16) with the ASCII string 'STRING':

```
DB/change_memory address=b04700000598 ..
DB../value='string' type=ascii
```

The following command displays the new value of address location B04700000598:

```
DB/dism a=b04700000598
STARTING ADDRESS:  B  047 00000598
00000000 7374 7269 6E67 1B00 string??
```

The following example replaces eight bytes of memory beginning at location B02300000223(16) with the integer value 44:

```
DB/change_memory address=b02300000223 value=44
```

The following example changes the value of 5 1/2 bytes of memory starting at the address specified by the SCL variable TCOUNT to '4a000000103' (after TCOUNT has been created with the SCL CREATE_VARIABLE command):

```
DB/disv tcount
tcount=B02300000223(16)
DB/change_memory address=?tcount value='4a000000103'
```

CHANGE_PROGRAM_VALUE (CHAPV)

Purpose: Changes the value of named program variables.

Format:

```
CHANGE_PROGRAM_VALUE or
CHAPV
    NAME = variable                               (required)
    VALUE = variable or Constant                  (required)
    MODULE = name                                 (optional)
    PROCEDURE = name                              (optional)
    RECURSION_LEVEL = positive integer            (optional)
    RECURSION_DIRECTION = keyword                 (optional)
    STATUS = status variable                      (optional)
```

Parameters: NAME or N

Simple variable names, subscripted names (subscripts can be constants or variables, but not expressions), field references and pointer dereferences can be specified. SCL string, integer, and boolean variables can be used as aliases for program variable names. To do this, assign the SCL string variable to a string containing the identifier. Then use the SCL Variable preceded by a question mark (?) as the value for the name. Options are:

Omitted

The NAME parameter is required for all languages except COBOL. If NAME is omitted while debugging a COBOL program, you are prompted for the name.

Variable

Specifies the name of a program variable or data structure to be changed. The value of the named identifier is changed. The MODULE and PROCEDURE parameters can be specified to qualify the name.

Source	Parameter Dependency
BASIC	A variable can be an array element, but cannot be a whole array. Reference substrings by MID$(string_var_name, first_char_position, length) and string_var_name(first_char_position:last_char_position) formats where first_char_position, last_char_position are positive integer values.
COBOL	Reference modification is supported. If the data-name contains blanks, hyphens, a digit as the first character, or other non-SCL syntax, you may not be able to use the NAME parameter to specify the data-name; omit NAME and you will be prompted for the data-name. Data_names with qualified subscripts are not supported. However, a qualified subscript can be displayed first then the displayed value may be substituted for the subscript reference. To display qualified data_names, omit the NAME parameter. When the NAME parameter is not specified, an ENTER_NAME prompt is issued to the current DEBUG_INPUT file. ENTER_NAME recognizes the COBOL qualifiers OF and IN and permits blanks around tokens, such a ´(´, ´)´, and ´,´.
FORTRAN	Extensible common blocks can be specified. The values of named constants created by the PARAMETER statement cannot be changed.
Pascal	A set can be specified.

VALUE or V

Specifies the new value of the variable or constant for the NAME parameter variable. Replacement values must be entered in the same format as defined in your program, not as they are represented in memory. Options are:

Variable or Constant

The VALUE parameter variable must be the same type as the NAME parameter variable or an SCL constant of an appropriate type.

Source	Parameter Dependency
BASIC	You cannot change a variable to another variable.
C	If name is a pointer, VALUE must be integer, NIL, or a variable of the same type.
COBOL	If the value is a string that is shorter than the NAME variable, then the value is blank-filled on the right to equal the NAME variable length. If the value is longer than the NAME variable, then the value is truncated on the right to equal the NAME variable length. No additional editing is done.
CYBIL	If NAME is a pointer, VALUE can be an integer constant, a variable of the same type, or NIL. If NAME is a cell, VALUE can be an integer constant or a variable of the same type. If NAME is an ordinal, VALUE can be an ordinal name or a variable of the same type. If NAME is an array, record, or sequence, VALUE must be a variable of the same type, byte-aligned and unpacked. If NAME is a set, VALUE can be an allowable set element or a variable of the same type. The set element must be enclosed in brackets. For example: `CHAPV set_name [value]`
FORTRAN	The value of any variable can be changed to a constant of the same type, to a variable reference of the same type, or to a symbolic constant (created by the PARAMETER statement) of the same type. Constants cannot be changed. Boolean constants must be entered as hexadecimal integers.

If the value of a string that is shorter than the NAME variable, then the value is extended to the right with blanks to the same length as the NAME variable. If the value is longer than the NAME variable, then the value is truncated on the right to equal the NAME variable length.

Unsubscripted arrays can only be changed to a variable reference of the same type and size.

Source	Parameter Dependency
Pascal	If NAME is a set, VALUE can be an allowable set element or a variable of the same type. The set element must be enclosed in brackets. For example: `CHAPV set_name [value]`

MODULE or M

This parameter qualifies the NAME parameter. It specifies the name of the module containing the named program variable or data structure specified by the NAME parameter. Options are:

Omitted

The module that is executing when Debug gains control is used unless the Debug CHANGE_DEFAULTS command has specified another MODULE value. The default module can be changed to the module specified by the CHANGE_DEFAULTS MODULE parameter.

Name

Specifies the name of the module that contains the name of the program variable or data structure specified by the NAME parameter.

Source	Parameter Dependency
BASIC	MODULE is the main program referred to as $MAIN. $MAIN can reference internal subroutines and internal functions which are procedures within the module $MAIN. MODULE can also name an external subroutine or external function. An external subroutine and an external function can reference internal subroutines and internal functions which are procedures within that module.
C	MODULE names a C compilation unit (a C source file). The initial module is EM, a startup module. The EM module has no Debug tables so the default, $CURRENT, is not useful until program execution reaches a module with Debug tables. Global variables are in the module c_globals.
COBOL	MODULE names a program specified on the Program-id statement. Since MODULE and PROCEDURE parameters name the same program, both parameters should be assigned the same name or only one parameter should be specified.
CYBIL	MODULE names a module which may contain a program, procedure, or function. MODULE must be specified if you want to change a variable value after a runtime error occurs. This is described in more detail in chapter 6, Debugging a CYBIL Runtime Error.)
FORTRAN	MODULE names a program, subroutine, function, or block data subprogram. Since the MODULE and PROCEDURE parameters name the same program unit, both parameters should be assigned the same name or only one parameter should be specified. The MODULE name is the name of the program unit containing the definition of the variable specified by the NAME parameter.
Pascal	The MODULE name is the program name.

PROCEDURE or P

This parameter qualifies the NAME parameter. It specifies the name of the procedure containing the named program variable or data structure specified by the NAME parameter. If a procedure is inactive (not in an active call chain), its variables cannot be displayed by DISPV because the procedure has no stack frame and therefore no existent automatic variables. Options are:

Omitted

The procedure that is executing when Debug gains control is used unless the Debug CHANGE_DEFAULTS command has specified another PROCEDURE value. The default procedure can be changed to the procedure specified by the CHANGE_DEFAULTS PROCEDURE parameter.

Name

Specifies the name of the procedure that contains the name of the program variable or data structure specified by the NAME parameter.

Source	Parameter Dependency
BASIC	PROCEDURE names internal subroutines and internal functions within a module.
C	PROCEDURE names a function or a block within a function.
COBOL	PROCEDURE names a program specified on the Program-id statement. Because MODULE and PROCEDURE parameters name the same program, both parameters should be assigned the same name or only one parameter should be specified.
CYBIL	PROCEDURE names a program, procedure, or function in the module specified by the MODULE parameter. PROCEDURE must be specified if you want to change a variable value after a runtime error occurs. (This is described in detail in chapter 6, Debugging a CYBIL Runtime Error.)

Source	Parameter Dependency
FORTRAN	PROCEDURE names a program, subroutine, function, or block data subprogram. Because the PROCEDURE and MODULE parameters name the same program unit, both parameters should be assigned the same name or only one parameter should be specified.
Pascal	PROCEDURE names a program, procedure, or function name.

RECURSION_LEVEL or RL

Specifies the recursion level of the procedure specified by the PROCEDURE parameter. The value of the data item specified by the NAME parameter and known to this recursive level of the procedure is displayed. Recursion does not affect static or global variables. Options are:

Omitted

A value of 1 is used.

Positive integer

The specified value must be a positive integer greater than 0. If the RECURSION_DIRECTION parameter specifies BACKWARD, 1 is the most recent invocation of the procedure, 2 is its predecessor, and so on (backward count from the most recent call). If the RECURSION_DIRECTION parameter specifies FORWARD, 1 is the first invocation of the procedure, 2 is the procedure called by the first invocation, and so on (forward count from the first call).

Source	Parameter Dependency
BASIC	RECURSION_LEVEL specifies the recursive call instance.
COBOL FORTRAN	Not supported. Omit the RECURSION_LEVEL parameter.
CYBIL	RECURSION_LEVEL specifies the particular call of a recursive procedure to be used.
Pascal	RECURSION_LEVEL specifies which invocation of a named recursive procedure is to be used.

RECURSION_DIRECTION or RD

Specifies whether the RECURSION_LEVEL is counted forward from the first call or backward from the most recent call. Options are:

Omitted

The value of BACKWARD is used.

FORWARD

A RECURSION_LEVEL of 1 specifies the first call to the procedure, 2 specifies the second call, and so on.

BACKWARD

A RECURSION_LEVEL of 1 specifies the most recent call to the procedure, 2 specifies its predecessor, and so on.

Source	Parameter Dependency
BASIC C CYBIL Pascal	RECURSION_DIRECTION specifies the order in which calls to a recursive procedure are searched.
COBOL FORTRAN	Not supported. Omit the RECURSION_ DIRECTION parameter.

STATUS

Specifies a variable to receive the return status of the command. Options are:

Omitted

The next command is processed if an error does not occur. The return status is output to $RESPONSE (and to the Debug_output file if $RESPONSE is connected to that file) if an error does occur. $RESPONSE is normally connected during interactive debugging.

Status variable

The named variable receives the return status.

Remarks: Replacement values are entered in the same format as defined in your program, not as they are represented in memory.

Examples: The following three examples refer to the FORTRAN definition below:

```
    COMMON /BLK/ DVAL, RVAL, IVAL, ZVAL
    DATA DVAL, RVAL, IVAL, ZVAL /20.0D+0, 3.45E+01, 30,
   +(+20.0,20.3)/
```

The following command displays the initial value of variable DVAL:

```
DB/display_program_value name=dval
dval = 20.
```

The following command changes the value of variable DVAL to 30.0:

```
DB/change_program_value name=dval value=+30.0d+0
```

The following command displays the new value of DVAL:

```
DB/dispv dval
dval = 30.
```

The following command changes the value of variable INDEX:

```
DB/chapv name=index value=63 module=test1
```

The following command changes the value of logical variable VAR:

```
DB/change_program_value var value=true
```

The following commands display the new values of INDEX and VAR:

```
DB/dispv index
index = 63
DB/dispv var
var = TRUE
```

CHANGE_REGISTER (CHAR)

Purpose: Changes the value of the P, A, or X registers that are associated with the program executing when Debug gains control.

Format:

```
CHANGE_REGISTER or
CHANGE_REGISTERS or
CHAR
    KIND = keyword                              (optional)
    NUMBER = integer or range of integers       (optional)
             or ALL
    VALUE = integer or String                   (required)
    TYPE = keyword                              (optional)
    STATUS = status variable                    (optional)
```

Parameters: KIND or K

Specifies the register to change. The KIND parameter changes the value of the registers for the function currently being executed. Options are:

Omitted

Same as KIND=P.

P

Changes the P register. Changing the P register changes the point in the program at which Debug resumes execution.

A

Changes the A registers.

X

Changes the X registers.

NUMBER or N

Indicates which A or X registers specified by the KIND parameter will change. This parameter is ignored if KIND = P because there is only one P register.

The A and X registers can be saved and changed when Debug gains control. However, some registers are not always saved; a message is issued for each register that cannot be changed because it was not saved. Options are:

Omitted

Changes the zero register.

Integer or range of integers

Changes a set of registers. Integer can be 0 through 15.

ALL

Changes all A (if KIND = A) or all X (if KIND = X) or both sets of (if KIND = ALL) registers.

VALUE or V

Specifies the new value of the register. Options are:

Omitted

Same as integer or string allowed for KIND=P.

Integer

If KIND = P or A, integer must be in the range 0 through 0FFFFFFFFFFF(16).

If KIND = X, integer must be in the range -7FFFFFFFFFFFFFFF(16) through 7FFFFFFFFFFFFFFF(16).

The upper 4 bits are ignored when changing the P register because the ring number in P cannot be changed.

The upper bits of the register are set to zero if an integer is negative or to 1 if an integer is positive when the value does not fill the register.

String

If KIND = P or A, string can be a hexadecimal string containing a maximum of 12 hexadecimal digits (spaces are ignored); each hexadecimal digit corresponds to 4 bits.

If KIND = X, string can be a hexadecimal string containing a maximum of 16 hexadecimal digits (spaces are ignored); each hexadecimal digit corresponds to 4 bits or an ASCII string containing a maximum of eight ASCII characters; each character corresponds to one byte.

If a string value does not fill the register (it is less than 16 hexadecimal digits or 8 ASCII characters), the string value is left-justified with remaining bytes unchanged.

TYPE or T

Specifies the type of data specified by the VALUE parameter. Options are:

Omitted

HEX is used for string values and INTEGER is used for numeric values.

ASCII or A

Data is an ASCII string. Each ASCII character corresponds to 1 byte. Spaces are significant.

HEX or H

Data is a hexadecimal string. Each hexadecimal digit corresponds to 1/2 byte (4 bits). Spaces have no effect on the result.

INTEGER or I

Data is an integer value.

STATUS

Specifies a variable to receive the return status of the command. Options are:

Omitted

The next command is processed if an error does not occur. The return status is output to $RESPONSE (and to the Debug_output file if $RESPONSE is connected to that file) if an error does occur. $RESPONSE is normally connected during interactive debugging.

Status variable

The specified variable receives the return status.

Examples: The following example changes the current value of the P register to 0A02200004500(16). Because the register is the P register, its ring number (upper 4 bits) is ignored.

```
DB/change_register kind=p value=0a02200004500(16)
```

The following command displays the new value of the P register:

```
DB/display_register kind=p
P=B 022 00004500
```

The following command changes the leftmost 5 bytes of the X7 register to ´abcde´. The rightmost bytes are left unchanged.

```
DB/char kind=x number=7 value=´abcde´ type=ascii
```

The following command displays the new value of the X register:

```
DB/disr k=x n=7
X7=61626364 65000EBA
```

DELETE_BREAK (DELB)

Purpose: Deletes one or more previously defined breaks.

Format:

```
DELETE_BREAK or
DELETE_BREAKS or
DELB
    BREAK = list of name or ALL                 (required)
    STATUS = status variable                    (optional)
```

Parameters: BREAK or BREAKS or B

Specifies the break definitions to be deleted. Options are:

List of names

Deletes the break associated with each name. If the keyword ALL appears in the list of names, all breaks are deleted. An informative message is issued if a specified break name does not exist, however, all subsequent breaks in the list are processed.

ALL

Deletes all breaks.

STATUS

Specifies a variable to receive the return status of the command. If the DELETE_BREAK command contains an error before the STATUS parameter, the remainder of the command is not read and the return status is not stored in the specified variable. Options are:

Omitted

The next command is processed if an error does not occur. The return status is output to $RESPONSE (and to the Debug_output file if $RESPONSE is connected to that file) if an error does occur. $RESPONSE is normally connected during interactive debugging.

Status variable

The specified variable receives the return status.

PERFORMANCE HINT

Because execution takes longer when there are breaks set, you should delete a break as soon as it is no longer needed.

Examples: The following command deletes break definitions B1, B2, and B3:

```
DB/delete_breaks breaks=(b1,b2,b3)
```

or abbreviated,

```
DB/delb b=(b1,b2,b3)
```

The following command deletes all break definitions:

```
DB/delete_breaks all
```

The following command deletes break definition B4:

```
DB/delete_break b4
```

DISPLAY_BREAK (DISB)

Purpose: Displays specified break definitions. The break name, events, address, and any commands associated with the break are displayed.

Format:

```
DISPLAY_BREAK or
DISPLAY_BREAKS or
DISB
    BREAK = list of name or ALL                (optional)
    OUTPUT = file                              (optional)
    STATUS = status variable                   (optional)
```

Parameters: BREAK or BREAKS or B

Specifies the break definitions to be displayed. Options are:

Omitted

Displays all breaks.

List of names

Displays the break associated with each name. If the keyword ALL appears in the list of names, all breaks are displayed. An informative message is issued if a specified break name does not exist, however, all subsequent breaks in the list are processed.

ALL

Displays all breaks.

OUTPUT or O

Specifies the file on which the break definitions are to be written. Options are:

Omitted

Writes to the current default Debug output file.

File

Writes to the named file. You can position the file by appending a position indicator to the file name (.$BOI, .$ASIS, .$EOI).

STATUS

Specifies a variable to receive the return status of the command. Options are:

Omitted

The next command is processed if an error does not occur. The return status is output to $RESPONSE (and to the Debug output file if $RESPONSE is connected to that file) if an error does occur. $RESPONSE is normally connected during interactive debugging.

Status variable

The named variable receives the return status.

Examples: The following command displays break definitions B1, B2, and B5:

```
DB/display_breaks breaks=(b1,b2,b5)

    --  Break B1
    --  event(s) = execution
    --  location: M=main L=26

    --  Break B2
    --  event(s) = execution
    --  location: M=m L=13 BO=16

    --  Break B5
    --  event(s) = execution
    --  location: M=s L=16
```

The following command displays all break definitions:

```
DB/display_breaks

   --  Break B1
   --  event(s) = execution
   --  location: M=main L=26

   --  Break B2
   --  event(s) = execution
   --  location: M=m L=13 BO=16

   --  Break B3
   --  event(s) = execution
   --  location: M=m L=16

   --  Break B4
   --  event(s) = execution
   --  location: M=mult L=7
```

The following command writes the break definitions of all existing breaks to permanent file $USER.DEBUG_OUTPUT:

```
DB/display_breaks output=$user.debug_output
```

or abbreviated,

```
DB/disb o=$user.debug_output
```

DISPLAY_CALL (DISC)

Purpose: Traces back the call chain; information about the active call chain is displayed. Information includes which procedure called the current procedure, which procedure called its caller, which procedure called its caller's caller, and so on.

Format:

```
DISPLAY_CALL or
DISPLAY_CALLS or
DISC
    COUNT = positive integer or ALL             (optional)
    START = positive integer                    (optional)
    DISPLAY_OPTION = list of keyword            (optional)
    OUTPUT = file                               (optional)
    STATUS = status variable                    (optional)
```

Parameters: COUNT or C

Specifies the number of calls to be displayed. Options are:

Omitted

Displays all calls.

Positive integer

Displays the specified number of calls; all calls are displayed if the integer is greater than the number of existing calls.

ALL

Displays all calls.

START or S

Specifies which call on the chain to display first. Thus, it is possible to skip the most recent calls. Options are:

Omitted

Displays the most recent call.

Positive integer

Displays the specified call; 1 represents the most recent call, 2 represents the predecessor of the most recent call, and so forth.

An informative message is issued if the specified number of calls is greater than the actual number of calls.

Source	Parameter Dependency
C	The first two calls are always to startup modules. The initial call to your program is the third call in the call chain.

DISPLAY_OPTION or DISPLAY_OPTIONS or DO

Specifies the type of information to be displayed. Options are one or more of the following:

Omitted

Displays calls that are in the user program only.

USER_CALLS (UC)

Displays calls that are in the user program only.

SYSTEM_CALLS (SC)

Displays calls that are not part of the user's program only.

ALL_CALLS (AC)

Displays both user calls and system calls.

VARIABLE_VALUES (VV)

Displays all variables known to the procedure.

Source	Parameter Dependency
C	The parameter option VARIABLE_VALUES does not display global variables.

OUTPUT or O

Specifies the file on which the call information is to be written. Options are:

Omitted

Writes to the current Debug output file.

File

Writes to the named file. You can position the file by appending a position indicator to the file name ($.BOI, .$ASIS, .$EOI).

STATUS

Specifies a variable to receive the return status of the command. Options are:

Omitted

The next command is processed if an error does not occur. The return status is output to $RESPONSE (and to the Debug output file if $RESPONSE is connected to that file) if an error does occur. $RESPONSE is normally connected during interactive debugging.

Status variable

The named variable receives the return status.

Remarks: Usually the procedure name, module name, and line number of each call are shown. If you inhibit Debug tables when compiling your program, only the procedure or module name and byte offset from the beginning of the procedure or module are shown. Machine addresses are shown for internal NOS/VE calls only.

Examples: The following command displays the first call on the call chain:

```
DB/display_calls
-- Called from procedure TEST module TEST at line 1
   byte offset 22
```

The following command displays the first two calls on the call chain:

```
DB/display_calls count=2 display_options=all_calls

-- Traceback from address B 22 4500
-- Called from module FLM$BOUND_CORE byte offset
   10CC(16)
```

The following command displays all of the user calls on the call chain, as well as all program variable values known to each procedure:

```
DB/display_calls display_options=(user_calls ..
DB../variable_values)
-- Called from procedure TEST module TEST at line 1
   byte offset 22
-- DISPLAY OF ALL CONSTANTS AND VARIABLES IN TEST

I = 10
J = 0
VAR = TRUE
```

DISPLAY_DEBUGGING_ENVIRONMENT (DISDE)

Purpose: Displays the environment you have set up for the Debug session.

Format:

```
DISPLAY_DEBUGGING_ENVIRONMENT or
DISDE
    DISPLAY_OPTION = list of keyword          (optional)
    OUTPUT = file                             (optional)
    STATUS = status variable                  (optional)
```

Parameters: DISPLAY_OPTION or DISPLAY_OPTIONS or DO

Specifies the type of information to be displayed. Options are one or more of the following:

Omitted

Same as DISPLAY_OPTION = ALL

DEFAULTS or D

Displays the current default values for module, procedure, Debug input file, and Debug output file.

Unless the CHANGE_DEFAULTS command has been specified, the default module and procedure are $CURRENT, that is, the module and procedure where execution has stopped in your task.

BREAKS or B

Displays the number of breaks you have set, the number of breaks currently in use by Debug, and the maximum number of allowed breaks. However, the maximum number of breaks that you can set is 32; the remaining 32 are reserved for Debug use.

STEP_MODE or SM

Displays the current STEP_MODE attributes.

USER_ADDRESS or UA

Displays the location in your program where execution has stopped.

ALL

Displays the defaults, breaks, STEP_MODE attributes, and user_addresses.

Source	Parameter Dependency
C	The STEP_MODE option is not available while debugging C programs.

OUTPUT or O

Specifies the file where the debugging environment display is to be written. Options are:

Omitted

Writes to the current default Debug output file.

File

Writes to the named file. You can position the file by appending a position indicator to the file name (.$BOI, .$ASIS, .$EOI).

STATUS

Specifies a variable to receive the return status of the command. Options are:

Omitted

The next command is processed if an error does not occur. The return status is output to $RESPONSE (and to the Debug output file if $RESPONSE is connected to that file) if an error does occur. $RESPONSE is normally connected during interactive debugging.

Status variable

The named variable receives the return status.

DISPLAY_DEBUGGING_ENVIRONMENT (DISDE)

Remarks: The DISPLAY_DEBUGGING_ENVIRONMENT command displays the following information:

- Current defaults for module, procedure, Debug input file, and Debug output file.
- Total number of breaks you have set and Debug has set.
- Information about STEP_MODE
- The location in your program where execution has stopped.

Examples: The following command writes the display to the current default Debug output file:

```
DB/display_debugging_environment do=all

--  Default module is $CURRENT(XYZ).
--  Default procedure is $CURRENT(P1).
--  Default debug_input file is :$LOCAL.COMMAND.1.
--  Default debug_output file is :$LOCAL.$OUTPUT.1.
--  The number of breaks set by the user is 5.
--  The number of breaks in use by DEBUG is 0.
--  The number of available breaks is 64.
--  Step_mode is OFF.
--  Execution is currently stopped at B 02 00004500.
```

DISPLAY_MEMORY (DISM)

Purpose: Displays information located at any address to which you have read access.

Format:

```
DISPLAY_MEMORY or
DISM
    address                                         (required)
        address can be one or more of the following:
            SECTION = name or keyword
            MODULE = name
            ADDRESS = rsssoooooooo or ?svar or function
    BYTE_OFFSET = 0 or integer                      (optional)
    BYTE_COUNT = positive integer                   (optional)
    REPEAT_COUNT = positive integer or ALL          (optional)
    OUTPUT = file                                   (optional)
    STATUS = status variable                        (optional)
```

DISPLAY_MEMORY (DISM)

Parameters: address

Specifies the memory location to be displayed. The memory location is specified by one or more of the following address parameters:

SECTION = name or keyword
MODULE = name
ADDRESS = rsssoooooooo or ?svar or function

SECTION or SEC

Identifies the memory section that contains the data to be displayed. When you use SECTION to specify an address, you must qualify it with the MODULE parameter. You can use the BYTE_OFFSET parameter to modify the starting address of memory to be displayed. Options are:

Omitted

Displays the memory address specified by the ADDRESS parameter.

Name

Displays the named memory section or common block. Must be a CYBIL program working storage section name or a common block name.

$BLANK

Displays the memory section that contains unnamed common.

$BINDING

Displays the memory section that contains the links to external procedures and the data of the module.

$LITERAL

Displays the memory section that contains the literal data of the module (for example, long constants).

$STATIC

Displays the memory section that contains the static variables not explicitly allocated to a named section of the module. Static variables are those not on the run-time stack.

MODULE or M

This parameter qualifies the SECTION parameter. It specifies the module that contains the data to be displayed. Options are:

Omitted

Displays the memory address specified by the ADDRESS parameter.

Name

Displays the data in the named module.

Source	Parameter Dependency
BASIC	MODULE is the main program referred to as $MAIN. $MAIN can reference internal subroutines and internal functions which are procedures within the module $MAIN. MODULE can also name an external subroutine or external function. An external subroutine and an external function can reference internal subroutines and internal functions which are procedures within that module.
C	MODULE names a C compilation unit (a C source file). The initial module is EM, a startup module. The EM module has no Debug tables so the default, $CURRENT, is not useful until program execution reaches a module with Debug tables. Global variables are in the module c_globals.
COBOL	MODULE names the program specified on the Progran-id statement.
CYBIL	MODULE names a module which may contain a program, procedure, or function.
FORTRAN	MODULE names a program, subroutine, function, or block data subprogram.
Pascal	The MODULE name is the program name.

ADDRESS or A

Specifies the address of the first byte of memory to be displayed. Options are:

Omitted

Indicates that the address is specified by the SECTION and MODULE parameters.

rsssoooooooo

where r is the ring number, sss is the segment number, and oooooooo is the offset from the beginning of the segment. The specified value cannot contain any blanks. You can use the BYTE_OFFSET parameter to modify the starting address of memory to be displayed.

?svar

Specifies the address indicated by the value of the SCL variable.

function

Specifies the address indicated by a Debug or SCL function.

BYTE_OFFSET or BO

Specifies the offset to the base address established by one of the address parameters. The address generated by adding BYTE_OFFSET to the base address must be within the memory block implied by the base address. The block size is the length of the section when the SECTION parameter is specified, and the length of the segment containing the machine address when the ADDRESS parameter is specified. Options are:

Omitted

Zero is used.

0 or integer

Adds the integer to the base address to form a new address. Unless a radix is explicitly specified, the integer is interpreted as hexadecimal.

BYTE_COUNT or BC

Specifies the number of bytes in the item to be displayed. Options are:

Omitted

Displays eight bytes.

Positive integer

Displays the specified number of bytes.

REPEAT_COUNT or RC

Specifies the number of memory area (items) of length BYTE_COUNT to be displayed. Options are:

Omitted

Displays one item.

Positive integer

Displays the specified number of items. The maximum amount of memory that can be displayed is limited to the block size implied by address (section length for SECTION and segment length for ADDRESS). If you specify a value that would cause the display to exceed this limit, all memory from the specified address to the end of the memory block is displayed.

ALL

Displays all items from the specified address to the end of the memory block.

OUTPUT or O

Specifies the file on which the displayed information is to be written. Options are:

Omitted

Writes to $OUTPUT.

File

Writes to the named file. You can position the file by appending a position indicator to the file name (.$BOI, .$ASIS, .$EOI).

STATUS

Specifies a variable to receive the return status of the command. Options are:

Omitted

The next command is processed if an error does not occur. The return status is output to $RESPONSE (and to the Debug output file if $RESPONSE is connected to that file) if an error does occur. $RESPONSE is normally connected during interactive debugging.

Status variable

The named variable receives the return status.

Remarks:

- DISPLAY_MEMORY allows you to debug your program even when compiler-generated symbol tables are not available, and to display memory areas that do not correspond to program identifiers. Each display line shows the memory contents in hexadecimal and ASCII formats; the relative byte offset from the initial address is also shown.

- The compiler-generated attributes list shows the section name and offset for all variables. You can reference static variables by specifying section name and byte offset. You can reference variables on the stack by specifying the machine address of the stack frame and byte offset of the variable. You can obtain the address of the stack frame of the procedure executing when Debug got control by displaying register A1 (refer to the DISPLAY_REGISTER command description in this chapter). You can obtain the address of other stack frames by displaying the save area of the desired stack frame using the DISPLAY_STACK_FRAME command and obtaining the value of register A1 from that stack frame. You can also use the DISPLAY_PROGRAM_VALUE command to display program variables when symbol tables are available.

Examples: The following example displays the first two bytes of the literal memory section for module MOD1:

```
DB/display_memory section=$literal module=mod1 ..
DB../byte_count=2
STARTING ADDRESS:  B 049 00000000
00000000 304A 00                  0J
```

The following example displays the first 32 bytes of the memory section DATA1 for module MOD2 as separate items:

```
DB/display_memory sec=data1 module=mod2 rc=4

STARTING ADDRESS:  B 04A 00000000
00000000 4001 8000 0000 0000 @???????
00000008 4002 8000 0000 0000 @???????
00000010 4002 C000 0000 0000 @???????
00000018 4003 8000 0000 0000 @???????
```

The following example displays the first 200 bytes of memory starting from the specified address:

```
DB/dism a=b02400000224 bo=8 rc=25
```

DISPLAY_PROGRAM_VALUE (DISPV)

Purpose: Displays the values of named program variables and constants as they are written in the source program. The subordinated record and field values of complex data structures are indented to show the organization of the data structure.

Format:

```
DISPLAY_PROGRAM_VALUE or
DISPV
    NAME = list of name or $ALL                    (required)
    MODULE = name                                  (optional)
    PROCEDURE = name                               (optional)
    RECURSION_LEVEL = positive integer             (optional)
    RECURSION_DIRECTION = keyword                  (optional)
    TYPE = keyword                                 (optional)
    VARIANT_SELECTION = name, integer,
      boolean, string, or $FIRST                   (optional)
    NAME_OPTION = keyword                          (optional)
    SCOPE = keyword                                (optional)
    SECTION = name                                 (optional)
    OUTPUT = file                                  (optional)
    STATUS = status variable                       (optional)
```

Parameters: NAME or N

Simple variable names, subscripted names (subscripts can be constants or variables, but not expressions), field references and pointer dereferences can be specified. SCL string, integer, and boolean variables can be used as aliases for program variable names. To do this, assign the SCL string variable to a string containing the identifier. Then use the SCL variable preceded by a question mark (?) as the value for the name. Options are:

Variable

Specifies a list of names of program variables or data structures to be displayed. The value of the named identifier is displayed. The MODULE and PROCEDURE parameters can be specified to qualify the name.

$ALL

Displays all the values of the variables in the procedure defined by the PROCEDURE parameter.

Source	Parameter Dependency
BASIC	Reference substrings by MID$(string_var_name, first_char_position, length) and string_var_name(first_char_position:last_char_position) formats where first_char_position, last_char_position, and length are positive integer values.
COBOL	Reference modification is supported. Data_names with qualified subscripts are not supported. However, a qualified sub script can be displayed first then the displayed value may be substituted for the subscript reference. To display qualified data_names, omit the NAME parameter. When the NAME is not specified, an ENTER_NAME prompt is issued to the current DEBUG_INPUT file. ENTER_NAME recognizes the COBOL qualifiers OF and IN and permits blanks around tokens, such a ´(´, ´)´, and ´,´.
FORTRAN Version 1	Extensible common blocks can be specified. Assumed-size arrays cannot be specified by name only; you must also specify the array element.
FORTRAN Version 2	Assumed-size arrays cannot be specified by name only; you must also specify an array element or array section. In some cases, arrays of length one are treated as assumed-size arrays and are subject to the same naming conventions.

MODULE or M

This parameter qualifies the NAME parameter. It specifies the name of the module containing the named program variable or data structure specified by the NAME parameter. Options are:

Omitted

If the MODULE parameter is not specified, the module that is executing when Debug gains control is used unless the Debug CHANGE_DEFAULTS command has specified another MODULE value. The default module can be changed to the module specified by the CHANGE_DEFAULTS MODULE parameter.

Name

Specifies the name of the module that contains the name of the program variable or data structure specified by the NAME parameter.

Source	Parameter Dependency
BASIC	MODULE is the main program referred to as $MAIN. $MAIN can reference internal subroutines and internal functions which are procedures within the module $MAIN. MODULE can also name an external subroutine or external function. An external subroutine and an external function can reference internal subroutines and internal functions which are procedures within that module.
C	MODULE names a C compilation unit (a C source file). The initial module is EM, a startup module. The EM module has no Debug tables so the default, $CURRENT, is not useful until program execution reaches a module with Debug tables. Global variables are in the module c_globals.
COBOL	MODULE names the program specified on the Program-id statement. Since MODULE and PROCEDURE parameters name the same program, both parameters should be assigned the same name or only one parameter should be specified.
CYBIL	MODULE names a module which may contain a program, procedure, or function.
FORTRAN	MODULE names a program, subroutine, function, or block data subprogram. Since the MODULE and PROCEDURE parameters name the same program unit, both parameters should be assigned the same name or only one parameter should be specified.
Pascal	The MODULE name is the program name.

PROCEDURE or P

This parameter qualifies the NAME parameter. It specifies the name of the procedure containing the named program variable or data structure specified by the NAME parameter. If a procedure is inactive (not in an active call chain), its variables cannot be displayed by DISPV because the procedure has no stack frame and therefore no existent automatic variables. Options are:

Omitted

The procedure that is executing when Debug gains control is used unless the Debug CHANGE_DEFAULTS command has specified another PROCEDURE value. The default procedure can be changed to the procedure specified by the CHANGE_DEFAULTS PROCEDURE parameter.

Name

Specifies the name of the procedure that contains the name of the program variable or data structure specified by the NAME parameter.

Source	Parameter Dependency
BASIC	PROCEDURE names internal subroutines and internal functions within a module.
C	PROCEDURE names a function or a block within a function.
COBOL	PROCEDURE names a program specified on the Program-id statement. Since MODULE and PROCEDURE parameters name the same program, both parameters should be assigned the same name or only one parameter should be specified.
CYBIL	PROCEDURE names a program, procedure, or function in the module specified by the MODULE parameter.
FORTRAN	PROCEDURE names a program, subroutine, function, or block data subprogram. Since MODULE and PROCEDURE parameters name the same program unit, both parameters should be assigned the same name or only one parameter should be specified.
Pascal	PROCEDURE names a program, procedure or function name.

RECURSION_LEVEL or RL

Specifies the recursion level of the procedure specified by the PROCEDURE parameter. The value of the data item specified by the NAME parameter and known to this recursive level of the procedure is displayed. Recursion does not affect static or global variables. Options are:

Omitted

The default value is 1.

Positive integer

The specified value must be a positive integer greater than 0. If the DISPV RECURSION_DIRECTION parameter specifies BACKWARD, 1 is the most recent invocation of the procedure, 2 is its predecessor, and so on (backward count from the most recent call). If the RECURSION_DIRECTION parameter specifies FORWARD, 1 is the first invocation of the procedure, 2 is the procedure called by the first invocation, and so on (forward count from the first call).

Source	Parameter Dependency
COBOL FORTRAN	RECURSION_LEVEL is not supported. Omit this parameter.

RECURSION_DIRECTION or RD

Specifies whether the RECURSION_LEVEL is counted forward from the first call or backward from the most recent call. Recursion does not affect static or global variables. Options are:

Omitted

The default value is BACKWARD.

FORWARD

A RECURSION_LEVEL of 1 specifies the first call to the procedure, 2 specifies the second call, and so on.

BACKWARD

A RECURSION_LEVEL of 1 specifies the most recent call to the procedure, 2 specifies its predecessor, and so on.

Source	Parameter Dependency
COBOL FORTRAN	RECURSION_DIRECTION is not supported. Omit this parameter.

TYPE or T

Specifies how the data is to be represented for the value of the NAME parameter variable.

Omitted

The variable name and its value are displayed as the data is defined in the source program. Integers are displayed as decimal integers, strings are displayed as ASCII characters, booleans are displayed as TRUE or FALSE, pointers to procedures are displayed as the procedure name, other pointers are displayed as HEX addresses, records are displayed as a collection of elementary items each in its natural format, and so on.

HEX or H

Displays the value of the NAME parameter variable in hexadecimal format.

INTEGER or I

Interprets the value of the NAME parameter variable as an integer number and displays it as a decimal integer. The data must be from 1 through 8 bytes long. Each element in an array is displayed as a separate integer. Character, complex, and double precision variables cannot be displayed in integer format.

REAL or R

Interprets the value of the NAME parameter variable as a floating-point number. The data must be 8 bytes long. Each element in an array is displayed as a floating-point number. Character, complex and double precision variables cannot be displayed in real format.

Source	Parameter Dependency
BASIC COBOL	TYPE = INTEGER and TYPE = REAL are not supported. Omit these options.
FORTRAN	NAMELIST data cannot be displayed with TYPE = HEX.

VARIANT_SELECTION or VS

Specifies the value of the tag field for the variant part of a tagless record. The tag type specified by the CASE statement must be of the type specified by the VARIANT_SELECTION value. This parameter is used to display an entire tagless record. It is used mostly to display an array of tagless records. Options are:

Integer

Specifies the value of the tag field (selector value) as an integer. If the integer is outside the range defined in the CASE statement, an error message is issued.

Boolean

Specifies the value of the tag field (selector value) as a BOOLEAN. The value is compared to the tag type specified in the CASE statement. If the type is not BOOLEAN, an informative message is issued. If the type is BOOLEAN, the variant part of the record is interpreted based on the value specified by the VARIANT_SELECTION parameter.

Name

Specifies the value of the tag field (selector value) as a name. If the name is not found in the symbol table or if it is found, but it is not a valid ordinal for the tag type, an informative message is issued. If the name is a valid ordinal for the selector value of the record variant, the value of that ordinal from the symbol table is used to display the variant.

String

Specifies the value of the tag field (selector value) as a one-character string. If the type is not character, an informative message is issued. If the type is character, the variant is interpreted based on the value of the VARIANT_SELECTION parameter.

$FIRST

Specifies that the first valid tag field value found in the symbol table is used as the value of the tag field. $FIRST can be specified if you want to see the variant field, but are not sure how it was defined.

Source	Parameter Dependency
BASIC C COBOL FORTRAN	VARIANT_SELECTION is not supported. Omit this parameter.

NAME_OPTION or NO

Qualifies the identifier(s) given for the NAME parameter. Options are one or more of the following:

Omitted

There is no default for the NAME_OPTION parameter when a single identifier is specified for the NAME parameter. If $ALL is specified for the NAME parameter, the default for the NAME_OPTION parameter is VARIABLES.

CONSTANTS or C

The idenfifier in the source program must be a constant.

VARIABLES or V

The idenfifier in the source program must be a variable name.

PARAMETERS or P

The identifier in the source program must be a variable that was passed as a parameter to the default procedure or the procedure specified by the PROCEDURE parameter.

ALL

The identifier in the source program can be either a constant or a variable.

Source	Parameter Dependency
BASIC C COBOL	NAME_OPTION = CONSTANTS is not supported. Omit this option.

Invalid Parameter Combinations

NAME_OPTION=PARAMETERS cannot be used with the SECTION parameter. If the DISPV parameters are specified positionally and this parameter is omitted, its position must be indicated by a comma.

SCOPE or SCO

Determines the type of search for identifiers specified by the NAME parameter. Options are:

GLOBAL or G

The value of the NAME parameter must reference identifier(s) known outside the defining module. The Entry_Point_Table is searched to locate the identifier(s).

Invalid Parameter Combinations

GLOBAL cannot be used with the MODULE, PROCEDURE, RECURSION_LEVEL, and RECURSION_DIRECTION parameters or with NAME_OPTION = PARAMETERS or with NAME = $ALL.

MODULE or M

The value of the NAME parameter must reference identifier(s) defined at the outermost level of a module.

Source	Parameter Dependency
BASIC C COBOL FORTRAN Pascal	SCOPE = MODULE is not supported. Omit this option.
CYBIL	SCOPE = MODULE is the identifiers defined at the module level outside all procedures.

LOCAL or L

The identifier(s) referenced by the NAME parameter must be defined in the procedure specified by the PROCEDURE parameter or by default.

Source	Parameter Dependency
BASIC	SCOPE = LOCAL is the formal parameters and the BASIC internal variables in an internal subroutine.
C FORTRAN	SCOPE = LOCAL is not supported. Omit this option.
COBOL	SCOPE = LOCAL is the data names defined in a program.
CYBIL Pascal	SCOPE = LOCAL is the identifiers defined in the procedure, not at an outer level.

SECTION or SEC

Displays a group of identifiers by specifying the section where they are stored. This parameter is valid only when the value of the NAME parameter is $ALL. Options are:

Omitted

Displays all the values of all the variables in the procedure defined by the PROCEDURE parameter.

Name

Displays the named memory section or common block.

$BLANK

Displays the memory that contains unnamed common.

$LITERAL

Displays the memory section that contains the literal data of the module (for example, long constants).

$STATIC

Displays the memory section that contains the static variables not explicitly allocated to a named section of the module. Static variables are those not on the run-time stack.

Source	Parameter Dependency
COBOL	SECTION is not supported. Omit this parameter.

Invalid Parameter Combinations

The SECTION parameter cannot be used with the RECURSION_LEVEL and RECURSION_DIRECTION parameters or with NAME_OPTION = PARAMETERS or SCOPE = GLOBAL.

OUTPUT or O

Specifies the file where the display information is to be written.

Omitted

Writes to the current Debug output file.

File

Writes to the named file. You can position the file by appending a position indicator to the file name (.$BOI, .$ASIS, .$EOI).

STATUS

Specifies a variable to receive the return status of the command. Options are:

Omitted

The next command is processed if an error does not occur. The return status is output to $RESPONSE (and to the Debug output file if $RESPONSE is connected to that file) if an error does occur. $RESPONSE is normally connected during interactive debugging.

Status variable

The specified variable receives the return status.

Examples: The following command displays the value of variable VAR:

```
DB/display_program_value var
var = TRUE
```

The following command displays the value of array TABLE in procedure GGG of module FFF_DD:

```
DB/dispv name=table module=FFF_DD p=GGG
table = 'W' 'O' 'R' 'L' 'D'
```

The following command displays the value of variable RECORD5 to output file FILE1; the return status is written to variable STATVAR.

```
DB/dispv record5 output=file1 status=statvar
```

The following command displays the values of all variables in the current procedure and module:

```
DB/dispv $all
-- DISPLAY OF ALL VARIABLES IN MOD2
a = 4.
b = 5.
c = 6.
table = 'W' 'O' 'R' 'L' 'D'
x = 1.
y = 2.
z = 3.
```

The following command displays all global variable values of a C program:

```
DB/dispv n=$all m=c_globals
```

The following command displays the values of all variables in common block section DATA1:

```
DB/dispv $all name_option=variable section=data1
-- DISPLAY OF ALL VARIABLES IN MOD1
a = 4.
b = 5.
c = 6.
x = 1.
y = 2.
z = 3.
```

The following examples refer to the FORTRAN definitions below:

```
    COMMON /BLK/ DVAL, RVAL, IVAL, ZVAL
    DATA DVAL, RVAL, IVAL, ZVAL/20.0D+0, 3.45E+01, 30,
   +(+20.0,20.3)/
```

To display the value of DVAL:

```
DB/display_program_value name=dval
dval = 20.
```

To display the value of RVAL in integer format:

```
DB/dispv name=rval type=integer
rval = 4613526600892284928
```

To display the value of IVAL:

```
DB/display_program_value ival
ival = 30
```

To display the value of ZVAL:

```
DB/dispv name=zval
zval = 20.
```

DISPLAY_REGISTER (DISR)

Purpose: Displays the contents of the P, A, or X registers that are associated with the procedure or function executing when Debug gained control.

Format:

```
DISPLAY_REGISTER or
DISPLAY_REGISTERS or
DISR
    KIND = list of keyword                          (optional)
    NUMBER = integer or list of integers or
             ALL                                    (optional)
    TYPE = keyword                                  (optional)
    OUTPUT = file                                   (optional)
    STATUS = status variable                        (optional)
```

Parameters: KIND or K

Specifies the register to be displayed. Options are:

Omitted

Same as KIND = ALL.

P

Displays the P register.

A

Displays the A registers.

X

Displays the X registers.

ALL

Displays all registers.

NUMBER or N

Indicates which A or X registers specified by the KIND parameter are displayed. This parameter is ignored if KIND=P because there is only one P register. Options are:

Omitted

Displays the zero register.

Integer or range of integers

Displays a set of registers. Integer can be 0 through 15.

ALL

Displays all A (if KIND = A) or X (if KIND = X) or both sets of (if KIND = ALL) registers.

TYPE or T

Specifies the format in which the register is to be displayed. Options are:

Omitted

Same as TYPE = HEX.

ASCII or A (X registers only)

The register is displayed as an ASCII string.

HEX or H

HEX is used for string values and INTEGER is used for numeric values. The A and P registers are formatted as a PVA: R sss 00000000. The X registers are formatted as two hexadecimal half words: HHHHHHHH HHHHHHHH.

INTEGER or I (X registers only)

The register is displayed as a decimal integer.

REAL or R (X registers only)

The register is displayed as a real number.

OUTPUT or O

Specifies the file where the display information is to be written.

Omitted

Writes to the current Debug output file.

File

Writes to the named file. You can position the file by appending a position indicator to the file name (.$BOI, .$ASIS, .$EOI).

STATUS

Specifies a variable to receive the return status of the command. Options are:

Omitted

The next command is processed if an error does not occur. The return status is output to $RESPONSE (and to the Debug output file if $RESPONSE is connected to that file) if an error does occur. $RESPONSE is normally connected during interactive debugging.

Status variable

The named variable receives the return status.

Examples: The following command displays the contents of the P register in hexadecimal format:

```
DB/display_register p
P=B 031 00000040
```

The following command displays the contents of the A8 register in hexadecimal format:

```
DB/display_register kind=a number=8 type=hex
A8=B 047 00000B10
```

The following command displays the contents of the X4, X5, X6, X7, X8, X9, and X10 registers in hexadecimal format:

```
DB/disr kind=x number=4..10

X4=00000000 10000000
X5=00000000 00000008
X6=00000000 0000000D
X7=00000000 0000001D
X8=00000000 00000000
X9=00000000 00000008
XA=00000000 00000300
```

DISPLAY_STACK_FRAME (DISSF)

Purpose: Displays the contents of one or more stack frames. Values are displayed in hexadecimal.

Format:

```
DISPLAY_STACK_FRAME or
DISPLAY_STACK_FRAMES or
DISSF
    COUNT = integer or ALL                      (optional)
    START = positive integer                    (optional)
    DISPLAY_OPTION = list of keyword            (optional)
    OUTPUT = file                               (optional)
    STATUS = status variable                    (optional)
```

Parameters: COUNT or C

Specifies the number of stack frames to be displayed. Options are:

Omitted

Displays one stack frame.

Positive integer

Displays the specified number of stack frames; if the integer is greater than the number of existing stack frames, all stack frames are displayed.

ALL

Displays all stack frames.

START or S

Specifies the stack frame on the stack to be displayed first. Thus, it is possible to skip the most recent stack frames. Options are:

Omitted

Displays the most recent stack frame.

Positive integer

Displays the specified stack frame; 1 represents the most recent stack frame, 2 represents the predecessor of the most recent stack frame, and so forth.

An informative message is issued if the specified number of stack frames is greater than the actual number of stack frames.

Source	Parameter Dependency
C	Each function has its own stack frame, but all blocks in a function share the same stack frame.

DISPLAY_OPTION or DISPLAY_OPTIONS or DO

Specifies the area of the stack frames to be displayed. Options are one of the following:

Omitted

Same as DISPLAY_OPTION = ALL.

AUTO or A

Displays the area that contains the automatic (dynamically allocated) variables of the procedure.

SAVE or S

Displays the area that contains a copy of the registers of the procedure as they existed at the time of a call or trap.

ALL

Displays both the automatic and save areas.

OUTPUT or O

Specifies the file on which the stack frame information is to be written. Options are:

Omitted

Writes to the current Debug output file.

File

Writes to the named file. You can position the file by appending a position indicator to the file name (.$BOI, .$ASIS, .$EOI).

STATUS

Specifies a variable to receive the return status of the command. Options are:

Omitted

The next command is processed if an error does not occur. The return status is output to $RESPONSE (and to the Debug output file if $RESPONSE is connected to that file) if an error does occur. $RESPONSE is normally connected during interactive debugging.

Status variable

The named variable receives the return status.

Examples: The command below displays the save area of the most recent stack frame:

```
DB/display_stack_frame display_option=save

SAVE AREA
P=B 035 00000026         VMID=0
UM=FFF7      UCR=0040    MCR=0000

A0=B 032 00000460        A1=B 032 00000408
A2=B 032 000003C0        A3=B 030 00000000
A4=B 032 00000390        A5=B 02F 00000020
A6=B 02E 00000000        A7=B 02F 00000000
A8=B 00F 00000018        A9=B 032 00000630
AA=B 032 00000A30        AB=F FFF 80000000
AC=F FFF 80000000        AD=B 032 00001058
AE=F FFF 80000000        AF=B 00B 000557F8

X0=0000B01D  00020060       X1=00000000  00000000
X2=0000FFFF  80000000       X3=000007FF  FFFFFFFF

X4=00000000  10000000       X5=00000000  00000008
X6=00000000  0000000D       X7=00000000  0000001D
X8=00000000  00000000       X9=00000000  00000008
XA=00000000  00000300       XB=00000000  00000000
XC=00000000  00000001       XD=00000000  00000022
XE=00000000  00010040       XF=00000000  0000004E
```

The following command displays the automatic and save areas of three stack frames beginning with the second most recent one:

```
DB/display_stack_frames count=3 start=2
```

The following command displays the automatic and save areas of the most recent stack frame:

```
DB/dissf  count=1

STACK FRAME 001        SEGMENT=032
00000000    00000000  00000000
00000008    00000000  00000000
00000010    30300000  00C0FFFF    00
00000018    80000000  00000000
00000020    B032B031  00000000    2 1
00000028    0000B01D  0009B346          F
00000030    0000B032  00000430      2   0
00000038    0040B032  00000400    @ 2
00000040    FF77B032  000003C0    w 2
00000048    FFFCB01B  00020F78          x
00000050    0000B032  00000390      2

SAVE AREA

P=B 035 00000026         VMID=0
UM=FFF7     UCR=0040     MCR=0000


A0=B 032 00000460        A1=B 032 00000408
A2=B 032 000003C0        A3=B 030 00000000
A4=B 032 00000390        A5=B 02F 00000020
A6=B 02E 00000000        A7=B 02F 00000000
A8=B 00F 00000018        A9=B 032 00000630
AA=B 032 00000A30        AB=F FFF 80000000
AC=F FFF 80000000        AD=B 032 00001058
AE=F FFF 80000000        AF=B 00B 000557F8

X0=0000B01D  00020060       X1=00000000  00000000
X2=0000FFFF  80000000       X3=000007FF  FFFFFFFF

X4=00000000  10000000       X5=00000000  00000008
X6=00000000  0000000D       X7=00000000  0000001D
X8=00000000  00000000       X9=00000000  00000008
XA=00000000  00000300       XB=00000000  00000000
XC=00000000  00000001       XD=00000000  00000022
XE=00000000  00010040       XF=00000000  0000004E
```

QUIT (QUI)

Purpose: Ends the Debug session and returns control to the activity that was in use before the Debug session began. The session is terminated immediately; the program is not executed to completion.

Format: QUIT or
QUI
STATUS = status variable (optional)

Parameter: STATUS

Specifies a variable to receive the return status of the command. Options are:

Omitted

The next command is processed if an error does not occur. The return status is output to $RESPONSE (and to the Debug_output file if $RESPONSE is connected to that file) if an error does occur. $RESPONSE is normally connected during interactive debugging.

Status variable

The named variable receives the return status.

RUN

Purpose: Begins or resumes program execution once Debug has gained control. Execution continues until Debug again gains control. If the program has run to completion, entering the RUN command ends the program and returns control to the activity in use before the Debug session began.

Format:

```
RUN
    STATUS = status variable                    (optional)
```

Parameter: STATUS

Specifies a variable to receive the return status of the command. Options are:

Omitted

The next command is processed if an error does not occur. The return status is output to $RESPONSE (and to the Debug output file if $RESPONSE is connected to that file) if an error does occur. $RESPONSE is normally connected during interactive debugging.

Status variable

The named variable receives the return status.

Examples: The following example initiates execution of a program. Execution continues until a break is encountered.

```
DB/run
 --DEBUG:  break DBB$2, execution at M=DT L=6
```

SET_BREAK (SETB)

Purpose: Defines a break. You can specify one or more events and the location at which Debug is to take control. When the specified event occurs, program execution is suspended and a message informs you which break occurred. At this point, you can enter another Debug command or an SCL command that can be processed by the operating system.

Format:

```
SET_BREAK or
SET_BREAKS or
SETB
    BREAK = name                                    (optional)
    EVENT = list of keywords                        (optional)
    address                                         (required)
        address can be one or more of the following:
            LINE = integer
            STATEMENT = positive integer
            STATEMENT_LABEL = integer or name
            NAME = name
            SECTION = name or keyword
            MODULE = name
            PROCEDURE = name
            ENTRY_POINT = name
            ADDRESS = rsssoooooooo or ?svar or function
    BYTE_OFFSET = 0 or integer                      (optional)
    BYTE_COUNT = positive integer                   (optional)
    COMMAND = string                                (optional)
    STATUS = status variable                        (optional)
```

Parameters: BREAK or B

Specifies the name of the break. This name is used to reference the break definition in the DISPLAY_BREAK and DELETE_BREAK commands. This name is displayed in the break report message when the break occurs. A break cannot be named ALL. The break name must not contain the $ character. Options are:

Omitted

Debug assigns a unique name, and identifies the assignment to the user. The assignment is written to the default debug output file.

Name

The break is assigned the specified name.

For Better Performance

Because execution takes longer when there are breaks set, you should delete a break as soon as it is no longer needed (see the DELETE_BREAK command described in this chapter).

EVENT or EVENTS or E

Specifies the events that must occur for the break to occur. If you specify more than one event, the break occurs if any of the events occur. Options are one or more of the following:

Omitted

Same as EXECUTION.

ARITHMETIC_OVERFLOW or AO

Breaks when an arithmetic overflow occurs on an instruction in the specified address range. The P register points to the instruction that caused the overflow.

ARITHMETIC_SIGNIFICANCE or AS

Breaks when arithmetic significance is lost on an instruction in the specified address range. The P register points to the instruction that caused the loss of significance.

BRANCH or B

Breaks before either a branch to or a return from any location in the specified address range.

CALL or C

Breaks before a subprogram call occurs to any address in the specified address range.

DIVIDE_FAULT or DF

Breaks when division by zero occurs in an instruction in the specified address range. The P register points to the instruction that caused the division by zero.

EXECUTION or E

Breaks before the instruction in the specified address range is executed.

To set an execution break, the break location must be an executable statement. For example, an assignment is executable, but a data declaration is not.

EXPONENT_OVERFLOW or EO

Breaks when an exponent overflow occurs in an instruction in the specified address range. The P register points to the instruction following the one that caused the overflow.

EXPONENT_UNDERFLOW or EU

Breaks when an exponent underflow occurs in an instruction in the specified address range. The P register points to the instruction following the one that caused the underflow.

FLOATING_POINT_INDEFINITE or FPI

Breaks when the result of a floating-point operation is indefinite in an instruction in the specified address range. The P register points to the instruction following the one that caused the results to be indefinite.

FLOATING_POINT_SIGNIFICANCE or FPS

Breaks when significance is lost during a floating-point operation in an instruction in the specified address range. The P register points to the instruction following the one that caused the loss of significance. This event does not occur unless your program sets the floating-point loss-of-significance bit in the user mask register.

INVALID_BDP_DATA or IBD

Breaks when a business data processing (BDP) instruction fault occurs in an instruction in the specified address range. The P register points to the instruction that caused the fault. The BDP instructions are described in the Virtual State Hardware reference manual.

READ or R

Breaks before a read occurs from the specified address range. The break occurs only if the first byte of the item to be read is within the address range.

READ_NEXT_INSTRUCTION or RNI

Breaks before the instruction in the specified address range is executed.

WRITE or W

Breaks before a write occurs into the specified address range. The break occurs only if the first byte of the item to be written is within the address range.

Source	Parameter Dependency
BASIC	The ARITHMETIC_SIGNIFICANCE, FLOATING_POINT_INDEFINITE, and INVALID_BDP_DATA options are not supported. Omit these options. READ or WRITE break cannot be set on a variable of zero length.
CYBIL	Breaks cannot be set on ELSE or IFEND statements.
FORTRAN	A READ or WRITE break cannot be set on an assumed-size array.
Pascal	A READ or WRITE break cannot be set on a variable of zero length.

address

Specifies the location at which the break occurs. For the break to occur, the specified event must occur within the range defined by the address parameters. All address parameters are interpreted as a single address. You can use the BYTE_COUNT and BYTE_OFFSET parameters to specify an address range. Options are:

Omitted

Indicates an address range of one byte.

One or more of the following parameters:

LINE = integer
STATEMENT = positive integer
STATEMENT_LABEL = integer or name
NAME = name
SECTION = name or keyword
MODULE = name
PROCEDURE = name
ENTRY_POINT = name
ADDRESS = rsssoooooooo or ?svar or function

LINE or L

Line at which Debug gains control. The line number must exist in the current default module (that was explicitly set with the CHANGE_DEFAULTS command or was executing when Debug gained control) unless the MODULE parameter is also specified. Options are:

Omitted

Indicates that the break address is specified by another parameter.

Integer

Indicates the line number on which the break occurs in the specified address range.

You can use BYTE_OFFSET and BYTE_COUNT to modify this parameter.

Not all lines of a program can be referenced. Only executable statements that begin on a separate line can be referenced. A line that contains the continuation of a statement cannot be referenced. All lines in an IF block can be referenced.

If the source code was compiled at a high optimization level, you will not be able to reference some lines because they have been moved or deleted.

Source	Parameter Dependency
BASIC	LINE specifies the line number that precedes the line on the source listing; it is not a BASIC statement label.

STATEMENT or S

Used with the LINE parameter to specify a statement following the first statement in a multi-statement line. Options are:

Omitted

The first statement in the line is used.

Positive integer

Specifies which statement in the line at which the break is set.

Source	Parameter Dependency
C COBOL FORTRAN	These languages do not support multiple statements per line, therefore, this parameter should be omitted; a value specified is ignored.

STATEMENT_LABEL or SL

Specifies a statement label on which a break can be set. Options are:

Omitted

Indicates that the break address is specified by another parameter.

Integer

Indicates the statement label where the break is set. Unless the MODULE or PROCEDURE parameter is also specified, the statement label must exist in the current default module.

The statement label is assumed to represent the first byte of code contained in the source line represented by the statement label. The calculated byte address and length can be modified by the BYTE_OFFSET and BYTE_COUNT parameters.

Name

Indicates the statement label where the break is set.

Source	Parameter Dependency
BASIC	A statement label is an integer and is part of the source program; it is not a line number added during compilation. BASIC does not reference statements as names, therefore, the name option is not valid.
C	STATEMENT_LABEL is not supported. Omit this option.
COBOL	A statement label is a Cobol-paragraph statement or Cobol-section statement. STATEMENT_LABEL can be qualified by the MODULE or SECTION parameters.
CYBIL	A statement label is the name which is enclosed in slashes in the source listing such as, /name/.
FORTRAN Pascal	A statement label is an integer. These languages do not reference statements as names, therefore, the name option is not valid.

Invalid Parameter Combinations

The STATEMENT_LABEL parameter cannot be used with the STATEMENT parameter.

NAME or N

Specifies a variable on which a READ or WRITE break can be set. Options are:

Omitted

Indicates that the break address is specified by another parameter.

Variable

Indicates the variable where the READ or WRITE break is set. Simple unsubscripted variable names, subscripted names (subscripts can be constants or variables, but not expressions), field references, and pointer dereferences can be specified. When specifying NAME = variable, you must also specify EVENT = READ or EVENT = WRITE. The MODULE and PROCEDURE parameters can be specified to qualify the name.

SECTION or SEC

Identifies a memory section.

You can use the BYTE_OFFSET and BYTE_COUNT parameters to modify this parameter.

The section must exist for the current default module (that was explicitly set with the CHANGE_DEFAULTS command or was executing when Debug gained control) unless the MODULE parameter is also specified.

The SECTION parameter cannot be specified for modules that are components of a bound module unless the section is a common block (see Addressing Bound Modules in chapter 6). Options are:

Omitted

Indicates that the break address is specified by another address parameter.

Name

Displays the named memory section or common block. Must be a CYBIL program working storage section, a common block, or a FORTRAN extensible common block.

$BINDING

Identifies the memory section that contains the links to external procedures and the data of the module.

$BLANK

Identifies the memory section that contains unnamed common.

$LITERAL

Identifies the memory section that contains the literal data (for example, long constants) of the module.

$STATIC

Identifies the memory section that contains the static (not on the run-time stack) variables not explicitly allocated to a named section of the module.

CYB$DEFAULT_HEAP

Identifies the memory section containing the default heap.

Source	Parameter Dependency
BASIC COBOL	The $BLANK and CYB$DEFAULT_HEAP parameter options are not supported. Omit these options.
C	The SECTION parameter is not supported. Omit this parameter.
COBOL	The $BLANK and CYB$DEFAULT_HEAP parameter options are not supported. Omit these options. If the STATEMENT_LABEL parameter is specified, the SECTION parameter can be used to qualify the statement label.
CYBIL	The $BLANK parameter option is not supported. Omit this option. SECTION = name can also specify the name of the working storage section.
FORTRAN	The CYB$DEFAULT_HEAP parameter option is not supported. Omit this option. SECTION = name can also specify an extensible common block. To identify an extensible common block, you can also specify the EVENT = READ or EVENT = WRITE parameter.
Pascal	The $BLANK and CYB$DEFAULT_HEAP parameter options are not supported. Omit these options.

MODULE or M

Specifies an address (the first byte of the first code section of the module) or qualification of another address parameter. Options are:

Omitted

The current default module (that was explicitly set with the CHANGE_DEFAULTS command or was executing when Debug gained control) is used.

Name

The named module is used.

If used to specify an address, the BYTE_OFFSET and BYTE_COUNT parameters can be used to modify this parameter.

If MODULE is used with the LINE, SECTION, or PROCEDURE address parameters, the MODULE parameter identifies the module containing the line, section, or procedure.

Source	Parameter Dependency
BASIC	MODULE is the main program referred to as $MAIN. $MAIN can reference internal subroutines and internal functions which are procedures within the module $MAIN. MODULE can also name an external subroutine or external function. An external subroutine and an external function can reference internal subroutines and internal functions which are procedures within that module.
C	MODULE names a C compilation unit (a C source file). The initial module is EM, a startup module. The EM module has no Debug tables so the default, $CURRENT, is not useful until program execution reaches a module with Debug tables. Global variables are in the module c_globals.

COBOL — MODULE names the program specified on the Program-id statement. Since MODULE and PROCEDURE parameters name the same program, both parameters should be assigned the same name or only one parameter should be specified.

CYBIL — MODULE names a module which contains a program, procedure, or function.

If the module contains more than one section code, MODULE refers to the first one.

FORTRAN — MODULE names a program, subroutine, function, or block data subprogram. Since MODULE and PROCEDURE parameters name the same program unit, both parameters should be assigned the same name or only one parameter should be specified.

Pascal — MODULE is the program name.

PROCEDURE or P

Specifies an address (the first byte of the first code section of the procedure) or qualification of another address parameter. A procedure is a program unit subordinate to a module. Options are:

Omitted

The current default procedure (that was explicitly set with the CHANGE_DEFAULTS command or was executing when Debug gained control) is used. Otherwise, the break address is specified by the another address parameter.

Name

The named procedure is used as the break address.

If used to specify an address, the BYTE_OFFSET and BYTE_COUNT parameters can be used to modify this parameter.

Source	Parameter Dependency
BASIC	PROCEDURE names internal subroutines and internal functions within a module.
C	PROCEDURE names a function or a block within a function.
COBOL	PROCEDURE names a program specified on the Program-id statement. Since MODULE and PROCEDURE parameters name the same program, both parameters should be assigned the same name or only one parameter should be specified.
CYBIL	PROCEDURE names a program, procedure, or function in the module specified by the MODULE parameter.
FORTRAN	PROCEDURE names a program, subroutine, function, or block data subprogram. Since MODULE and PROCEDURE parameters name the same program unit, both parameters should be assigned the same name or only one parameter should be specified.
Pascal	PROCEDURE names a program, procedure or function name.

Invalid Parameter Combinations

You cannot specify the LINE or SECTION address parameters with the PROCEDURE parameter.

ENTRY_POINT or EP

Specifies an entry point expressed as a name known to the loader. ENTRY_POINT must be a program or subprogram name. See the Object Code Management manual for information on restrictions.

Omitted

Indicates that the break address is specified by another parameter.

Name

The specified name is used as the entry point name.

You can use the BYTE_OFFSET and BYTE_COUNT parameters to modify the ENTRY_POINT parameter.

Source Parameter Dependency

BASIC	ENTRY_POINT is an external procedure or external function.
COBOL	ENTRY_POINT must be an SCL name. Program-name is converted from the Program-id paragraph if the Identification Division by changing lowercase letters to uppercase letters, replacing hyphens (-) with the underscore (_), and adding the number sign (#) as a prefix if the first character is a digit.

Invalid Parameter Combinations

You cannot use other address parameters with the ENTRY_POINT parameter.

ADDRESS or A

Specifies the address of the first byte of memory at which the break is to occur. Options are:

Omitted

Indicates that the break address is specified by another parameter.

rsssoooooooo

where r is the ring number, sss is the segment number, and oooooooo is the offset from the beginning of the segment. You can use the BYTE_OFFSET parameter to modify the starting address of memory to be displayed. You can use the BYTE_OFFSET parameter to modify the ADDRESS.

?svar

Specifies the address indicated by the value of the SCL variable.

function

Specifies the address indicated by a Debug or SCL function.

Invalid Parameter Combinations

The LINE_STATEMENT and MODULE parameters cannot be used with this parameter.

BYTE_OFFSET or BO

Specifies the offset to the base address established by one of the address parameters. The address generated by adding BYTE_OFFSET to the base address must be within the memory block implied by the base address. The block size is the length of the section when the SECTION parameter is specified, and the length of the segment containing the machine address when the ADDRESS parameter is specified. Options are:

Omitted

Zero is used.

0 or positive integer

Adds the specified integer to the base address to form a new address. Unless a radix is explicitly specified, the integer is interpreted as hexadecimal.

Invalid Parameter Combinations

The BYTE_OFFSET parameter cannot be applied to an address defined by the NAME parameter.

BYTE_COUNT or BC

Specifies the number of bytes in the address range. Options are:

Omitted

The address range is one byte.

positive integer

The address range is the specified number of bytes.

Invalid Parameter Combinations

The BYTE_COUNT parameter cannot be used to change the number of bytes in the address range specified by the NAME Parameter.

COMMAND or COMMANDS or C

Specifies the string of commands to be executed by Debug, SCL, or any other active command processor when the break is honored. Options are:

Omitted

Indicates that no commands are associated with the break.

string

Specifies a string of commands to be used.

If a command in the string includes a quoted string, that string must be enclosed in two single apostrophes. After the commands in the string have been executed, commands are read from the current Debug input file unless the string contains a RUN command.

No break report message is issued before the commands in the string are executed. If you want a message to be displayed, include an SCL DISPLAY_VALUE command in the string. If an error is detected in one of the commands in the string, the break report message is issued, the error is reported, and commands are read from the Debug input file. The remaining commands in the string are not executed.

STATUS

Specifies a variable to receive the return status of the command. Options are:

Omitted

The next command is processed if an error does not occur. The return status is output to $RESPONSE (and to the Debug output file if $RESPONSE is connected to that file) if an error does occur. $RESPONSE is normally connected during interactive debugging.

status variable

The named variable receives the return status.

Remarks:

- Because execution takes longer when there are breaks set, you should delete a break as soon as it is no longer needed.

- EXECUTION and READ_NEXT_INSTRUCTION breaks cannot be set for all lines that can be referenced in a program, only those that are executable, except for BASIC programs where a break can be set on all lines.

- The maximum number of breaks Debug can handle is 64 per task. Of these, 32 may be of the types detected by Debug hardware (such as, read, write, call, branch, execution, read-next-instruction). Also, for every break you set, Debug may need to use one or more additional breaks internally. As a result, the actual maximum breaks you can set is not a fixed number. An error message is issued when a break cannot be set for the specified event.

- You cannot set overlapping breaks, that is, breaks for the same event that have overlapping address ranges specified. An error message is issued if this occurs.

- Debug gains control when the following events occur, even if you do not set a break for them:

  ARITHMETIC_OVERFLOW
  ARITHMETIC_SIGNIFICANCE
  DIVIDE_FAULT
  EXPONENT_OVERFLOW
  EXPONENT_UNDERFLOW
  FLOATING_POINT_INDEFINITE
  FLOATING_POINT_SIGNIFICANCE
  INVALID_BDP_DATA

Examples: The command below causes a break to occur when execution reaches line 10 of module PROG1:

```
DB/set_break break=b1 line=10 module=prog1
```

The following command causes a break to occur when a branch or return to line 40 (of the module executing when Debug gained control) occurs. (Debug assigns a unique name.):

```
DB/set_break event=branch line=40
-- Break name DBB$5 assigned to this break
```

The following command causes a break to occur prior to execution of line 10 (of the module executing when Debug gained control). (Debug assigns a unique name.):

```
DB/setb l=10
-- Break name DBB$6 assigned to this break
```

The following command causes a break to occur immediately before a WRITE event occurs on variable VAR in module TEST. (Debug assigns a unique name).

```
DB/setb name=var module=test event=write
-- Break name DBB$1 assigned to this break
```

The following command causes a break to occur prior to execution of line 6 (of module executing when Debug gained control). When the break is reached, the command, DISDE, is executed displaying the current debugging environment.

```
DB/setb l=6 c='disde'
```

The following command causes a break to occur prior to the execution of the SQR function of a C source code module SOURCE_C:

```
DB/setb m=source_c p=sqr
```

The following command causes a break to occur prior to the execution of the LCLBCK0200 block of a C source code module SOURCE_C:

```
DB/setb m=source_c p=lclbck0200
```

SET_STEP_MODE (SETSM)

Purpose: SET_STEP_MODE (SETSM)

Executes a specified subset of a task and receive control. You can select the unit of stepping as well as the span in number of units. SET_STEP_MODE is not available for C programs.

Format:

```
SET_STEP_MODE or
SETSM
    MODE = keyword                              (required)
    UNIT = keyword                              (optional)
    MODULE = list of name or keyword            (optional)
    PROCEDURE = list of name or keyword         (optional)
    SPAN = integer                              (optional)
    COMMAND = string                            (optional)
    STATUS = status variable                    (optional)
```

Parameters: MODE

Specifies whether to activate or deactivate step mode. Options are:

ON

Step mode is activated. When step mode is on, a RUN command causes one step to be executed. A step is defined by the UNIT and SPAN parameters.

If you specify MODE = ON and step mode is already on, all previous values will be replaced with the new values.

OFF

Step mode is deactivated. When step mode is off, any remaining parameters are ignored.

UNIT or U

Specifies the length of the step. Options are:

Omitted

Same as UNIT = LINE.

LINE or L

A step is reported before the code is executed for each line, except for the procedure lines.

PROCEDURE or P

A step is reported each time a new procedure begins and after any prolog code for the procedure has executed.

COBOL_SECTION or CS

A step is reported each time a section header in a COBOL program is reached.

COBOL_PARAGRAPH or CP

A step is reported each time a paragraph in a COBOL program is reached.

MODULE or M

Used with the UNIT parameter, specifies the modules reported on. Options are:

Omitted

A step is reported that is in the current default module.

$ALL

A step is reported that is in any module.

$CURRENT

A step is reported only if the step occurs in the module where the program is executing when step mode is activated.

list of names

A step is reported if the step occurs in any of the named modules.

Source	Parameter Dependency
BASIC	MODULE is the main program referred to as $MAIN. $MAIN can reference internal subroutines and internal functions which are procedures within the module $MAIN. MODULE can also name an external subroutine or external function. An external subroutine and an external function can reference internal subroutines and internal functions which are procedures within that module.
COBOL	MODULE names a program specified on the Program-id statement. Since MODULE and PROCEDURE parameters name the same program, both parameters should be assigned the same name or only one parameter should be specified.
CYBIL	MODULE names a module which may contain a program, procedure, or function.
FORTRAN	MODULE names a program, subroutine, function, or block data subprogram. Since MODULE and PROCEDURE parameters name the same program unit, both parameters should be assigned the same name or only one parameter should be specified.
Pascal	MODULE is the program name.

Invalid Parameter Combinations

You cannot specify both the MODULE and PROCEDURE parameters in the same SET_STEP_MODE command.

PROCEDURE or P

Used with the UNIT parameter; specifies the procedure reported. Options are:

Omitted

A step is reported that is in the current default procedure.

$ALL

A step is reported that is in any procedure.

$CURRENT

A step is reported only if the step occurs in the procedure where the program is executing when step mode is activated.

list of names

A step is reported if the step occurs in any of the named procedures.

Source	Parameter Dependency
BASIC	PROCEDURE names internal subroutines and internal functions within a module.
COBOL	PROCEDURE names a program specified on the Program-id statement. Since MODULE and PROCEDURE parameters name the same program, both parameters should be assigned the same name or only one parameter should be specified.
CYBIL	PROCEDURE names a program, procedure, or function in the module specified by the MODULE parameter.
FORTRAN	PROCEDURE names a program, subroutine, function, or block data subprogram. Since MODULE and PROCEDURE parameters name the same program unit, both parameters should be assigned the same name or only one parameter should be specified.
Pascal	PROCEDURE names a program, procedure or function name.

Invalid Parameter Combinations

You cannot specify both the MODULE and PROCEDURE parameters in the same SET_STEP_MODE command.

SPAN or S

Specifies how many steps must occur before execution stops and the step is reported.

Omitted

Debug will report every step that occurs.

integer

Specifies the number of steps that occurs before execution stops.

COMMAND or COMMANDS or C

Specifies the string of commands to be executed by Debug, SCL, or any other active command processor when a step occurs. Options are:

Omitted

Indicates that no commands are associated with the step.

string

Specifies a string of commands to be used.

If a command in the string includes a quoted string, that string must be enclosed in two single apostrophes. After the commands in the string have been executed, commands are read from the current Debug input file unless the string contains a RUN command.

STATUS

Specifies a variable to reveive the return status of the command. Options are:

Omitted

Debug processes the next command if an error does not occur. The return status is output to $RESPONSE (and to the Debug output file if $RESPONSE is connected to that file) if an error does occur. $RESPONSE is normally connected during interactive debugging.

status variable

The specified variable receives the return status.

SET_STEP_MODE (SETSM)

Remarks:

- SET_STEP_MODE is not available for programs written in C.
- If step mode is activated, a RUN command causes your program to execute for the specified unit. You are then prompted for further command input.
- A string of commands can be associated with the step and is processed each time the step completes.
- Stepping with a unit of line or procedure is only available if the source program was compiled with OPTIMIZATION_LEVEL = DEBUG.
- Activating step mode is an effective debugging aid, but uses a lot of execution time.
- If you specify MODE=ON and step mode is already on, all previous values are replaced with the new values.

Examples: The following commands activate step mode with a unit of line in the current module, execute the entire program and display each line that is executed, then deactivate step mode:

```
DB/setsm on c='disde ua; run'
DB/run
-- Execution is currently stopped at B 046 00000048
   which, in symbolic terms is M=TEST L=6
-- DEBUG:  divide_fault at M=TEST L=6 BO=12
DB/setsm off
DB/quit
```

When using RUN on the command stream, SETSM runs continuously until the program terminates, reaches another break, or reaches an execution error.

Debug Line Mode Functions

The Debug line mode functions are intended for use with NOS/VE System Command Language during a Debug session. These functions are only available while Debug has control. They are not known while the user program is executing or after the Debug session has been terminated.

The Debug functions follow the syntax and conventions for SCL functions, as described in the SCL Language Definition Usage manual.

The source language in which your program is written determines the use of some of the function parameters. Source language dependencies are identified in the applicable parameter description.

$CURRENT_LINE ($CL)

Purpose: Returns the value of the current line number from the program at the point where Debug has control.

Format: $CURRENT_LINE or $CL

Parameters: None.

Example: Using SCL control statements, the DISPLAY_CALLS command is executed only if the current value of the line number returned by the $CURRENT_LINE function is greater than 100:

```
DB/if $current_line > 100 then
if/display_calls
if/ifend
```

$CURRENT_MODULE ($CM)

Purpose: Returns the name of the module where execution is stopped. A null string is returned if no module name is found.

Format: $CURRENT_MODULE or $CM

Parameters: None.

Remarks:

- In a BASIC program, a module is the main program referred to as $MAIN. A module can also be an external subroutine or an external function.
- In a C program, a module is a C compilation unit (a C source file).
- In a COBOL program, a module is the name given as the Program-id.
- In a CYBIL program, a module is a program module.
- In a FORTRAN program, a module is a program, subroutine, function, or block data subprogram.
- In a Pascal program, a module is the program name.

Example: Using SCL control statements, break1 is set on line 234 if execution is stopped within the module 'MAIN':

```
DB/if $current_module = 'MAIN' then
if/setb break1 line=234
if/ifend
```

$CURRENT_PROCEDURE ($CP)

Purpose: Returns the name of the procedure where execution is stopped. A null string is returned if no procedure name is found.

Format: $CURRENT_PROCEDURE or $CP

Parameters: None.

Remarks:

- In a BASIC program, a procedure is an internal subroutine or an internal function within a module.
- In a C program, a procedure is a function or a block within a function.
- In a COBOL program, a procedure is a program specified on the Program-id statement.
- In a CYBIL program, a procedure is a program, a procedure, or a function.
- In a FORTRAN program, a procedure is a program, subroutine, function, or block data subprogram.
- In a Pascal program, a procedure is a program, an internal function, or procedure.

Example: Using the SET_STEP_MODE command, only the procedure named 'SUB12' is reported on. Execution occurs one line at a time.

```
DB/setsm on unit=procedure c='if ..
DB../$current_procedure=''SUB12'' then; setsm on ..
DB../unit=line; else; run; ifend'
```

$CURRENT_PVA ($CPVA)

Purpose: Returns an integer value for the process virtual address [PVA] where Debug execution has stopped.

Format: $CURRENT_PVA or $CPVA

Parameters: None.

Example: Using SCL control statements, the DISPLAY_CALLS command is executed only if the current program virtual address is greater than the hexadecimal value 0b03500000026:

```
DB/if $current_pva > 0b03500000026(16) then
if/display_calls
if/ifend
```

Because the IF statement is an SCL control statement, not a Debug statement, the hexadecimal constant must begin with a 0 and end with the radix specifier (16).

$MEMORY ($MEM)

Purpose: Returns the contents of memory which can be used as input to the DISPLAY_MEMORY or CHANGE_MEMORY commands. You can display memory which is the object of a pointer in memory if the pointer is contained in a register.

Format: $MEMORY or $MEM(pva, number, kind)

Parameters: pva

Specifies the process virtual address.

number

Specifies the number of bytes to return. If kind is an integer, the number of bytes returned must be in the range 1 through 8. If the value returned is a string, number must be in the range 1 through 256. If number is omitted, 6 bytes are returned.

kind

Specifies the type of value returned. If kind is specified as integer, the value is returned as a hexadecimal integer with radix specified. If kind is specified as string, the value is returned as a string. If kind is omitted, the value of integer is returned.

Example: If register A7 contains a pointer to memory, the following DISPLAY_MEMORY command displays the object of that pointer:

```
DB/display_memory address=$memory($register(a,7))
```

The $REGISTER function returns the pva in register A7, the $MEMORY function returns the pva to which A7 points, and the DISPLAY_MEMORY command displays the desired piece of memory.

$PROGRAM_VALUE ($PV)

Purpose: Returns the displayable value of a program variable, array element or substring. Input values for MODULE, PROCEDURE, RECURSION_LEVEL, and RECURSION_DIRECTION can be entered to provide a more complete identification of the named variable.

Format: $PROGRAM_VALUE or $PV(name,module,procedure, recursion_level,recursion_direction)

Parameters: name

This parameter is required; it specifies the name of the program variable, array element, string, substring, field reference, or pointer dereference in the source program whose value is to be displayed. The name can be any variable defined and used in your program. SCL string variables can be used to name long program names. To do this, assign the SCL variable to a string containing the identifier. Then use the SCL variable preceded by a question mark (?) as the value for the name.

Source	Parameter Dependency
FORTRAN	The name cannot be a namelist entry.

module

Specifies the name of the module that contains the name parameter variable. Omission causes the default module (the module executing when Debug gained control or the module specified by the CHANGE_DEFAULTS command) to be used.

Source	Parameter Dependency
BASIC	The module parameter names the main program referred to as $MAIN. $MAIN can reference internal subroutines and internal functions which are procedures within the module $MAIN. The module parameter can also name an external subroutine or external function. An external subroutine and an external function can reference internal subroutines and internal functions which are procedures within that module.

C	The module parameter names a C compilation unit (a C source file). The initial module is EM, a startup module. The EM module has no Debug tables so the default, $CURRENT, is not useful until program execution reaches a module with Debug tables. Global variables are in the module c_globals.
COBOL	The module parameter names a program specified on the Program-id statement. Since the module and procedure parameters name the same program, both parameters should be assigned the same name or only one parameter should be specified.
CYBIL	The module parameter names a module which may contain a program, procedure, or function.
FORTRAN	The module parameter names a program, subroutine, function, or block data subprogram. Since the module and procedure parameters name the same program unit, both parameters should be assigned the same name or only one parameter should be specified.
Pascal	The module parameter names the program name.

procedure

Specifies the name of the procedure that contains the name parameter variable. Omission causes the default procedure (the procedure executing when Debug gained control or the procedure specified by the CHANGE_DEFAULTS command) is used.

Source	Parameter Dependency
BASIC	The procedure parameter names internal subroutines and internal functions within a module.
C	The procedure parameter names a function or a block within a function.
COBOL	The procedure parameter names a program specified on the Program-id statement. Since the module and procedure parameters name the same program, both parameters should be assigned the same name or only one parameter should be specified.

CYBIL	The procedure parameter names a program, a procedure, or a function in the module specified by the module parameter.
FORTRAN	The procedure parameter names a program, subroutine, function, or block data subprogram. Since the module and procedure parameters name the same program unit, both parameters should be assigned the same name or only one parameter should be specified.
Pascal	The procedure parameter names a program, procedure, or function name.

recursion_level

Specifies the recursion level of the procedure specified by the procedure parameter. The value of the data item specified by the name parameter and known to this recursive level of the procedure is displayed. Recursion does not affect static or global variables. Omission specifies a recursion_level of 1.

Source	Parameter Dependency
COBOL FORTRAN	Recursion_level is not supported. Omit this parameter.

recursion_direction

Specifies whether the recursion_level is counted forward from the first call or backward from the most recent call. Recursion does not affect static or global variables. Omission specifies a recursion_level of backward.

Source	Parameter Dependency
COBOL FORTRAN	Recursion_direction is not supported. Omit this parameter.

Example: A break occurs at line 23 of the program currently executing. If the value of the variable INDEX at that point is less than 45, the program resumes execution:

```
DB/setb bl l=23 c='if $program_value(index) ..
DB../< 45 then; run; ifend'
```

$REGISTER ($REG)

Purpose: Returns the contents of a specified register in hexadecimal integer format, including radix. $REGISTER is useful when specified for the ADDRESS parameter value on the DISPLAY_MEMORY and CHANGE_MEMORY commands.

Format: $REGISTER or $REG(kind, number)

Parameters: kind

Specifies the type of register the value is returned from. P specifies a P register, A specifies an A register, and X specifies an X register.

number

Specifies the register number the value is returned from.

Invalid Parameter Dependencies

If the kind is specified as P, the number must not be specified because there is only one P register.

Example: For example, if register A4 contains a pointer to memory, the following DISPLAY_MEMORY command displays the object of that pointer and the CHANGE_MEMORY command changes the value of the object of that pointer to ´010105aaab´.

```
DB/display_memory address=$memory($register(a,4))
DB/change_memory address=$memory($register(a,5)) ..
DB../value=´010105aaab´
```

To verify that the value has changed, the following command is entered:

```
DB/display_memory address=$memory($register(a,4))
```

The $REGISTER function returns the PVA in register A4, the $MEMORY function returns the PVA to which A4 points, and the DISPLAY_MEMORY command displays the desired piece of memory.

Comprehensive Debugging 6

This chapter describes some additional features and concepts of Debug.

Addressing 6-1
- Reported Addresses 6-1
- Referenced Addresses 6-3
- Addressing Bound Modules 6-4
  - Module/Procedure Offset Addressing 6-5
  - Module/Section Offset Addressing 6-5
  - Module Block Referencing 6-5

Interrupt Processing While Debugging 6-6
- Pause Break Interrupt 6-6
- Terminate Break Interrupt 6-6
- Nearly Exhausted Resource 6-6

Debugging Optimized Code 6-7

Optimizing Debug Performance 6-7

Debugging a Terminated Program 6-8

Debugging a CYBIL Runtime Error 6-8

Debugging Condition Handlers 6-9

Debug Rings 6-10
- Deferred Breaks 6-10
- Multiple Breaks 6-11
- Multi-ring Environment 6-11

Multi-task Debugging 6-11

Comprehensive Debugging 6

The features and concepts described in this chapter will allow you to use the Debug utility to its fullest. The use of addressing, interrupt processing, optimizing Debug performance, and condition handlers are a few of the concepts described.

Addressing

Debug makes use of source program addresses in two ways:

1. Addresses are reported when Debug gains control and in Debug command output, such as when DISPLAY_CALL or DISPLAY_BREAK is executed.

2. Addresses are referenced in Debug commands, such as SET_BREAK and DISPLAY_MEMORY.

Reported Addresses

The level of reported address is determined by the information available to Debug via tables. Module address tables indicating where modules are located are available for all languages by default. For C, COBOL, CYBIL, FORTRAN, FORTRAN Version 2, and Pascal programs, the following additional information must be requested when the program is compiled (see chapter 2 for compilation requirements):

- Line address tables indicating where code for each source line is located.

- Symbol tables indicating where each program variable is located.

Addresses in break report messages (issued when Debug gains control) are formatted as follows, depending upon the level of information available.

- When line and module tables are available (symbolic addressing):

  - If the address corresponds to the beginning of a line, then

    ```
    M=module_name L=line_number
    ```

  - If the address corresponds to the beginning of the specified statement of the line, then

    ```
    M=module_name L=line_number S=statement_number
    ```

  - Otherwise, if the address is somewhere within the line, then

    ```
    M=module_name L=line_number BO=byte_offset_from_start_of_
    line
    ```

- When only the module table is available (module addressing):

  - If the module is not bound (refer to the discussion of bound modules later in this chapter), then

    M=module_name P=procedure_name BO=byte_offset_from_start_procedure

  - Otherwise, if the module is bound, then

    M=module_name BO=byte_offset_from_start_of_bound_module

- When the line table is not available and the address is not within any module covered by the module table (machine addressing):

  A=address

Within the address formats:

- Module_name and procedure_name correspond to the source program module and procedure names.

- Line_number corresponds to a line number on the source listing.

- Byte_offset is a decimal number corresponding to the number of bytes beyond the beginning of a line or a hexadecimal number corresponding to the number of bytes beyond the start of a procedure or bound module.

- Address is a set of three hexadecimal numbers representing the ring number, segment number, and segment offset of a machine address.

Addresses reported in command output also provide the highest address level possible, but they are not always formatted the same as in break report messages. Addresses shown in DISPLAY_BREAK output are very similar, but addresses shown in DISPLAY_CALL output contain both the procedure name and line number. Typical DISPLAY_CALL output might look like the following:

```
-- Traceback from procedure PROC2 module MOD2 at line 34
-- Called from procedure PROC1 module MOD2 at line 55 byte
   offset 4
-- Called from procedure BEGIN_PROCESS module MOD1 byte offset
   1A3(16)
```

Addresses shown in DISPLAY_REGISTER output for the P and A registers are formatted only as hexadecimal addresses in the form

```
r sss oooooooo
```

where r is the ring number, sss is the segment number, and oooooooo is the offset from the start of the segment. Pointer addresses displayed by DISPLAY_PROGRAM_VALUE are also formatted as hexadecimal machine addresses; dereferenced pointers to procedures are displayed as the procedure name if possible.

Referenced Addresses

Several Debug commands reference program code and data addresses. For example, SET_BREAK designates an address or address range for break events, DISPLAY_MEMORY specifies the address of memory to be displayed, and DISPLAY_PROGRAM_VALUE names a program variable whose value is to be displayed.

Just as for reporting addresses, the capabilities available when referencing program addresses depend on the information available:

- Symbolic addressing (source level addressing) is available if line and symbol tables exist (they exist when line number and symbol tables are generated at compile time (see chapter 2 for compilation requirements)).

- Module/procedure offset addressing is available if module tables exist (they always do for user programs).

- Machine-level addressing is always available.

Addresses can be referenced in many more forms than the form in which they are reported. For example, entry point names, section names, common block names, statement labels, and program variables can be referenced, but addresses are never reported in these terms. Machine addresses can be referenced only as a single integer (a 12-digit hexadecimal value); they are reported, however, either as a 12-digit hexadecimal integer or as three separate integers corresponding to ring number, segment number, and byte offset.

Not all address forms, however, are used by all commands. For example, the DISPLAY_PROGRAM_VALUE command allows a program variable to be referenced by name, including all of the subscripting and qualification syntax. But, the DISPLAY_PROGRAM_VALUE command does not allow machine-level addressing. However, the DISPLAY_MEMORY command allows machine and module addressing and limited symbolic-level addressing. The SET_BREAK command allows all forms.

The different forms of addresses are specified by different parameters or parameter combinations. LINE, MODULE, PROCEDURE, NAME, ENTRY_POINT, SECTION, STATEMENT, STATEMENT_LABEL, and ADDRESS are typical address parameter names. Many of these address parameters can be used in combination to specify an address. For example, LINE and MODULE together specify a particular line of a particular module. NAME, MODULE, and PROCEDURE together specify a particular name of a particular procedure in a particular module. Similarly, SECTION, LINE, STATEMENT, and STATEMENT_LABEL can be used in conjunction with MODULE. ENTRY_POINT and ADDRESS, however, cannot be used in conjunction with MODULE or with each other because each one specifies an address independent of any module. An error message is displayed if an invalid combination of address parameters is used.

The BYTE_OFFSET parameter can be used to modify the address parameters. For example, the MODULE parameter without he BYTE_OFFSET parameter specifies the first byte of the module; the MODULE parameter modified with BYTE_OFFSET=4, however, specifies the fifth byte of the module.

Another parameter, BYTE_COUNT, can be used to establish or modify the block size (address range) associated with a referenced address. The BYTE_COUNT parameter indicates how many memory bytes are to be included in the block. For example,

```
section=trap, byte_count=3
```

identifies a three-byte block that begins at section TRAP. BYTE_COUNT and BYTE_OFFSET can be used to modify any referenced address except a program variable (NAME parameter).

Addressing Bound Modules

Using the SCL utility CREATE_OBJECT_LIBRARY, you can combine individual object modules into a single bound load module that loads and executes faster than the original separate modules. (For more information, see the CREATE_OBJECT_LIBRARY command in the SCL Object Code Management manual.) Binding modules together has no effect on address reporting or address referencing at the symbolic level; you can symbolically debug bound modules in the same way as its component object modules. If the component modules have Debug tables, the bound module keeps those Debug tables so program locations and variables can be referenced symbolically. However, binding removes entry points from the entry point table; Debug will not be able to locate a removed entry point by use of the ENTRY_POINT parameter specified on the SET_BREAK command after modules have been bound. If two or more modules with the same name are ever loaded together or bound together, only the first one loaded can be referenced in Debug commands.

Module/Procedure Offset Addressing

Binding modules also has an effect on module/procedure offset addressing. After binding, the original module and procedure names are not available if the tables that support symbolic addressing are not available; addresses are reported and must be referenced in terms of the new bound module name and byte offsets from the beginning of the module. Code from all original component modules is combined into one code section, static data from all original modules is combined into one static data memory section, and so forth, such that the original component portions of each section cannot be distinguished by Debug. You can deduce where each component portion is by inspecting the bind map produced by the CREATE_OBJECT_LIBRARY (described in the Object Code Management manual.)

Module/Section Offset Addressing

Module/Section offset addressing is not available for the original component modules of a bound module since binding merges all like sections together and Debug cannot determine where any particular component's section is located. Common blocks are an exception. Debug can determine where each module's common blocks are located after binding since a common block starts at the same location for each module that uses it.

Module Block Referencing

Binding also has an effect on the ability to reference a module as a block. After binding, the new bound module can be referenced as a whole block but the original component modules can be referenced, even if supporting debug tables are available.

Interrupt Processing While Debugging

Three external events can interrupt an executing user program or the Debug utility. These events are pause break, terminate break, and nearly exhausted resource. The effects of these interrupts are described below.

Pause Break Interrupt

When your source program is executing:

Default system action occurs. You are allowed to enter other commands.

When Debug is executing:

Default system action occurs. If you have established a handler for this condition, that handler gains control. Debug does not gain control unless the handler returns with normal status.

Terminate Break Interrupt

When your source program is executing:

Default system action occurs. You are allowed to enter other commands.

When Debug is executing:

If a Debug command is where the program was executing, that command is terminated and you are prompted for a new command. If Debug is already waiting for a command, the terminate break is ignored.

Nearly Exhausted Resource

When your source program is executing:

Debug does not get control. If you have established a handler for this condition, that handler gains control; otherwise, the default system action occurs.

When Debug is executing:

Debug does not get control. If you have established a handler for this condition, that handler gains control; otherwise, the default system action occurs. Debug does not gain control unless the handler returns with normal status.

Debugging Optimized Code

Most compilers can generate more than one level of object code. When specifying a special level of optimization for Debug on the compile command (see chapter 2 for compilation requirements), the most debuggable object code possible is generated. This level of object code contains a separate packet of machine instructions for each executable source statement and carries no altered variable values across statement boundaries in registers without also updating their values in memory. It also recognizes the start of execution of each new line that starts a statement or procedure and ensures that Debug can always find actual parameter lists.

If some higher level of optimization is selected, you can still use Debug, but with restricted capabilities. For example, you cannot display program values that are permanently allocated to machine registers. When values are temporarily carried in registers between statements, or when code for several source statements is mixed together, displayed values may not be the most recent values. Because some lines may be removed during optimization at higher levels, break report locations may not be as precise and variables may not be available. In addition, the SET_STEP_MODE command cannot step lines or procedures.

Optimizing Debug Performance

Debug will necessarily have some performance impact on the program being debugged. The benefits gained in terms of programmer time saved during debugging, however, far outweigh any additional execution costs. The execution time degradation is only suffered while the program is being debugged and is normally not perceptible.

The degree of degradation depends mainly on both the number and types of breaks set. Special debug hardware is used to detect certain break events such as EXECUTION at a specific address, READ from a specific address range, and WRITE into a specific address range. Performance is impacted during debugging because the hardware must check to see if each instruction it executes requires a Debug hardware break to be set internally. You are encouraged to delete breaks after they are no longer needed to improve debugging performance. The SET_STEP_MODE command is a most effective debugging aid, however, because of the use of internal breaks while debugging, the degradation of execution time can be very costly.

There is no significant additional load or execution cost associated with the line address and symbol tables which are a part of binary object modules. The loader does not process or manipulate the tables in any way and they are never paged into memory unless they are used within Debug.

Debugging a Terminated Program

In a Debug session, very little debugging can be done once your program has terminated. Currently, no facility is provided to reexecute all or part of your terminated program. You must quit the Debug session and reexecute your program with DEBUG_MODE=ON if you want to continue debugging. The only debugging facilities available to you after your program terminates are the Debug DISPLAY commands.

If your program terminated by returning from its starting procedure (the Debug report message will say so), only static variables can be displayed. No active procedure or automatic variables exist, so DISPLAY_CALLS provides no information. If your program terminates by a call to PMP$EXIT or PMP$ABORT, at least one active procedure exists (the one that called PMP$EXIT or PMP$ABORT). In this case, DISPLAY_CALLS can provide some useful information and automatic variables of the active procedures can be displayed.

Debugging a CYBIL Runtime Error

To display or change program variables after receiving a CYBIL runtime error, you must specify the MODULE and PROCEDURE parameters on the Debug command. This is required because the runtime error has caused the program to branch to the runtime error processing procedure (the pocket code).

The pocket code performs runtime error processing; it issues the runtime error message giving the line number at which the error was detected. However, when a program branches to its pocket code, it branches outside the user's procedure and module so the default module and procedure ($CURRENT) are no longer the user's procedure and module. Therefore, to display or change program variables, you must specify the module and procedure containing the variables.

To illustrate this, the following program is compiled with the parameter RUNTIME_CHECKS=ALL:

```
module mode ;
    program p ;
        pr ;
    procend ;
    procedure pr ;
        var v3 : 1..10 ;
        v3 := 6 ;
        v3 := v3 * 2 :
    procend ;
modend ;
```

When the program is executed, the following runtime error occurs:

```
--ERROR-- CYBIL run time error, range error, detected at line 8
          of MOD.
```

When this error occurs while the program is executing in Debug mode, the program variables can be displayed only if the module and procedure are specified, as follows:

```
DB/display_program_value, $all, m=mode, p=pr

-- DISPLAY OF ALL VARIABLES IN PR

V3 = 6
```

Debugging Condition Handlers

Condition handlers are special procedures whose purpose is to process exception conditions when they arise. They are automatically activated by NOS/VE when the conditions for which they have been established occur. Condition handlers can be established at the program level (PMP$ESTABLISH_CONDITION_HANDLER) and at the SCL level (WHEN/WHENEND block). Condition handlers can be established for one or more of the following classes of conditions:

- System conditions (exponent overflow, divide fault, etc.).
- Block exit condition.
- Interactive conditions (pause break, terminate break).
- Job resource conditions (time nearly expired).
- Segment access conditions.
- Process interval timer condition.
- User-defined conditions.

When executing with DEBUG_MODE=ON, Debug first gains control when any condition occurs, except for some conditions such as, job resource conditions and detected_uncorrected_error (one of the system conditions) occurs. The condition handler of the program, if one exists, is not executed until a Debug RUN command is executed.

The condition handler of the program can be debugged using Debug, but the program does not execute until you have a chance to respond to the condition. For conditions for which breaks can be set, a RUN commnd can be associated with the break so that the command is automatically executed when the break occurs. (Refer to the COMMAND parameter of the SET_BREAK command in chapter 5.) This mechanism makes it possible to effectively circumvent the preemptive control of Debug. It appears as though Debug did not get control since the RUN command automatically executes the instant the condition arises.

Debug Rings

Debug normally runs in the same ring as the program being debugged. A ring is the level of hardware protection given to a program. A program is protected from unauthorized access by tasks executing in higher rings. (See the SCL Object Code Management manual for more information about rings.)

You can, however, control the ring in which Debug executes by using the SET_DEBUG_RING command. The SET_DEBUG_RING command specifies the ring in which Debug executes. The Debug ring cannot be set to a ring more privileged than the lowest ring for which you are validated.

You are responsible for ensuring that the program being executed also runs at the same ring as set by the SET_DEBUG_RING command. (The ring attributes of the program to be executed can be changed using the SCL CHANGE_FILE_ATTRIBUTES command.)

If your program runs entirely in one ring, you need not be concerned with the Debug ring except to understand deferred breaks and multiple breaks.

Deferred Breaks

Breaks which are set below the Debug ring, that is, the break occurs in a lower numbered ring than the Debug ring, are deferred, or delayed, until execution again reaches the Debug ring. The break is deferred so that you do not get control in a ring more privileged than your own. If you had control at a lower ring, you could view or change data that you normally do not have access to, thereby compromising system security.

Deferred breaks can occur even when your program runs in a single ring. Many of the operating system services used by the program execute in more privileged rings. For example, if you set a READ or WRITE break on a status variable used in some NOS/VE request and that variable is accessed in a lower ring, the break is delayed until NOS/VE returns control to your program.

When a break is deferred, a special break report message is displayed. The break is reported as having happened at the line that made the call, and a second line indicating the actual address of the event is displayed. The second line is formatted as follows:

```
Trap deferred from  address
```

where address is where the event actually occurred.

Multiple Breaks

Because breaks below the Debug ring are deferred until control returns to the Debug ring, several breaks can be stacked up before Debug gains control. When this happens, multiple breaks must be processed.

If there are several unprocessed breaks outstanding when Debug gains control, each one is reported in the usual way but only the first one is processed. No commands are processed for the most recent breaks, not even commands associated with the break definition, since execution of the commands could destroy the environment that existed when the first break occurred.

Multiple breaks can also occur when execution is not below the Debug ring. For example, two terminal breaks or an execution break and a terminal break could occur before Debug gets control. If this happens, only the first break is processed.

Multi-ring Environment

The ability of Debug to function in a multi-ring environment is limited. If a break event occurs in a lower ring than the Debug ring, Debug gains control, but your options are limited. You can only resume execution of the interrupted procedure or terminate the Debug session. Any program condition handlers established for that event are not processed.

Multi-task Debugging

The use of Debug in a multi-task environment is very restricted. If an initial task executes with DEBUG_MODE=ON and then spins off a second task, the second task executes with DEBUG_MODE=ON (if its program description says to). This causes two separate instances of Debug to be active. You may have difficulty distinguishing between them, as well as determining to which task a terminal is connected. One way to determine which instance of Debug has control is to inspect the output from the DISPLAY_CALL calling chain or from the user address displayed by DISPLAY_DEBUGGING_ENVIRONMENT.

Source Language Debug Examples 7

This chapter demonstrates examples of Debug sessions in screen mode.

Debugging a BASIC Program 7-1

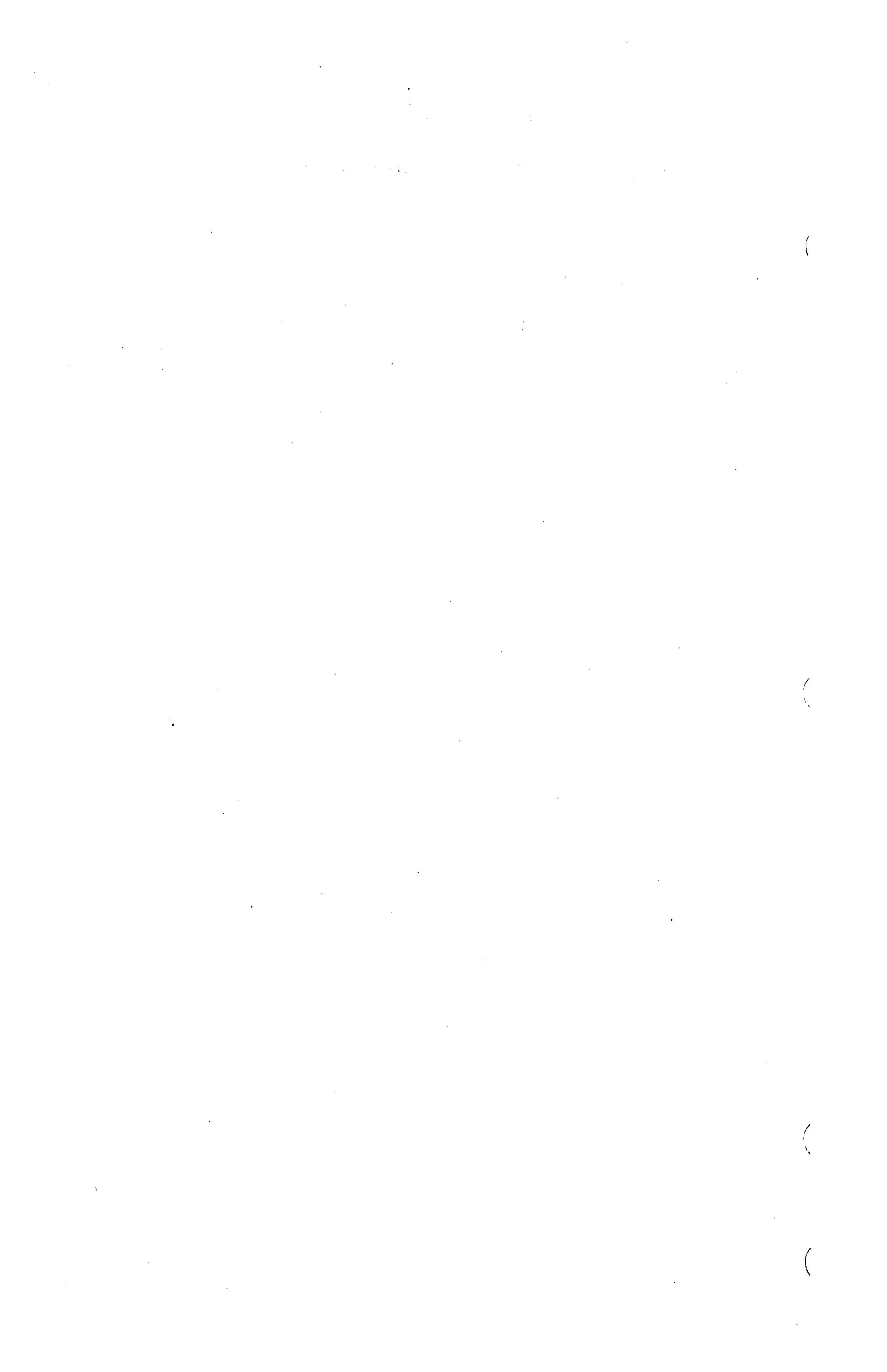
Debugging a C Program 7-14

Debugging a COBOL Program 7-26

Debugging a CYBIL Program 7-40

Debugging a FORTRAN Version 1 or
FORTRAN Version 2 Program 7-55

Debugging a Pascal Program 7-68



Source Language Debug Examples 7

As previously mentioned, Debug can be used in line mode or screen mode. You can also use Debug to perform machine-level debugging as well as symbolic debugging. This chapter demonstrates examples using screen mode Debug for symbolic debugging.

An example is demonstrated in screen mode Debug for each of the following source languages:

BASIC
C
COBOL
CYBIL
FORTRAN Versions 1 and 2
Pascal

Debugging a BASIC Program

This example is presented as a sequence of steps. To get the most benefit, you should create the sample program illustrated in figure 7-1, then perform each step as you read it. The sample program, EXAMPLE_BAS, provides three test cases to debug. Each test case in EXAMPLE_BAS is used to demonstrate the application of some Debug function keys. After you work this example, you will be able to debug your BASIC programs using Screen Mode Debug.

EXAMPLE_BAS is divided into the following test cases:

TEST1

A loop that increments a counter and then calls a subprogram to square and display the count. TEST1 demonstrates the use of the CHAVAL, GOTO, HSPEED, SEEVAL, STEP1 and STEPN functions.

TEST2

A loop that builds a 6-row table of 3-character strings. Input to the table is an 18-character list for the months JAN through JUN. TEST2 moves three characters at a time from the character list to the table and displays each entry. TEST2 shows how to step through loops, use line mode Debug commands in screen mode Debug, and how to scroll through Debug and program output data.

TEST3

A division test that results in a divide fault. TEST3 demonstrates how Debug handles execution errors.

```
DIM MONTHTABLE$(16)
DEFINT C,M

LET DIVIDEND = -100
LET DIVISOR = 0

LET MONTHCOLUMN = 1
LET MONTHLIST$ = "JANFEBMARAPRMAYJUN"

LET COUNTER = 0

REM TEST1:  Add to counter and call subroutine to square and
REM         display count.

LET COUNTER = 1
FOR COUNTER = 1 TO 10
    CALL SQUAREPROCEDURE (COUNTER)
NEXT COUNTER

REM TEST2:  Create single column table for each month.

FOR MONTHROW = 0 TO 5
    MONTHTABLE$(MONTHROW)=MONTHLIST$(MONTHCOLUMN:MONTHCOLUMN+2)
    PRINT "THE MONTH IS:  " MONTHTABLE$(MONTHROW)
    LET MONTHCOLUMN = MONTHCOLUMN + 3
NEXT MONTHROW

REM TEST3:  Create divide fault.

LET QUOTIENT = DIVIDEND / DIVISOR
PRINT "ANSWER IS:  " ANSWER

END

REM Subroutine SQUAREPROCEDURE

SUB SQUAREPROCEDURE (COUNTER)
    LET RESULT = 0
    LET RESULT = COUNTER * COUNTER
    PRINT COUNTER " TIMES" COUNTER " =" RESULT
END SUB
```

Figure 7-1 Debug Example: Source file EXAMPLE_BAS

Preparing to Debug

After you create EXAMPLE_BAS, you must compile your program and prepare your Debug session for the screen mode environment. You can then execute EXAMPLE_BAS under Debug control. Do this as follows:

1. Assuming EXAMPLE_BAS is contained in permanent file $USER.EXAMPLE_BAS, prepare the screen mode environment and compile EXAMPLE_BAS by entering the following commands:

   ```
   /change_interaction_style style=screen
   /basic input=$user.example_bas binary=lgo
   ```

2. Execute EXAMPLE_BAS under control of Debug by entering the following command:

   ```
   /execute_task file=lgo debug_mode=on
   ```

   The source module of EXAMPLE_BAS is displayed in the Source window. The Debug functions are displayed at the bottom of the screen.

Display Screen Mode Commands

The functions below are used to display helpful information about the Debugging enviornment:

HELP

Displays the Help window. Press a function key and a short explanation of the function's use appears in the Help window.

ZMIN

Used to display the source listing in the Source window.

Now perform the following steps to become familiar with the Debug functions:

1. Press the HELP key. The Help window is displayed.

2. Press each function key corresponding to a function displayed at the bottom of the screen. As you press each function key, a short explanation of the purpose of each function is displayed in the Help window.

3. Press RETURN. Exit HELP.

4. Press the ZMIN function key. The following message is displayed in the upper right hand corner of the screen:

   ```
   Enter compiler input file for $MAIN
   ```

5. Enter the source file name:

   ```
   example_bas
   ```

   The source listing of EXAMPLE_BAS is displayed in the Source window. Also, some new functions are displayed at the bottom of the screen.

6. Press the HELP key. The Help window is displayed again.

7. Press each function key corresponding to the new function displayed at the bottom of the screen. As you press each function key, a short explanation of the purpose of each new function is displayed in the Help window.

8. Press RETURN. Exit HELP.

Setting Breaks

It is often helpful to suspend program execution when debugging a program. The device for suspending execution of a program is called a break. In this sample session, the following functions are used to illustrate setting breaks:

FWD

Scrolls forward to the next screen of text.

FIRST

Displays the first screen of the source listing. Because FIRST is a lower priority function, it may not be assigned to a function key on terminals with only 16 function keys. Instead, FIRST is entered on the home line.

LOCATE

Prompts you to type in text, then searches the source listing for matching text. If a match is found, the cursor is moved to the line containing the matching text.

SETBRK

Sets an execution break on the line containing the cursor. The line is highlighted to show that it contains a break. Execution is suspended before the line containing the break is executed. Execution resumes with the statement on the line containing the break.

This section also uses the following item:

HOME

Moves the cursor to the home line. Line mode Debug commands can be entered on the home line for execution in screen mode Debug.

Perform the following steps to place three execution breaks in EXAMPLE_BAS:

1. Press the LOCATE function key. At the top right hand corner of the screen, you are prompted for the text to be located.

2. Enter the following text exactly as it appears in EXAMPLE_BAS:

   FOR MONTHROW

   The cursor is moved to the line:

   FOR MONTHROW = 0 TO 5

3. Press the SETBRK function key. A break is set and the line containing the cursor is highlighted to show that it contains an execution break.

4. Use the down-arrow key to move the cursor to the line containing:

   ```
   LET MONTHCOLUMN = MONTHCOLUMN + 3
   ```

   If you do not see this line on your screen, press the FWD key. The next screen of the EXAMPLE_BAS source listing is displayed. Use the down-arrow key to position the cursor on the correct line.

5. Press the SETBRK function key. The line is highlighted to show that it contains an execution break.

6. Use the down-arrow key to move the cursor to the line:

   ```
   LET QUOTIENT = DIVIDEND / DIVISOR
   ```

   If you do not see this line on your screen, press the FWD key. The next screen of the EXAMPLE_BAS source listing is displayed. Use the down-arrow key to position the cursor on the correct line.

7. Press the SETBRK function key. The line is highlighted to show that it contains an execution break.

8. Press the FIRST function key. The first screen of the EXAMPLE_BAS source listing is displayed in the Source window.

   If FIRST is not assigned to a function key, FIRST must be entered on the home line. To do this, press the HOME key. The cursor moves to the home line. Enter the following on the home line:

   ```
   first
   ```

   The first screen of the EXAMPLE_BAS source listing is displayed in the Source window.

Debugging TEST1

Using Debug, you can execute a program one line or several lines at a time. Also, you can examine a variable's contents, change its contents, and execute code containing the variable several times. These capabilities are demonstrated in this sample session using the following functions:

CHAVAL

Prompts you to enter a variable name and the value you want it to contain, then changes the variable's contents to the new value.

GOTO

Moves the execution pointer to the line that contains the cursor. Execution resumes with the statement on this line.

HSPEED

Executes a program until a break is encountered or the program ends.

SEEVAL

Prompts you to enter a variable name, then displays the value of the variable in the Output window.

STEP1

Executes a program one line at a time.

STEPN

Executes N lines of a program, where N is an integer.

Perform the following steps to demonstrate the use of the CHAVAL, GOTO, HSPEED, SEEVAL, STEP1, STEPN:

1. Press the STEPN function key. In the upper right corner of the screen you are prompted for the number of lines to execute; enter:

   17

   STEPN executes 17 lines of EXAMPLE_BAS, moving the execution arrow to the statement:

   FOR COUNTER = 1 TO 10

2. Press the STEP1 function key. The FOR statement is executed; the execution arrow points to the statement:

   CALL SQUAREPROCEDURE (COUNTER)

3. Press the STEP1 function key seven times. An iteration of TEST1 is executed one line at a time. The output generated by the iteration is displayed in the Output window.

4. Press the SEEVAL function key. A prompt to enter a variable name is printed in the upper right hand corner of the screen. Enter the name:

```
counter
```

   The value of COUNTER is displayed in the Output window:

```
counter = 2
```

   Thus, you can use SEEVAL to observe the contents of a variable.

5. Press the CHAVAL function key. A prompt for a variable name and its new value is displayed in the upper right hand corner of the screen; enter:

```
counter=8
```

   The value of COUNTER is changed to 8.

6. Press the SEEVAL function key. When you are prompted for a variable name, enter:

```
counter
```

   The following message is displayed in the Output window:

```
counter = 8
```

   Thus, the change of COUNTER's value is verified.

7. Press the STEPN function key. When you are prompted for the number of lines to execute; enter:

   7

   STEPN executes 7 lines of TEST1. The output generated by this loop iteration is displayed in the Output window.

8. Press the SEEVAL function key. When you are prompted for a variable name, enter:

   counter

   The value of COUNTER is displayed in the Output window:

   counter = 9

   Therefore, the value given to COUNTER in step 5 is used by the FOR statement.

9. Use the up-arrow key to move the cursor to the line:

   FOR COUNTER = 1 TO 10

10. Press the GOTO function key. The execution arrow moves to the line containing the cursor; execution resumes with this statement.

11. Press the HSPEED function key. Execution resumes from the FOR statement; COUNTER is initialized to 1. Execution of EXAMPLE_BAS continues until an execution break is encountered.

Debugging TEST2

After program execution is resumed in step 11 of TEST1, it stops at the break set on the PERFORM statement in TEST2. The following functions are used in TEST2 to illustrate more Debug capabilities:

BKW

Scrolls backward to the previous screen of text.

DELBRK

Deletes execution breaks.

HSPEED

Executes a program until a break is encountered or the program ends.

This section also uses the following items:

HOME

Press the HOME key to move the cursor to the home line. line mode Debug commands can be entered on the home line for execution in screen mode Debug.

DISPLAY_PROGRAM_VALUE

A line mode Debug command that displays the values of program variables.

Perform the following steps to learn how to execute loops one iteration at a time, execute line mode Debug commands, and scroll output data when using Debug:

1. Press the HSPEED function key. Execution stops at the break set on the last line of the FOR loop in TEST2; output generated by the loop is displayed in the Output window.

2. Press the HSPEED function key again. One iteration of the FOR loop is executed; execution stops at the break set at the statement, LET MONTHCOLUMN = MONTHCOLUMN + 3. Each time HSPEED is used, an iteration of the loop is performed. By using strategically placed execution breaks, as in this example, a loop can be executed one iteration at a time.

3. Press the HSPEED function key. One more loop iteration is performed.

4. Press the HOME key. The cursor moves to the home line.

5. Enter the line mode Debug command:

```
display_program_value name=$all
```

   The values of all variables in EXAMPLE_BAS are displayed in the Output window. Thus, line mode Debug commands can be used in screen mode Debug by entering them on the home line. For more information about using line mode Debug commands see the Debug Usage Manual.

6. Press the DELBRK key. The execution break is deleted. The highlight is removed from the line when the break is removed.

7. Press the down-arrow key until the cursor is inside of the Output window.

8. Press the BKW key. The data in the Output window scrolls backward. When the cursor is contained within the Output window, you can use the BKW and FWD keys to scroll backward and forward through the data in the window .

9. Press the HSPEED function key. The execution of EXAMPLE_BAS resumes, stopping when the line containing the third break is reached. The execution arrow points to the beginning of TEST3.

Debugging TEST3

After resuming execution of EXAMPLE_BAS in step 9 of section TEST2, execution stops at the begining of TEST3. In TEST3, Debug is presented with an execution error. The following functions are used in this sample session to demonstrate how Debug can be used when an exectuion error is encountered:

CHAVAL

Prompts you to enter a variable name and the value you want it to contain, then changes the variable's contents to the new value.

GOTO

Moves the execution pointer to the line that contains the cursor. Execution resumes with the statement on this line.

SEEVAL

Prompts you to enter a variable name, then displays the value of the variable in the Output window.

STEP1

Executes a program one line at a time.

QUIT

Used to exit Debug.

Perform the following steps to finish the example:

1. Press the STEP1 function key. The DIVISION statement is executed, execution of EXAMPLE_BAS halts, and the following message flashes in the upper right hand corner of the screen:

   ```
   divide_fault
   ```

2. Press the SEEVAL function key. When you are prompted for a variable name, enter:

   ```
   divisor
   ```

   The following message is displayed in the Output window:

   ```
   divisor = 0.
   ```

   A division by zero caused the execution error.

3. Press the CHAVAL function key. When you are prompted, enter:

   ```
   divisor=1.0
   ```

   The value of DIVISOR is changed to 1.

4. Press the SEEVAL function key. When you are prompted, enter:

   ```
   divisor
   ```

   The following text is displayed in the Output window:

   ```
   divisor = 1.0000E+0000
   ```

   The change to DIVISOR is verified.

5. Press the GOTO function key. The execution arrow points at the DIVISION statement and program execution resumes with this statement.

6. Press the STEP1 function key. The DIVISION statement is executed. Therefore, the GOTO and CHAVAL functions can be used in concert to recover from execution errors. However, to correct execution errors permanently, you must exit Debug, edit the program, and recompile it.

7. Press the STEP1 function key again. The result of the DIVISION statement is displayed in the Output window.

8. Press the STEP1 function key two times. EXAMPLE_BAS ends and the following message is displayed in the Output window:

   ```
   -- DEBUG: The status at termination was: NORMAL.
   ```

9. Press the QUIT function key. Exit Debug.

Now that you have concluded this example, you should be able to begin using screen mode Debug to debug your BASIC programs.

Debugging a C Program

This example is presented as a sequence of steps. To get the most benefit, you should create the sample program illustrated in figure 7-2, then perform each step as you read it. The sample program, EXAMPLE_C, provides three test cases to debug. Each test case in EXAMPLE_C is used to demonstrate the application of some Debug function keys. After you work this example, you will be able to debug your C programs using Screen Mode Debug.

EXAMPLE_C is divided into the following test cases:

TEST1

A loop that increments a counter and then calls a function to square and display the count. TEST1 demonstrates the use of the HSPEED, SEEVAL, CHAVAL, GOTO, and DELBRK functions.

TEST2

A loop that builds a 6-row table of 3-character strings. Input to the table is an 18-character list for the months JAN through JUN. TEST2 moves three characters at a time from the character list to the table, appends an end-of-line character, and displays each entry. TEST2 shows how to step through loops, use line mode Debug commands in screen mode Debug, and how to scroll through Debug and program output data.

TEST3

A division test that results in a divide fault. TEST3 demonstrates how Debug handles execution errors.

```
#include <stdio_h>

/*  Divide fault data  */

    int dividend = 100;
    int divisor  = 0;
    int quotient = 0;

/*  Month data */

    char *month_list_1 = "JANFEBMARAPRMAYJUN";
    char month[6][4];

main ()
{
    int month_column;
    int month_row;
    int counter;

 /* TEST1: Add to counter and call procedure to square       */
 /* and display. */

    for (counter=1; counter<=10; counter++)
        squaring_procedure (counter);

/* TEST2: Create a single column table for each month.       */

    month_column = 0;
    for (month_row=0; month_row<6; month_row++)
    {
        month[month_row][0] = month_list_1[month_column];
        month[month_row][1] = month_list_1[month_column+1];
        month[month_row][2] = month_list_1[month_column+2];
        month[month_row][3] = '\0';
        printf ("  The month is %s\n", month[month_row]);
        month_column = month_column + 3;
    }

/* TEST3:  Create a divide fault.                            */

    quotient = dividend/divisor;
    printf (" The answer is %d\n", quotient);

    return (0);
}

squaring_procedure (counter)
    int counter;
{
    int result;
    result = counter * counter;
    printf (" %d times %d = %d\n", counter, counter, result);
    return (0);
}
```

Figure 7-2 Debug Example: Source File EXAMPLE_C

Preparing to Debug

After you create EXAMPLE_C, you must prepare it for use with screen mode Debug. This requires preparing the screen mode environment and compiling EXAMPLE_C for use with Debug. You can then execute it under Debug control. Do this as follows:

1. Prepare and compile EXAMPLE_C using the cc command with the -g and -R compiler options by entering the following commands:

```
/set_working_catalog $user
/$system.cve.setup
/change_interaction_style style=screen
/cc -o lgo -g -R example_c
```

2. Execute EXAMPLE_C under control of Debug by entering the following command:

```
/execute_task file=lgo debug_mode=on
```

   The source listing of EXAMPLE_C is displayed in the Source window. Debug functions are displayed at the the bottom of the screen.

Displaying Screen Mode Commands

The functions below are used to display helpful information about the Debugging enviornment:

HELP

Displays the Help window. Press a function key and a short explanation of the function's use appears in the Help window.

ZMIN

Used to display the source listing in the Source.window.

ZMOUT

Used to display the C program modules and functions in the Source window.

Now perform the following steps to become familiar with the Debug functions:

1. Press the HELP key. The Help window is displayed.

2. Press each function key corresponding to the functions displayed at the bottom of the screen. As you press each function key, a short explanation of the purpose of each function is displayed in the Help window.

3. Press RETURN. Exit HELP.

4. Press the ZMOUT function key. The modules and functions of EXAMPLE_C are displayed.

5. Press the ZMIN function key. The source listing of EXAMPLE_C is displayed again. The ZMIN and ZMOUT function keys are used to display the source listing or the modules and functions of the program.

Setting Breaks

It is often helpful to suspend program execution when debugging a program. The device for suspending execution of a program is called a break. In this sample session, the following functions are used to illustrate setting breaks:

FIRST

Displays the first screen of the source listing. Because FIRST is a lower priority function, it may not be assigned to a function key on therminals with only 16 function keys. Instead, FIRST is entered on the home line.

FWD

Scrolls forward to the next screen of text.

LOCATE

Prompts you to type in text, then searches the source listing for matching text. If a match is found, the cursor is moved to the line containing the matching text.

SETBRK

Sets an execution break on the line containing the cursor. The line is highlighted to show that it contains a break. Execution is suspended before the line containing the break is executed. Execution resumes with the statement on the line containing the break.

This section also uses the following item:

HOME

Moves the cursor to the home line. Line mode Debug commands can be entered on the home line for execution in screen mode Debug.

Perform the following steps to place four execution breaks in EXAMPLE_C:

1. Press the LOCATE function key. At the top right hand corner of the screen, you are prompted for the text to be located.

2. Enter the following text exactly as it appears in EXAMPLE_C:

   ```
   squaring_procedure
   ```

   The cursor is moved to the line:

   ```
   squaring_procedure (counter);
   ```

3. Press the SETBRK function key. A break is set and the line containing the cursor is highlighted to show that it contains an execution break.

4. Use the down-arrow key to move the cursor to the statement:

   ```
   for (month_row=0; month_row<6; month_row++)
   ```

   If you do not see this line on your screen, press the FWD key. The next screen of the EXAMPLE_C source listing is displayed. Use the down-arrow key to position the cursor on the correct line.

5. Press the SETBRK function key. This line is highlighted to show that it contains an execution break.

6. Use the down-arrow key to move the cursor to the statement:

   ```
   month_column = month_column+3;
   ```

   If you do not see this line on your screen, press the FWD key. The next screen of the EXAMPLE_C source listing is displayed. Use the down-arrow key to position the cursor on the correct line.

7. Press the SETBRK function key. An execution break is placed on this line.

8. Use the down-arrow key again to move the cursor to the line:

   ```
   quotient = dividend/divisor;
   ```

   If you do not see this line on your screen, press the FWD key. The next screen of the EXAMPLE_C source listing is displayed. Use the down-arrow key to position the cursor on the correct line.

9. Press the SETBRK function key. The fourth execution break is set on this line.

10. Press the FIRST function key. The first screen of the EXAMPLE_C source listing is displayed in the Source window.

    If FIRST is not assigned to a function key, FIRST must be entered on the home line. To do this, press the HOME key. The cursor moves to the home line. Enter the following on the home line:

    ```
    first
    ```

    The first screen of the EXAMPLE_C source listing is displayed in the Source window.

Debugging TEST1

Using Debug, you can examine a variable's contents, change its contents, and execute code containing the variable several times. These capabilities are demonstrated in this sample session using the following functions:

CHAVAL

Prompts you to enter a variable name and the value you want it to contain, then changes the variable's contents to the new value.

DELBRK

Deletes execution breaks.

GOTO

Moves the execution pointer to the line that contains the cursor. Execution resumes with the first statement on this line.

HSPEED

Executes a program until a break is encountered or the program ends.

SEEVAL

Prompts you to enter a variable name, then displays the value of the variable in the Output window.

Perform the following steps to demonstrate the use of the HSPEED, SEEVAL, CHAVAL, GOTO, and DELBRK functions:

1. Press the HSPEED function key. Execution of EXAMPLE_C begins and the message HIGH SPEED is displayed in the upper right hand corner of the screen. Execution of EXAMPLE_C stops at the execution break that you set on the line:

   ```
   squaring_procedure (counter);
   ```

2. Press the HSPEED function key. Execution of EXAMPLE_C resumes from the SQUARING_PROCEDURE function call. Execution continues until the break on the function call is encountered again. The output generated by the SQUARING_PROCEDURE function call is displayed in Output window.

3. Press the SEEVAL function key. A prompt for the variable name is displayed at the top right hand corner of the screen. Enter the name:

   ```
   counter
   ```

The value of COUNTER is displayed in the Output window:

```
counter=2
```

4. Press the CHAVAL function key. A prompt for the variable name and its new value is displayed at the top of the screen; enter:

```
counter=8
```

The value of the FOR loop control variable, COUNTER, is changed to 8.

5. Press the SEEVAL function key. When you are prompted for a variable name, enter:

```
counter
```

The following message is displayed in the Output window:

```
counter=8
```

Thus, the change of COUNTER's value is verified.

6. Press the HSPEED function key. EXAMPLE_C executes until the break on the SQUARING_PROCEDURE function call is encountered. The output generated by this iteration of the FOR loop is displayed in the Output window.

7. Press the SEEVAL function key. When you are prompted for a variable name, enter:

```
counter
```

The value of COUNTER is displayed in the Output window:

```
counter=9
```

Thus, the FOR statement used the value given COUNTER in step 4.

8. Press the DELBRK function. This removes the break set on the SQUARING_PROCEDURE function call. The highlight is removed from the line when the break is removed.

9. Use the up-arrow key to move the cursor to the line:

```
for (counter=0; counter<=10; counter++)
```

10. Press the GOTO function key. The execution pointer moves to the line containing the cursor; execution resumes here when the HSPEED function key is pressed.

12. Press the HSPEED function key. Execution resumes at the FOR statement, and COUNTER is initialized to 0 within the statement. Therefore, the change made to COUNTER in step 4 is erased. Execution of EXAMPLE_C continues until the next break is encountered.

Debugging TEST2

After program execution is resumed in step 12 of TEST1, it stops at the break set on the FOR statement in TEST2. The following functions are used in TEST2 to illustrate more Debug capabilities:

BKW

Scrolls backward to the previous screen of text.

DELBRK

Deletes execution breaks.

FWD

Scrolls forward to the next screen of text.

HSPEED

Executes a program until a break is encountered or the program ends.

This section also uses the following items:

HOME

Moves the cursor to the home line. Line mode Debug commands can be entered on the home line for execution in screen mode Debug.

DISPLAY_PROGRAM_VALUE

A line mode Debug command that displays the values of program variables.

Perform the following steps to learn how to execute loops one iteration at a time, execute line mode Debug commands, and scroll output data when using Debug:

1. Press the HSPEED function key. Execution stops at the break set on the last statement of the FOR block; output generated by the block is displayed in the Output window.

2. Press the HSPEED function key again. One iteration of the FOR loop is executed; execution stops at the break set in the FOR block. Each time HSPEED is used, an iteration of the loop is performed. By using strategically placed execution breaks, as in this example, a loop can be executed one iteration at a time.

3. Press the HSPEED function key. One more loop iteration is performed.

4. Press the HOME key. The cursor moves to the home line.

5. Enter the line mode Debug command:

   display_program_value name=$all

   The values of all variables declared in the main function are displayed in the Output window. Thus, line mode Debug commands can be used in screen mode Debug by entering them on the home line. For more information about using line mode Debug commands see the Debug Usage Manual.

6. Press the DELBRK key. The execution break is deleted.

7. Press the down-arrow key until the cursor is inside of the Output window.

8. Press the BKW key. The data in the Output window scrolls backward. When the cursor is contained within the Output window, you can use the BKW and FWD keys to scroll backward and forward through the data in the window .

9. Press the HSPEED function key. Execution of EXAMPLE_C resumes.

Debugging TEST3

After resuming execution of EXAMPLE_C in step 9 of section TEST2, execution stops at the break set on the statement:

```
quotient = dividend/divisor;
```

Execution is suspended before this statement is performed.

In TEST3, Debug is presented with an execution error. The following functions are used in this sample session to demonstrate how Debug can be used when an exectuion error is encountered:

CHAVAL

Prompts you to enter a variable name and the value you want it to contain, then changes the variable's contents to the new value.

GOTO

Moves the execution pointer to the line that contains the cursor. Execution resumes with the first statement on this line.

HSPEED

Executes a program until a break is encountered or the program ends.

SEEVAL

Prompts you to enter a variable name, then displays the value of the variable in the Output window.

TRACE

Displays the chain of modules and functions that were called to reach the current line.

QUIT

Used to leave Debug.

Perform the following steps to finish this example:

1. Press the HSPEED function key. Execution resumes and then immediately halts. The following message flashes in the upper right hand corner of the screen:

   ```
   divide_fault
   ```

2. Press the SEEVAL function key. When you are prompted for a variable name, enter:

   divisor

   The following message is displayed in the Output window:

   divisor=0

   A division by zero caused the execution error.

3. Press the CHAVAL function key. When you are prompted, enter:

   divisor=1

   The value of DIVISOR is changed to 1.

4. Press the SEEVAL function key. When you are prompted, enter:

   divisor

   The following text is displayed in the Output window:

   divisor=1

   The change to DIVISOR is verified.

5. Press the GOTO function key. The execution arrow points to the DIVISION statement, so program execution resumes with this statement.

6. Press the HSPEED function key. The DIVISION statement is executed, the result of the DIVISION statement is displayed in the Output window, and EXAMPLE_C ends. The following message is displayed in the Output window:

   DEBUG: The status at termination was: NORMAL.

   Therefore, the GOTO and CHAVAL functions can be used in concert to recover from execution errors. However, to correct execution errors permanently, you must exit Debug, edit the program, and recompile it.

7. Press the TRACE function key. The modules and functions that were called upon program termination are displayed.

8. Press the QUIT function key. Exit Debug.

Now that you have concluded this example, you should be able to begin using screen mode Debug to debug your C programs.

Debugging a COBOL Program

This example is presented as a sequence of steps. To get the most benefit, you should create the sample program illustrated in figure 7-3, then perform each step as you read it. The sample program, EXAMPLE_COB, provides three test cases to debug. Each test case in EXAMPLE_COB is used to demonstrate the application of some Debug function keys. After you work this example, you will be able to debug your COBOL programs using Screen Mode Debug.

EXAMPLE_COB is divided into the following test cases:

TEST1

A loop that increments a counter and then calls a subprogram to square and display the count. TEST1 demonstrates the use of the CHAVAL, GOTO, HSPEED, SEEVAL, STEP1 and STEPN functions.

TEST2

A loop that builds a 6-row table of 3-character strings. Input to the table is an 18-character list for the months JAN through JUN. TEST2 moves three characters at a time from the character list to the table and displays each entry. TEST2 shows how to step through loops, use line mode Debug commands in screen mode Debug, and how to scroll through Debug and program output data.

TEST3

A division test that results in a divide fault. TEST3 demonstrates how Debug handles execution errors.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.    EXAMPLE-COB.

ENVIRONMENT DIVISION.

CONFIGURATION SECTION.

INPUT-OUTPUT SECTION.
FILE-CONTROL.
       SELECT IN-FILE ASSIGN I.
       SELECT OUT-FILE ASSIGN O.

DATA DIVISION.

FILE SECTION.
FD  IN-FILE LABEL RECORDS OMITTED.
01  IN-REC PIC X(256).
FD  OUT-FILE LABEL RECORDS OMITTED.
01  OUT-REC PIC X(80).

WORKING-STORAGE SECTION.
01  DIVIDE-FAULT-DATA.
    02  DIVIDEND         PIC S999     VALUE 100.
    02  DIVISOR          PIC 99       VALUE 0.
    02  QUOTIENT         PIC S9999    VALUE 0.

01  MONTH-DATA.
    02  MONTH-LIST-1     PIC X(18)    VALUE "JANFEBMARAPRMAYJUN".
    02  MONTH-TABLE.
        03  MONTH        PIC X(3)     OCCURS 6 TIMES.

77  MONTH-COLUMN         PIC 99       VALUE 0.
77  MONTH-ROW            PIC 99       VALUE 0.
77  COUNTER              PIC 99       VALUE 0.
77  RESULT               PIC 99       VALUE 0.


 PROCEDURE DIVISION.

*  TEST1:  Add to counter and call procedure to square and     *
*          display count.                                      *

 TEST1.
      MOVE 1 TO COUNTER.
         PERFORM 10 TIMES
            CALL "SQUARING-PROCEDURE" USING COUNTER
            ADD 1 TO COUNTER
         END-PERFORM.
```

(Continued)

Figure 7-3. Debug Example: Source File EXAMPLE_COB

(Continued)

```
 *  TEST2:  Create single column table for each month.        *

  TEST2.
       MOVE 1 TO MONTH-COLUMN.
       MOVE 1 TO MONTH-ROW.
          PERFORM 6 TIMES
          MOVE MONTH-LIST-1(MONTH-COLUMN:3) TO MONTH(MONTH-ROW)
          DISPLAY "THE MONTH IS " MONTH(MONTH-ROW)
          ADD 3 TO MONTH-COLUMN
          ADD 1 TO MONTH-ROW
          END-PERFORM.


 *  TEST3:  Create divide fault.                               *

  TEST3.
       DIVIDE DIVIDEND BY DIVISOR GIVING QUOTIENT.
       DISPLAY "ANSWER IS:  " QUOTIENT.
       STOP RUN.
  END PROGRAM EXAMPLE-COB.



 *  Subprogram SQUARING-PROCEDURE                              *


  IDENTIFICATION DIVISION.
  PROGRAM-ID.     SQUARING-PROCEDURE.
  ENVIRONMENT DIVISION.
  DATA DIVISION.
  WORKING-STORAGE SECTION.
  77  RESULT                          PIC 999       VALUE 0.

  LINKAGE SECTION.
  77  COUNTER                         PIC 99.

  PROCEDURE DIVISION USING COUNTER.
  START-SQUARING-PROCEDURE.
       MULTIPLY COUNTER BY COUNTER GIVING RESULT.
       DISPLAY COUNTER " TIMES" COUNTER " = " RESULT.
       EXIT PROGRAM.
```

Figure 7-3. Debug Sample: Source File EXAMPLE_COB

Preparing to Debug

After you create EXAMPLE_COB, you must prepare it for use with screen mode Debug. This requires preparing the screen mode environment and compiling EXAMPLE_COB for use with Debug. You can then execute it under Debug control. Do this as follows:

1. Assuming EXAMPLE_COB is contained in permanent file $USER.EXAMPLE_COB, prepare and compile EXAMPLE_COB using the COBOL command with the OPTIMIZATION_LEVEL=DEBUG and DEBUG_AIDS=DT compiler parameters by entering the following commands:

```
/change_interaction_style style=screen
/cobol input=$user.example_cob binary_object=lgo ..
../optimization_level=debug debug_aids=all
```

2. Execute EXAMPLE_COB under control of Debug by entering the following command:

```
/execute_task file=lgo debug_mode=on
```

   The source listing of EXAMPLE_COB is displayed in the Source window. The Debug functions are displayed at the bottom of the screen.

Display Screen Mode Commands

The function below are used to display helpful information about the Debugging enviornment:

HELP

Displays the Help window. Press a function key and a short explanation of the function's use appears in the Help window.

ZMIN

Used to display the source listing in the Source window.

ZMOUT

Used to display the COBOL program modules in the Source window.

Now perform the following steps to become familiar with the Debug functions:

1. Press the HELP key. The Help window is displayed.

2. Press each function key corresponding to a function displayed at the bottom of the screen. As you press each function key, a short explanation of the purpose of each function is displayed in the Help window.

3. Press RETURN. Exit HELP.

4. Press the ZMOUT function key. The modules of EXAMPLE_COB are displayed.

5. Press the ZMIN function key. The source listing of EXAMPLE_COB is displayed again. The ZMIN and ZMOUT function keys are used to display the source listing or the modules of the program, respectively.

Setting Breaks

It is often helpful to suspend program execution when debugging a program. The device for suspending execution of a program is called a break. In this sample session, the following functions are used to illustrate setting breaks:

FIRST

Displays the first screen of the source listing. Because FIRST is a lower priority function, it may not be assigned to a function key on terminals with only 16 function keys. Instead, FIRST is entered on the home line.

FWD

Scrolls forward to the next screen of text.

LOCATE

Prompts you to type in text, then searches the source listing for matching text. If a match is found, the cursor is moved to the line containing the matching text.

SETBRK

Sets an execution break on the line containing the cursor. The line is highlighted to show that it contains a break. Execution is suspended before the line containing the break is executed. Execution resumes with the statement on the line containing the break.

This section also uses the following item:

HOME

Moves the cursor to the home line. Line mode Debug commands can be entered on the home line for execution in screen mode Debug.

Perform the following steps to place two execution breaks in EXAMPLE_COB:

1. Press the LOCATE function key. At the top right hand corner of the screen, you are prompted for the text to be located.

2. Enter the following text exactly as it appears in EXAMPLE_COB:

   ```
   PERFORM 6
   ```

   The cursor is moved to the line:

   ```
   PERFORM 6 TIMES
   ```

3. Press the SETBRK function key. A break is set and the line containing the cursor is highlighted to show that it contains an execution break.

4. Use the down-arrow key to move the cursor to the line containing:

   ADD 1 TO MONTH-ROW

   If you do not see this line on your screen, press the FWD key. The next screen of the EXAMPLE_COB source listing is displayed. Use the down-arrow key to position the cursor on the correct line.

5. Press the SETBRK function key. The line is highlighted to show that it contains an execution break.

6. Press the FIRST function key. The first screen of the EXAMPLE_COB source listing is displayed in the Source window.

   If FIRST is not assigned to a function key, FIRST must be entered on the home line. To do this, press the HOME key. The cursor moves to the home line. Enter the following on the home line:

   first

   The first screen of the EXAMPLE_COB source listing is displayed in the Source window.

Debugging TEST1

Using Debug, you can execute a program one line or several lines at a time. Also, you can examine a variable's contents, change its contents, and execute code containing the variable several times. These capabilities are demonstrated in this sample session using the following functions:

CHAVAL

Prompts you to enter a variable name and the value you want it to contain, then changes the variable's contents to the new value.

GOTO

Moves the execution pointer to the line that contains the cursor. Execution resumes with the statement on this line.

HSPEED

Executes a program until a break is encountered or the program ends.

SEEVAL

Prompts you to enter a variable name, then displays the value of the variable in the Output window.

STEP1

Executes a program one line at a time.

STEPN

Executes N lines of a program, where N is an integer.

Perform the following steps to demonstrate the use of the CHAVAL, GOTO, HSPEED, SEEVAL, STEP1, STEPN:

1. Press the STEP1 function key. The first line of EXAMPLE_COB is executed, moving the execution arrow to the statement:

   ```
   MOVE 1 TO COUNTER.
   ```

2. Press the STEP1 function key again. The MOVE statement is executed; the execution arrow points to the statement:

   ```
   PERFORM 10 TIMES
   ```

3. Press the STEP1 function key six times. An iteration of TEST1 is executed one line at a time. The output generated by the iteration is displayed in the Output window.

4. Press the SEEVAL function key. A prompt to enter a variable name is printed in the upper right hand corner of the screen. Enter the name:

   counter

   The value of COUNTER is displayed in the Output window:

   counter=+2

   Thus, you can use SEEVAL to observe the contents of a variable.

5. Press the CHAVAL function key. A prompt for a variable name and its new value is displayed in the upper right hand corner of the screen; enter:

   counter=8

   The value of COUNTER is changed to 8.

6. Press the SEEVAL function key. When you are prompted for a variable name, enter:

   counter

   The following message is displayed in the Output window:

   counter=+8

   Thus, the change of COUNTER's value is verified.

7. Press the STEPN function key. In the upper right hand corner of the screen, you are prompted for the number of lines to execute; enter:

   5

STEPN executes 5 statements of TEST1. The output generated by this loop iteration is displayed in the Output window.

8. Press the SEEVAL function key. When you are prompted for a variable name, enter:

   counter

   The value of COUNTER is displayed in the Output window:

   counter=+9

   Therefore, the value given to COUNTER in step 5 is used by the PERFORM statement.

9. Use the up-arrow key to move the cursor to the line:

   MOVE 1 TO COUNTER.

10. Press the GOTO function key. The execution arrow moves to the line containing the cursor; execution resumes with this statement.

11. Press the HSPEED function key. Execution resumes from the MOVE statement; COUNTER is initialized to 0. Execution of EXAMPLE_COB continues until an execution break is encountered.

Debugging TEST2

After program execution is resumed in step 11 of TEST1, it stops at the break set on the PERFORM statement in TEST2. The following functions are used in TEST2 to illustrate more Debug capabilities:

BKW

Scrolls backward to the previous screen of text.

DELBRK

Deletes execution breaks.

HSPEED

Executes a program until a break is encountered or the program ends.

LSPEED

Executes a program one paragraph at a time.

This section also uses the following items:

HOME

Moves the cursor to the home line. Line mode Debug commands can be entered on the home line for execution in screen mode Debug.

DISPLAY_PROGRAM_VALUE

A line mode Debug command that displays the values of program variables.

Perform the following steps to learn how to execute loops one iteration at a time, execute line mode Debug commands, and scroll output data when using Debug:

1. Press the HSPEED function key. Execution stops at the break set on the last line of the PERFORM loop in TEST2; output generated by the loop is displayed in the Output window.

2. Press the HSPEED function key again. One iteration of the PERFORM loop is executed; execution stops at the break set at the statement, ADD 1 TO MONTH_ROW. Each time HSPEED is used, an iteration of the loop is performed. By using strategically placed execution breaks, as in this example, a loop can be executed one iteration at a time.

3. Press the HSPEED function key. One more loop iteration is performed.

4. Press the HOME key. The cursor moves to the home line.

5. Enter the line mode Debug command:

   ```
   display_program_value name=$all
   ```

   The values of all variables in EXAMPLE_COB are displayed in the Output window. Thus, line mode Debug commands can be used in screen mode Debug by entering them on the home line. For more information about using line mode Debug commands see the Debug Usage Manual.

6. Press the DELBRK key. The execution break is deleted. The highlight is removed from the line when the break is removed.

7. Press the down-arrow key until the cursor is inside of the Output window.

8. Press the BKW key. The data in the Output window scrolls backward. When the cursor is contained within the Output window, you can use the BKW and FWD keys to scroll backward and forward through the data in the window .

9. Press the LSPEED function key. The execution of EXAMPLE_COB resumes, stopping when TEST2 is exited. The execution arrow points to the beginning of TEST3.

Debugging TEST3

After resuming execution of EXAMPLE_COB in step 9 of section TEST2, execution stops at the begining of TEST3. In TEST3, Debug is presented with an execution error. The following functions are used in this sample session to demonstrate how Debug can be used when an exectuion error is encountered:

CHAVAL

Prompts you to enter a variable name and the value you want it to contain, then changes the variable's contents to the new value.

GOTO

Moves the execution pointer to the line that contains the cursor. Execution resumes with the statement on this line.

SEEVAL

Prompts you to enter a variable name, then displays the value of the variable in the Output window.

STEP1

Executes a program one line at a time.

TRACE

Displays the chain of program units that were called to reach the current line.

QUIT

Used to exit Debug.

Perform the following steps to finish the example:

1. Press the STEP1 function key. The first statement of TEST3 is executed and the execution arrow points to the next line to be executed:

   ```
   DIVIDE DIVIDEND BY DIVISOR GIVING QUOTIENT.
   ```

   Since this statement has not been executed, you can use CHAVAL to change the value of any variable in the statement before continuing execution. In this example, such a change will not be made.

2. Press the STEP1 function key again. The DIVIDE statement is executed, execution of EXAMPLE_COB halts, and the following message flashes in the top right hand corner of the screen:

   ```
   divide_fault
   ```

3. Press the SEEVAL function key. When you are prompted for a variable name, enter:

   divisor

   The following message is displayed in the Output window:

   divisor=+0

   A division by zero caused the execution error.

4. Press the CHAVAL function key. When you are prompted, enter:

   DIVISOR=1

   The value of DIVISOR is changed to 1.

5. Press the SEEVAL function key. When you are prompted, enter:

   DIVISOR

   The following text is displayed in the Output window:

   divisor=+1

   The change to DIVISOR is verified.

6. Press the GOTO function key. The execution arrow points at the DIVISION statement and program execution resumes with this statement.

7. Press the STEP1 function key. The DIVISION statement is executed. Therefore, the GOTO and CHAVAL functions can be used in concert to recover from execution errors. However, to correct execution errors permanently, you must exit Debug, edit the program, and recompile it.

8. Press the STEP1 function key again. The result of the DIVISION statement is displayed in the Output window.

9. Press the STEP1 function key. EXAMPLE_COB ends and the following message is displayed in the Output window:

   DEBUG: The status at termination was: NORMAL.

10. Press the TRACE function key. The call chain of program units that led to the last executed statement in TEST3 is displayed.

11. Press the QUIT function key. Exit Debug.

Now that you have concluded this example, you should be able to begin using screen mode Debug to debug your COBOL programs.

Debugging a CYBIL Program

This example is presented as a sequence of steps. To get the most benefit, you should create the sample program illustrated in figure 7-4, then perform each step as you read it. The sample program, EXAMPLE_CYB, provides three test cases to debug. Each test case in EXAMPLE_CYB is used to demonstrate the application of some Debug function keys. After you work this example, you will be able to debug your CYBIL programs using Screen Mode Debug.

EXAMPLE_CYB is divided into the following test cases:

TEST1

A loop that increments a counter and then calls a procedure to square and display the count. TEST1 demonstrates the use of the CHAVAL, GOTO, HSPEED, SEEVAL, STEP1 and STEPN functions.

TEST2

A loop that builds a 6-row table of 3-character strings. Input to the table is an 18-character list for the months JAN through JUN. TEST2 moves three characters at a time from the character list to the table, and displays each entry. TEST2 shows how to step through loops, use line mode Debug commands in screen mode Debug, and how to scroll through Debug and program output data.

TEST3

A division test that results in a divide fault. TEST3 demonstrates how Debug handles execution errors.

```
MODULE example_cyb;

{  Copy I/O procedures. }

      TYPE

         column = array [1..3] of string(3),
         twodim_array = array [1..6] of column;

      CONST

         maximum_record_length = 40;

      VAR

{  Declare program variables. }

         divisor       : real           := 0.0,
         dividend      : real           := 100.0,
         quotient      : real,
         cntr          : integer,
         result        : integer,
         month         : twodim_array,
         month_list    : string (18)    := 'JANFEBMARAPRMAYJUN',
         month_row     : integer        := 0,
         l             : integer        := 10,
         i             : integer,

{  Declare variables for I/O. }

         lfn           : amt$local_file_name,
         o             : amt$file_identifier,
         s             : ost$status,
         f             : amt$file_byte_address,
         newline       : string (90),
         m1            : string (7)     := ' times ',
         m2            : string (3)     := ' = ',
         m3            : string (16)    := ' The month is:  ',
         m4            : string (19)    := ' The quotient is:  ';

   PROGRAM main;

{  These calls specify file attributes and open files. }

         lfn := '$OUTPUT';
         amp$open (lfn, amc$record, NIL, o, s);
```

(Continued)

Figure 7-4 Debug Example: Source File EXAMPLE_CYB

(Continued)

```
  {  TEST1:  Add to counter and call procedure SQUARE to square }
  {          and display count.                                  }

      FOR cntr := 1 TO 10 DO
        square (cntr,result);
        stringrep(newline,1,' ',cntr:3,m1,cntr:3,m2,result:4);
        amp$put_next(o,^newline,maximum_record_length,f,s);
      FOREND:


  {  TEST2:  Create single column table for each month. }

      WHILE month_row < 6 DO

        FOR i := 1 TO 3 DO
          month[month_row][i] := month_list(month_row*3+1,3);
        FOREND;

        stringrep(newline,1,' ',m3,month[month_row][i]:8);
        amp$put_next(o,^newline,maximum_record_length,f,s);
        month_row := month_row + 1;

      WHILEND;


  {  TEST3:  Create divide fault. }

      quotient := dividend / divisor;
      stringrep(newline,1,' ',m4:19,quotient:6:1);
      amp$put_next(o,^newline,maximum_record_length,f,s);

    PROCEND main;

  {  Procedure for squaring numbers. }

    PROCEDURE [XDCL] square (
           a : integer;
      VAR  b : integer;

      b := a * a;

    PROCEND square;

 MODEND example_cyb;
```

Figure 7-4. Debug Example: Source File EXAMPLE_CYB

Preparing to Debug

After you create EXAMPLE_CYB, you must prepare it for use with screen mode Debug. This requires preparing the screen mode environment and compiling EXAMPLE_CYB for use with Debug. You can then execute it under screen mode Debug control. Do this as follows:

1. EXAMPLE_CYB calls several file interface procedures that must be expanded through commands provided in the Source Code Utility (SCU) before the source code can be compiled. To do this, enter the following commands:

```
/create_source_library
/scu_create_deck deck=example_cyb modification=ml ..
../source=$user.example_cyb
/scu_expand_deck deck=example_cyb ..
../alternate_base=$system.cybil.osf$program_interface ..
../compile=$user.compile
```

2. Prepare for screen mode debugging and compile EXAMPLE_CYB now contained in permanent file $USER.COMPILE for use with Debug by entering the following commands:

```
/change_interaction_style style=screen
/cybil input=$user.compile binary=lgo ..
../optimization_level=debug debug_aids=all
```

3. Execute under control of Debug by entering the following command:

```
/execute_task file=lgo debug_mode=on
```

   The source module of EXAMPLE_CYB is displayed in the Source window. Debug functions are displayed at the the bottom of the screen.

Displaying Screen Mode Commands

The functions below are used to display helpful information about the Debugging enviornment:

HELP

Displays the Help window. Press a function key and a short explanation of the function's use appears in the Help window.

ZMIN

Used to display the source listing in the Source window.

Now perform the following steps to become familiar with the Debug functions:

1. Press the HELP key. The Help window is displayed.

2. Press each function key corresponding to the functions displayed at the bottom of the screen. As you press each function key, a short explanation of the purpose of each function is displayed in the Help window.

3. Press RETURN. Exit HELP.

4. Press the ZMIN function key. The following message is displayed in the upper right hand corner of the screen:

   Enter compiler input file for EXAMPLE_CYB

5. Enter the source file name:

   $user.compile

   The source listing of EXAMPLE_CYB is displayed in the Source window. Also, some new functions are displayed at the bottom of the screen.

6. Press the HELP key. The Help window is displayed again.

7. Press each function key corresponding to the new functions displayed at the bottom of the screen. As you press each function key, a short explanation of the purpose of each new function is displayed in the Help window.

8. Press RETURN. Exit HELP.

Setting Breaks

It is often helpful to suspend program execution when debugging a program. The device for suspending execution of a program is called a break. In this sample session, the following functions are used to illustrate setting breaks.

FIRST

Displays the first screen of the source listing. Because FIRST is a lower priority function, it may not be assigned to a function key on therminals with only 16 function keys. Instead, FIRST is entered on the home line.

FWD

Scrolls forward to the next screen of text.

LOCATE

Prompts you to type in text, then searches the source listing for matching text. If a match is found, the cursor is moved to the line containing the matching text.

SETBRK

Sets an execution break on the line containing the cursor. The line is highlighted to show that it contains a break. Execution is suspended before the line containing the break is executed. Execution resumes with the statement on the line containing the break.

This section also uses the following item:

HOME

Moves the cursor to the home line. Line mode Debug commands can be entered on the home line for execution in screen mode Debug.

Perform the following steps to place three execution breaks in EXAMPLE_CYB:

1. Press the LOCATE function key. At the top right hand corner of the screen, you are prompted for the text to be located.

2. Enter the following text exactly as it appears in EXAMPLE_CYB:

   ```
   WHILE
   ```

   The cursor is moved to the line:

   ```
   WHILE month_row < 6 DO
   ```

3. Press the SETBRK function key. A break is set and the line containing the cursor is highlighted to show that it contains an execution break.

4. Use the down-arrow key to move the cursor to the line:

   ```
   month_row := month_row + 1;
   ```

   If you do not see this line on your screen, press the FWD key. The next screen of the EXAMPLE_CYB source listing is displayed. Use the down-arrow key to position the cursor on the correct line.

5. Press the SETBRK function key. The line is highlighted to show that it contains an execution break.

6. Use the down-arrow key to move the cursor to the line:

   ```
   quotient := dividend / divisor;
   ```

   If you do not see this line on your screen, press the FWD key. The next screen of the EXAMPLE_CYB source listing is displayed. Use the down-arrow key to position the cursor on that line.

7. Press the SETBRK function key. The line is highlighted to show that it contains an execution break.

8. Press the FIRST function key. The first screen of the EXAMPLE_CYB source listing is displayed in the Source window.

   If FIRST is not assigned to a function key, FIRST must be entered on the home line. To do this, press the HOME key. The cursor moves to the home line. Enter the following on the home line:

   ```
   first
   ```

   The first screen of the EXAMPLE_CYB source listing is displayed in the Source window.

Debugging TEST1

Using Debug, you can execute a program one line or several lines at a time. Also, you can examine a variable's contents, change its contents, and execute code containing the variable several times. These capabilities are demonstrated in this sample session using the following functions:

CHAVAL

Prompts you to enter a variable name and the value you want it to contain, then changes the variable's contents to the new value.

GOTO

Moves the execution pointer to the line that contains the cursor. Execution resumes with the first statement on this line.

HSPEED

Executes a program until a break is encountered or the program ends.

SEEVAL

Prompts you to enter a variable name, then displays the value of the variable in the Output window.

STEP1

Executes a program one line at a time.

STEPN

Executes N lines of a program, where N is an integer.

Perform the following steps to demonstrate the use of the CHAVAL, GOTO, HSPEED, SEEVAL, STEP1, STEPN:

1. Press the STEP1 function key. The statement:

   ```
   lfn := '$OUTPUT';
   ```

   is executed; the execution arrow now points to the statement:

   ```
   amp$open(lfn,amc$record,NIL,o,s);
   ```

2. Press the STEP1 function key again. The AMP$OPEN procedure is executed; moving the execution arrow to the first executable line in TEST1:

   ```
   FOR cntr := 1 TO 10 DO
   ```

3. Press the STEP1 function key seven times. An iteration of FOR loop is executed one statement at a time. The output generated by the iteration is displayed in the Output window.

4. Press the SEEVAL function key. A prompt to enter a variable name is printed in the upper right hand corner of the screen. Enter the name:

   ```
   cntr
   ```

   The value of CNTR is displayed in the Output window:

   ```
   cntr = 2
   ```

   Thus, you can use SEEVAL to observe the contents of a variable.

5. Press the CHAVAL function key. A prompt for a variable name and its new value is displayed in the upper right hand corner of the screen; enter:

   ```
   cntr=8
   ```

   The value of CNTR is changed to 8.

6. Press the SEEVAL function key. When you are prompted for a variable name, enter:

   ```
   cntr
   ```

   The following message is displayed in the Output window:

   ```
   cntr = 8
   ```

   Thus, the change of CNTR's value is verified.

7. Press the STEPN function key. In the upper right hand corner of the screen, you are prompted for the number of lines to execute; enter:

   6

   STEPN executes 6 lines. The output generated by this loop iteration is displayed in the Output window.

8. Press the SEEVAL function key. When you are prompted for a variable name, enter:

   cntr

   The value of COUNTER is displayed in the Output window:

   cntr = 3

   Only the value of CNTR passed to the SQUARE call was changed. The value of a FOR loop control variable cannot be changed once the loop has been entered. Therefore, the value of CNTR used by this FOR loop remains unchanged.

9. Use the up-arrow key to move the cursor to the line:

   FOR cntr := 1 TO 10 DO

10. Press the GOTO function key. The execution arrow moves to the line containing the cursor; execution resumes at this line.

11. Press the HSPEED function key. Execution resumes from the FOR statement. Since the FOR loop is executed anew, CNTR is initialized to 1. Execution of EXAMPLE_CYB continues until an execution break is encountered.

Debugging TEST2

After program execution is resumed in step 12 of TEST1, execution stops at the break set on the first statement in TEST2. The following functions are used in TEST2 to illustrate more Debug capabilities:

BKW

Scrolls backward to the previous screen of text.

CHAVAL

Prompts you to enter a variable name and the value you want it to contain, then changes the variable's contents to the new value.

DELBRK

Deletes execution breaks.

FWD

Scrolls forward to the next screen of text.

HSPEED

Executes a program until a break is encountered or the program ends.

SEEVAL

Prompts you to enter a variable name, then displays the value of the variable in the Output window.

This section also uses the following items:

HOME

Moves the cursor to the home line. Line mode Debug commands can be entered on the home line for execution in screen mode Debug.

DISPLAY_PROGRAM_VALUE

A line mode Debug command that displays the values of program variables.

Perform the following steps to learn how to execute loops one iteration at a time, execute line mode Debug commands, and scroll output data when using Debug:

1. Press the HSPEED function key. Execution stops at the break set on the last line of the WHILE loop; output generated by the loop is displayed in the Output window.

2. Press the HSPEED function key again. One iteration of the WHILE loop is executed; execution stops at the break set in the WHILE loop again. Each time HSPEED is used, an iteration of the loop is performed. By using strategically placed execution breaks, as in this example, a loop can be executed one iteration at a time.

3. Press the HSPEED function key. One more loop iteration is performed.

4. Press the SEEVAL function key. When you are prompted, enter:

   ```
   month_row
   ```

   The following message is displayed in the Output window:

   ```
   month_row = 2
   ```

5. Press the CHAVAL function key. When you are prompted, enter:

   ```
   month_row=4
   ```

6. Press the SEEVAL function key. When you are prompted, enter:

   ```
   month_row
   ```

   The following message is displayed in the Output window:

   ```
   month_row = 4
   ```

   Thus, the change to MONTH_ROW is verified.

7. Press the HSPEED function key. One iteration of the WHILE loop is executed.

8. Press the SEEVAL function key. When you are prompted, enter:

   month_row

   the following message is then displayed in the Output window:

   month_row = 5

   The value given to MONTH_ROW in step 5 is used by the WHILE loop.

9. Press the HOME key. The cursor moves to the home line.

10. Enter the line mode Debug command:

    display_program_value name=$all

    The values of all variables declared in EXAMPLE_CYB are displayed in the Output window. Thus, line mode Debug commands can be used in screen mode Debug by entering them on the home line. For more information about using line mode Debug commands see the Debug Usage Manual.

11. Press the DELBRK key. The execution break is deleted. The highlight is removed from the line when the break is removed.

12. Press the down-arrow key until the cursor is inside of the Output window.

13. Press the BKW key. The data in the Output window scrolls backward. When the cursor is contained within the Output window, you can use the BKW and FWD keys to scroll backward and forward through the data in the window .

14. Press the HSPEED function key. The execution of EXAMPLE_CYB resumes, stopping at the next break. The execution arrow points to the first statement in TEST3.

Debugging TEST3

After resuming execution of EXAMPLE_CYB in step 14 of section TEST2, execution stops at the begining of TEST3. In TEST3, Debug is presented with an execution error. The following functions are used in this sample session to demonstrate how Debug can be used when an execution error is encountered:

CHAVAL

Prompts you to enter a variable name and the value you want it to contain, then changes the variable's contents to the new value.

GOTO

Moves the execution pointer to the line that contains the cursor. Execution resumes with the first statement on this line.

SEEVAL

Prompts you to enter a variable name, then displays the value of the variable in the Output window.

STEP1

Executes a program one line at a time.

QUIT

Used to leave Debug.

Perform the following steps to finish the example:

1. Press the STEP1 function key. Execution halts, and the following message flashes in the top right hand corner of the screen:

   divide_fault

2. Press the SEEVAL function key. When you are prompted for a variable name, enter:

   divisor

   The following message is displayed in the Output window:

   divisor = 0.

   A division by zero caused the execution error.

3. Press the CHAVAL function key. When you are prompted, enter:

    divisor=1.0

   The value of DIVISOR is changed to 1.

4. Press the SEEVAL function key. When you are prompted, enter:

    divisor

   The following text is displayed in the Output window:

    divisor=1.00000000000000E+0000

   The change to DIVISOR is verified.

5. Press the GOTO function key. The execution arrow points to the DIVISION statement, so program execution resumes with this statement.

6. Press the STEP1 function key. The DIVISION statement is executed. Therefore, the GOTO and CHAVAL functions can be used in concert to recover from execution errors. However, to correct execution errors permanently, you must exit Debug, edit the program, and recompile it.

7. Press the STEP1 function key two more times. The result of the DIVISION statement is displayed in the Output window.

9. Press the STEP1 function key. EXAMPLE_CYB ends and the following message is displayed in the Output window:

    DEBUG: The status at termination was: NORMAL.

12. Press the QUIT function key. Exit Debug.

Now that you have concluded this example, you should be able to begin using screen mode Debug to debug your CYBIL programs.

Debugging a FORTRAN Version 1 or FORTRAN Version 2 Program

This example is presented as a sequence of steps. To get the most benefit, you should create the sample program illustrated in figure 7-5, then perform each step as you read it. The sample program, FORTEX, provides three test cases to debug and can be compiled for FORTRAN Version 1 or FORTRAN Version 2. Each test case in FORTEX is used to demonstrate the application of some Debug function keys. After you work this example, you will be able to debug your FORTRAN programs using Screen Mode Debug.

EXAFORT is divided into the following test cases:

TEST1

A loop that increments a counter and then calls a subprogram to square and display the count. TEST1 demonstrates the use of the CHAVAL, GOTO, HSPEED, SEEVAL, STEP1 and STEPN functions.

TEST2

A loop that builds a 6-row table of 3-character strings. Input to the table is an 18-character list for the months JAN through JUN. TEST2 moves three characters at a time from the character list to the table and displays each entry. TEST2 shows how to step through loops, use line mode Debug commands in screen mode Debug, and how to scroll through Debug and program output data.

TEST3

A division test that results in a divide fault. TEST3 demonstrates how Debug handles execution errors.

```
      PROGRAM EXAFORT

      CHARACTER TABLE(6)*3, LIST*18
      REAL DIVDEND, DIVISOR, QUOTENT, COUNTER, RESULT
      INTEGER COLUMN, ROW
      DATA DIVDEND, DIVISOR, COLUMN/100.,0.,1/
      DATA LIST/'JANFEBMARAPRMAYJUN'/

***************************************************************
*  TEST1:  Add to counter and call procedure to square and    *
*          display count.                                     *
***************************************************************

      DO 10 COUNTER = 1,10
         CALL SQUARE (COUNTER)
 10   CONTINUE

***************************************************************
*  TEST2:  Create single column table for each month.         *
***************************************************************

      DO 20 ROW = 1,6
         TABLE(ROW) = LIST(COLUMN : COLUMN + 2)
         PRINT*, 'THE MONTH IS:  ', TABLE(ROW)
         COLUMN = COLUMN + 3
 20   CONTINUE

***************************************************************
*  TEST3:  Create divide fault.                               *
***************************************************************

      QUOTENT = DIVDEND / DIVISOR
      PRINT*, 'ANSWER IS:  ', QUOTENT

      END

***************************************************************
*  Subroutine SQUARE                                          *
***************************************************************

      SUBROUTINE SQUARE (COUNTER)
         RESULT = 0.
         RESULT = COUNTER * COUNTER
         PRINT*, COUNTER, ' TIMES ', COUNTER, ' = ', RESULT
      END
```

Figure 7-5. Debug Example: Source File EXAFORT

Preparing to Debug

After you create EXAFORT, you must prepare it for use with screen mode Debug. This requires preparing the screen mode environment and compiling EXAFORT for use with Debug. You can then execute it under screen mode Debug control. Do this as follows:

1. Assuming EXAFORT is contained in permanent file $USER.EXAFORT, prepare and compile EXAFORT using the FORTRAN or VECTOR_FORTRAN command with the OPTIMIZATION_LEVEL=DEBUG and DEBUG_AIDS=ALL compiler parameters by entering the following commands:

```
/change_interaction_style style=screen
/fortran input=$user.exafort binary_object=lgo ..
../optimization_level=debug debug_aids=all
```

   or, if compiling a FORTRAN Version 2 program:

```
/change_interaction_style style=screen
/vector_fortran input=$user.exafort binary_object=lgo ..
../optimization_level=debug debug_aids=all
```

2. Execute EXAFORT under control of Debug by entering the following command:

```
/execute_task file=lgo debug_mode=on
```

   The source listing of EXAFORT is displayed in the Source window. The Debug functions are displayed at the bottom of the screen.

Display Screen Mode Commands

The function below is used to display helpful information about the Debugging enviornment:

HELP

Displays the Help window. Press a function key and a short explanation of the function's use appears in the Help window.

ZMIN

Used to display the source listing in the Source window.

ZMOUT

Used to display the FORTRAN program modules in the Source window.

Now perform the following steps to become familiar with the Debug functions:

1. Press the HELP key. The Help window is displayed.

2. Press each function key corresponding to a function displayed at the bottom of the screen. As you press each function key, a short explanation of the purpose of each function is displayed in the Help window.

3. Press RETURN. Exit HELP.

4. Press the ZMOUT function key. The modules of EXAFORT are displayed.

5. Press the ZMIN function key. The source listing of EXAFORT is displayed again. The ZMIN and ZMOUT function keys are used to display the source listing or the modules of the program, respectively.

Setting Breaks

It is often helpful to suspend program execution when debugging a program. The Debug device for suspending execution of a program is called a break. In this sample session, the following functions are used to illustrate setting breaks:

FIRST

Displays the first screen of the source listing. Because FIRST is a lower priority function, it may not be assigned to a function key on therminals with only 16 function keys. Instead, FIRST is entered on the home line.

FWD

Scrolls forward to the next screen of text.

LOCATE

Prompts you to type in text, then searches the source listing for matching text. If a match is found, the cursor is moved to the line containing the matching text.

SETBRK

Sets an execution break on the line containing the cursor. The line is highlighted to show that it contains a break. Execution is suspended before the line containing the break is executed. Execution resumes with the statement on the line containing the break.

This section also uses the following item:

HOME

Moves the cursor to the home line. Line mode Debug commands can be entered on the home line for execution in screen mode Debug.

Debugging a FORTRAN Version 1 or FORTRAN Version 2 Program

Perform the following steps to place three execution breaks in EXAFORT:

1. Press the LOCATE function key. At the top right hand corner of the screen, you are prompted for the text to be located.

2. Enter the following text exactly as it appears in EXAFORT:

   ```
   DO 20
   ```

   The cursor is moved to the line:

   ```
   DO 20 ROW = 1,6
   ```

3. Press the SETBRK function key. A break is set and the line containing the cursor is highlighted to show that it contains an execution break.

4. Use the down-arrow key to move the cursor to the line containing:

   ```
   COLUMN = COLUMN + 3
   ```

   If you do not see this line on your screen, press the FWD key. The next screen of the EXAFORT source listing is displayed. Use the down-arrow key to position the cursor on the correct line.

5. Press the SETBRK function key. The line is highlighted to show that it contains an execution break.

6. Use the down-arrow key to move the cursor to the line:

   ```
   QUOTENT = DIVDEND / DIVISOR
   ```

   If you do not see this line on your screen, press the FWD key. The next screen of the EXAFORT source listing is displayed. Use the down-arrow key to position the cursor on the correct line.

7. Press the SETBRK function key. The line is highlighted to show that it contains an execution break.

8. Press the FIRST function key. The first screen of the EXAFORT source listing is displayed in the Source window.

   If FIRST is not assigned to a function key, FIRST must be entered on the home line. To do this, press the HOME key. The cursor moves to the home line. Enter the following on the home line:

   ```
   first
   ```

   The first screen of the EXAFORT source listing is displayed in the Source window.

Debugging TEST1

Using Debug, you can execute a program one line or several lines at a time. Also, you can examine a variable's contents, change its contents, and execute code containing the variable several times. These capabilities are demonstrated in this sample session using the following functions:

CHAVAL

Prompts you to enter a variable name and the value you want it to contain, then changes the variable's contents to the new value.

GOTO

Moves the execution pointer to the line that contains the cursor. Execution resumes with the statement on this line.

HSPEED

Executes a program until a break is encountered or the program ends.

SEEVAL

Prompts you to enter a variable name, then displays the value of the variable in the Output window.

STEP1

Executes a program one line at a time.

STEPN

Executes N lines of a program, where N is an integer.

Debugging a FORTRAN Version 1 or FORTRAN Version 2 Program

Perform the following steps to demonstrate the use of the CHAVAL, GOTO, HSPEED, SEEVAL, STEP1, STEPN:

1. Press the STEP1 function key. The first statement of EXAMPLE_FORT is executed, moving the execution arrow to the statement:

   ```
   DO 10 COUNTER = 1,10
   ```

2. Press the STEP1 function key again. The DO statement is executed; the execution arrow points to the statement:

   ```
   CALL SQUARE (COUNTER)
   ```

3. Press the STEP1 function key six times. An iteration of TEST1 is executed one line at a time. The output generated by the iteration is displayed in the Output window.

4. Press the SEEVAL function key. A prompt to enter a variable name is printed in the upper right hand corner of the screen. Enter the name:

   ```
   counter
   ```

   The value of COUNTER is displayed in the Output window:

   ```
   counter = 2.
   ```

   Thus, you can use SEEVAL to observe the contents of a variable.

5. Press the CHAVAL function key. A prompt for a variable name and its new value is displayed in the upper right hand corner of the screen; enter:

   ```
   counter=8
   ```

   The value of COUNTER is changed to 8.

6. Press the SEEVAL function key. When you are prompted for a variable name, enter:

   ```
   counter
   ```

   The following message is displayed in the Output window:

   ```
   counter = 8.
   ```

   Thus, the change of COUNTER's value is verified.

7. Press the STEPN function key. In the upper right hand corner of the screen, you are prompted for the number of lines to execute; enter:

   ```
   6
   ```

   STEPN executes 6 statements of TEST1. The output generated by this loop iteration is displayed in the Output window.

8. Press the SEEVAL function key. When you are prompted for a variable name, enter:

   ```
   counter
   ```

   The value of COUNTER is displayed in the Output window:

   ```
   counter = 9.
   ```

   Therefore, the value given to COUNTER in step 5 is used by the DO statement.

9. Use the up-arrow key to move the cursor to the line:

   ```
   DO 10 COUNTER = 1,10
   ```

10. Press the GOTO function key. The execution arrow moves to the line containing the cursor; execution resumes with this statement.

11. Press the HSPEED function key. Execution resumes from the DO statement; COUNTER is initialized to 1. Execution of EXAFORT continues until an execution break is encountered.

Debugging TEST2

After program execution is resumed in step 11 of TEST1, it stops at the break set on the DO statement in TEST2. The following functions are used in TEST2 to illustrate more Debug capabilities:

BKW

Scrolls backward to the previous screen of text.

DELBRK

Deletes execution breaks.

HSPEED

Executes a program until a break is encountered or the program ends.

This section also uses the following items:

HOME

Press the HOME key to move the cursor to the home line. line mode Debug commands can be entered on the home line for execution in screen mode Debug.

DISPLAY_PROGRAM_VALUE

A line mode Debug command that displays the values of program variables.

Perform the following steps to learn how to execute loops one iteration at a time, execute line mode Debug commands, and scroll output data when using Debug:

1. Press the HSPEED function key. Execution stops at the break set on the last line of the DO loop in TEST2; output generated by the loop is displayed in the Output window.

2. Press the HSPEED function key again. One iteration of the DO loop is executed; execution stops at the break set at the COLUMN = COLUMN + 3 statement. Each time HSPEED is used, an iteration of the loop is performed. By using strategically placed execution breaks, as in this example, a loop can be executed one iteration at a time.

3. Press the HSPEED function key. One more loop iteration is performed.

4. Press the HOME key. The cursor moves to the home line.

5. Enter the line mode Debug command:

   ```
   display_program_value name=$all
   ```

   The values of all variables in EXAFORT are displayed in the Output window. Thus, line mode Debug commands can be used in screen mode Debug by entering them on the home line. For more information about using line mode Debug commands see the Debug Usage Manual.

6. Press the DELBRK key. The execution break is deleted. The highlight is removed from the line when the break is removed.

7. Press the down-arrow key until the cursor is inside of the Output window.

8. Press the BKW key. The data in the Output window scrolls backward. When the cursor is contained within the Output window, you can use the BKW and FWD keys to scroll backward and forward through the data in the window.

9. Press the HSPEED function key. The execution of EXAFORT resumes, stopping when the line containing the third break is reached. The execution arrow points to the first statement of TEST3.

Debugging TEST3

After resuming execution of EXAFORT in step 9 of section TEST2, execution stops at the begining of TEST3. In TEST3, Debug is presented with an execution error. The following functions are used in this sample session to demonstrate how Debug can be used when an exectuion error is encountered:

CHAVAL

Prompts you to enter a variable name and the value you want it to contain, then changes the variable's contents to the new value.

GOTO

Moves the execution pointer to the line that contains the cursor. Execution resumes with the statement on this line.

SEEVAL

Prompts you to enter a variable name, then displays the value of the variable in the Output window.

STEP1

Executes a program one line at a time.

QUIT

Used to exit Debug.

Perform the following steps to finish the example:

1. Press the STEP1 function key again. The DIVISION statement is executed, execution of EXAFORT halts, and the following message flashes in the top right hand corner of the screen:

   divide_fault

2. Press the SEEVAL function key. When you are prompted for a variable name, enter:

   divisor

   The following message is displayed in the Output window:

   divisor = 0.

   A division by zero caused the execution error.

3. Press the CHAVAL function key. When you are prompted, enter:

   ```
   divisor=1
   ```

   The value of DIVISOR is changed to 1.

4. Press the SEEVAL function key. When you are prompted, enter:

   ```
   divisor
   ```

   The following text is displayed in the Output window:

   ```
   divisor = 1.
   ```

   The change to DIVISOR is verified.

5. Press the GOTO function key. The execution arrow points at the DIVISION statement and program execution resumes with this statement.

6. Press the STEP1 function key. The DIVISION statement is executed. Therefore, the GOTO and CHAVAL functions can be used in concert to recover from execution errors. However, to correct execution errors permanently, you must exit Debug, edit the program, and recompile it.

7. Press the STEP1 function key again. The result of the DIVISION statement is displayed in the Output window.

8. Press the STEP1 function key. EXAFORT ends and the following message is displayed in the Output window:

   ```
   DEBUG: The status at termination was: NORMAL.
   ```

9. Press the QUIT function key. Exit Debug.

Now that you have concluded this example, you should be able to begin using screen mode Debug to debug your FORTRAN or FORTRAN Version 2 programs.

Debugging a Pascal Program

This example is presented as a sequence of steps. To get the most benefit, you should create the sample program illustrated in figure 7-6, then perform each step as you read it. The sample program, EXAMPLE_PAS, provides three test cases to debug. Each test case in EXAMPLE_PAS is used to demonstrate the application of some Debug function keys. After you work this example, you will be able to debug your Pascal programs using Screen Mode Debug.

EXAMPLE_PAS is divided into the following test cases:

TEST1

A procedure that increments a counter and then calls a procedure to square and display the count. TEST1 demonstrates the use of the CHAVAL, GOTO, HSPEED, SEEVAL, STEP1, and STEPN functions.

TEST2

A loop that builds a 6-row table of 3-character strings. Input to the table is an 18-character list for the months JAN through JUN. TEST2 moves three characters at a time from the character list to the table, and displays each entry. TEST2 shows how to step through loops, use line mode Debug commands in screen mode Debug, and how to scroll through Debug and program output data.

TEST3

A division test that results in a divide fault. TEST3 demonstrates how Debug handles execution errors.

```
PROGRAM EXAMPLE_PAS (output);

{ The main program EXAMPLE_PAS makes a call to each of the }
{ three test cases. }

  VAR
    divisor, dividend, quotient : real;
    month                       : ARRAY[1..6, 1..3] OF char;
    month_list                  : string(18);

  VALUE
    divisor    = 0.0;
    dividend   = 100.0;
    month_list = 'JANFEBMARAPRMAYJUN';

  PROCEDURE squaring_procedure(counter : integer);

    VAR
      result : integer;
    BEGIN
      result := counter * counter;
      writeln(' ',counter:2,' times',counter:2,' =',result:2);
    END;

{ TEST1:  Add to counter and call procedure to square and }
{ display count. }

PROCEDURE test1;

  VAR
    counter : integer;

  BEGIN {test1}
    FOR counter := 1 TO 10 DO
      squaring_procedure (counter);
  END;  {test1}
```

(Continued)

Figure 7-6. Debug Example: Source File EXAMPLE_PAS

(Continued)

```
{ TEST2:  Create single column table for each month. }

PROCEDURE test2;

  VAR
    i, month_row : integer;

  VALUE
    month_row = 0;

  BEGIN {test2}
    WHILE month_row < 6 DO
      BEGIN
        write(' The month is');
        FOR i := 1 TO 3 DO
          BEGIN
            month[month_row][i]:=month_list[month_row*3+1];
            write(month[month_row] [i]);
          END;
        writeln(' ');
        month_row := month_row + 1;
      END;
  END;  {test2}

PROCEDURE test3;

{ TEST3:  Create divide fault. }

  BEGIN  {test3}
    quotient := dividend / divisor;
    writeln(' The quotient is:',quotient:4:2);
  END;  {test3}

{ Main program code follows. }

BEGIN  {Main}
  test1;
  test2;
  test3;
END.
```

Figure 7-6. Debug Example: Source File EXAMPLE_PAS

Preparing to Debug

After you create EXAMPLE_PAS, you must prepare it for use with screen mode Debug. This requires preparing the screen mode environment and compiling EXAMPLE_PAS for use with Debug. You can then execute it under screen mode Debug control. Do this as follows:

1. Assuming EXAMPLE_PAS is contained in permanent file $USER.EXAMPLE_PAS, prepare and compile EXAMPLE_PAS using the Pascal command with the OPTIMIZATION_LEVEL=DEBUG and DEBUG_AIDS=DT compiler parameters by entering the following commands:

```
/change_interaction_style style = screen
/pascal input = $user.EXAMPLE_PAS binary = lgo ..
../optimization_level = debug debug_aids = all
```

2. Execute EXAMPLE_PAS under control of Debug by entering the following command:

```
/execute_task file = lgo debug_mode = on
```

The source module of EXAMPLE_PAS is displayed in the Source window. Debug functions are displayed at the the bottom of the screen.

Displaying Screen Mode Commands

The functions below are used to display helpful information about the Debugging enviornment:

HELP

Displays the Help window. Press a function key and a short explanation of the function´s use appears in the Help window.

ZMIN

Used to display the source listing in the Source window.

Now perform the following steps to become familiar with the Debug functions:

1. Press the HELP key. The Help window is displayed.

2. Press each function key corresponding to the functions displayed at the bottom of the screen. As you press each function key, a short explanation of the purpose of each function is displayed in the Help window.

3. Press RETURN. Exit HELP.

4. Press the ZMIN function key. The following message is displayed in the upper right hand corner of the screen:

   Enter compiler input file for EXAMPLE_PAS

5. Enter the source file name:

   EXAMPLE_PAS

   The EXAMPLE_PAS source listing is displayed in the Source window. Also, some new functions are displayed at the bottom of the screen.

6. Press the HELP key. The Help window is displayed again.

7. Press each function key corresponding to the new functions displayed at the bottom of the screen. As you press each function key, a short explanation of the purpose of each new function is displayed in the Help window.

8. Press RETURN. Exit HELP.

Setting Breaks

It is often helpful to suspend program execution when debugging a program. The device for suspending execution of a program is called a break. In this sample session, the following functions are used to illustrate setting breaks.

FWD

Scrolls forward to the next screen of text.

FIRST

Displays the first screen of the source listing. Because first is a lower priority function, it may not be assigned to a function key on terminals with only 16 keys. Instead, FIRST is entered on the home line.

LOCATE

Prompts you to type in text, then searches the source listing for matching text. If a match is found, the cursor is moved to the line containing the matching text.

SETBRK

Sets an execution break on the line containing the cursor. The line is highlighted to show that it contains a break. Execution is suspended before the line containing the break is executed. Execution resumes with the first statement on the line containing the break.

This section also uses the following item:

HOME

Moves the cursor to the home line. Line mode Debug commands can be entered on the home line for execution in screen mode Debug.

Perform the following steps to place two execution breaks in EXAMPLE_PAS:

1. Press the LOCATE function key. At the top right hand corner of the screen, you are prompted for the text to be located.

2. Enter the following text exactly as it appears in EXAMPLE_PAS:

   ```
   month_row + 1
   ```

   The cursor is moved to the line:

   ```
   month_row := month_row + 1;
   ```

3. Press the SETBRK function key. A break is set and the line containing the cursor is highlighted to show that it contains an execution break.

4. Use the down-arrow key to move the cursor to the procedure call:

   ```
   test2;
   ```

   If you do not see this line on your screen, press the FWD key. The next screen of the EXAMPLE_PAS source listing is displayed. Use the down-arrow key to position the cursor on the correct line.

5. Press the SETBRK function key. The line is highlighted to show that it contains an execution break.

6. Press the FIRST function key. The first screen of the EXAMPLE_PAS source listing is displayed in the source window.

   If FIRST is not assigned to a function key, FIRST must be entered on the home line. To do this, press the HOME key. The cursor moves to the home line. Enter the following screen mode Debug function on the home line:

   ```
   first
   ```

   The first screen of the EXAMPLE_PAS source listing is displayed in the Source window.

Debugging TEST1

Using Debug, you can execute a program one line or several lines at a time. Also, you can examine a variable's contents, change its contents, and execute code containing the variable several times. These capabilities are demonstrated in this sample session using the following functions:

CHAVAL

Prompts you to enter a variable name and the value you want it to contain, then changes the variable's contents to the new value.

GOTO

Moves the execution pointer to the line that contains the cursor. Execution resumes with the first statement on this line.

HSPEED

Executes a program until a break is encountered or the program ends.

SEEVAL

Prompts you to enter a variable name, then displays the value of the variable in the Output window.

STEP1

Executes a program one line at a time.

STEPN

Executes N lines of a program, where N is an integer.

Perform the following steps to demonstrate the use of the CHAVAL, GOTO, HSPEED, SEEVAL, STEP1, STEPN:

1. Press the STEP1 function key. The PROGRAM statement is executed; the execution arrow now points to the statement:

   ```
   BEGIN  {Main}
   ```

2. Press the STEP1 function key again. The BEGIN statement is executed, moving the execution arrow to the procedure call:

   ```
   test1;
   ```

3. Press the STEP1 function key. The TEST1 procedure call is executed; the arrow points to the first executable statement in TEST1.

4. Press the STEP1 function key seven times. An iteration of TEST1 is executed one line at a time. The output generated by the iteration is displayed in the Output window.

5. Press the SEEVAL function key. A prompt to enter a variable name is printed in the upper right hand corner of the screen. Enter the name:

   ```
   counter
   ```

   The value of COUNTER is displayed in the Output window:

   ```
   counter = 2
   ```

   Thus, you can use SEEVAL to observe the contents of a variable.

6. Press the CHAVAL function key. A prompt for a variable name and its new value is displayed in the upper right hand corner of the screen; enter:

   ```
   counter = 8
   ```

   The value of COUNTER is changed to 8.

7. Press the SEEVAL function key. When you are prompted for a variable name, enter:

   ```
   counter
   ```

   The following message is displayed in the Output window:

   ```
   counter = 8
   ```

   Thus, the change of COUNTER's value is verified.

8. Press the STEPN function key. In the upper right hand corner of the screen, you are prompted for the number of lines to execute; enter:

    ```
    5
    ```

    STEPN executes 5 lines of TEST1. The output generated by this loop iteration is displayed in the Output window.

9. Press the SEEVAL function key. When you are prompted for a variable name, enter:

    ```
    counter
    ```

    The value of COUNTER is displayed in the Output window:

    ```
    counter = 3
    ```

    Only the value of COUNTER passed to the SQUARING_PROCEDURE call was changed. The value of a FOR loop control variable cannot be changed once the loop has been entered. Therefore, the value of COUNTER used by this FOR loop remains unchanged.

10. Use the up-arrow key to move the cursor to the line:

    ```
    BEGIN  {test1}
    ```

11. Press the GOTO function key. The execution arrow moves to the line containing the cursor; execution resumes at this line.

12. Press the HSPEED function key. Execution resumes from the BEGIN statement. Since the FOR loop is executed anew, COUNTER is initialized to 1. Execution of EXAMPLE_PAS continues until an execution break is encountered.

Debugging TEST2

After program execution is resumed in step 12 of TEST1, execution stops at the break set on the TEST2 procedure call. The following functions are used in TEST2 to illustrate more Debug capabilities:

BKW

Scrolls backward to the previous screen of text.

CHAVAL

Prompts you to enter a variable name and the value you want it to contain, then changes the variable's contents to the new value.

DELBRK

Deletes execution breaks.

FWD

Scrolls forward to the next screen of text.

HSPEED

Executes a program until a break is encountered or the program ends.

MSPEED

Executes a program one procedure at a time.

SEEVAL

Prompts you to enter a variable name, then displays the value of the variable in the Output window.

This section also uses the following items:

HOME

Moves the cursor to the home line. Line mode Debug commands can be entered on the home line for execution in screen mode Debug.

DISPLAY_PROGRAM_VALUE

A line mode Debug command that displays the values of program variables.

Perform the following steps to learn how to execute loops one iteration at a time, execute line mode Debug commands, and scroll output data when using Debug:

1. Press the HSPEED function key. Execution stops at the break set on the last line of the WHILE loop in TEST2; output generated by the loop is displayed in the Output window.

2. Press the HSPEED function key again. One iteration of the WHILE loop is executed; execution stops at the break set in the WHILE loop again. Each time HSPEED is used, an iteration of the loop is performed. By using strategically placed execution breaks, as in this example, a loop can be executed one iteration at a time.

3. Press the HSPEED function key. One more loop iteration is performed.

4. Press the SEEVAL function key. When you are prompted, enter:

   ```
   month_row
   ```

   The following message is displayed in the Output window:

   ```
   month_row = 2
   ```

5. Press the CHAVAL function key. When you are prompted, enter:

   ```
   month_row = 4
   ```

6. Press the SEEVAL function key. When you are prompted, enter:

   ```
   month_row
   ```

   The following message is displayed in the Output window:

   ```
   month_row = 4
   ```

   Thus, the change to MONTH_ROW is verified.

7. Press the HSPEED function key. One iteration of the WHILE loop is executed.

8. Press the SEEVAL function key. When you are prompted, enter:

   ```
   month_row
   ```

   the following message is then displayed in the Output window:

   ```
   month_row = 5
   ```

   The value given to MONTH_ROW in step 5 is used by the WHILE loop.

9. Press the HOME key. The cursor moves to the home line.

10. Enter the line mode Debug command:

    ```
    display_program_value name = $all
    ```

    The values of all variables declared in EXAMPLE_PAS are displayed in the Output window. Thus, line mode Debug commands can be used in screen mode Debug by entering them on the home line. For more information about using line mode Debug commands see the Debug Usage Manual.

11. Press the DELBRK key. The execution break is deleted. The highlight is removed from the line when the break is removed.

12. Press the down-arrow key until the cursor is inside of the Output window.

13. Press the BKW key. The data in the Output window scrolls backward. When the cursor is contained within the Output window, you can use the BKW and FWD keys to scroll backward and forward through the data in the window .

14. Press the MSPEED function key. The execution of EXAMPLE_PAS resumes, stopping when TEST2 is exited. The execution arrow points to the beginning of TEST3.

Debugging TEST3

After resuming execution of EXAMPLE_PAS in step 14 of section TEST2, execution stops at the begining of TEST3. In TEST3, Debug is presented with an execution error. The following functions are used in this sample session to demonstrate how Debug can be used when an execution error is encountered:

CHAVAL

Prompts you to enter a variable name and the value you want it to contain, then changes the variable's contents to the new value.

GOTO

Moves the execution pointer to the line that contains the cursor. Execution resumes with the first statement on this line.

SEEVAL

Prompts you to enter a variable name, then displays the value of the variable in the Output window.

STEP1

Executes a program one line at a time.

QUIT

Used to leave Debug.

Perform the following steps to finish the example:

1. Press the STEP1 function key two times. The first two statements of TEST3 are executed and the execution arrow points to the next line to be executed:

   ```
   quotient = dividend / divisor;
   ```

2. Press the STEP1 function key again. Execution halts, and the following message flashes in the top right hand corner of the screen:

   ```
   divide_fault
   ```

3. Press the SEEVAL function key. When you are prompted for a variable name, enter:

   divisor

   The following message is displayed in the Output window:

   divisor = 0

   A division by zero caused the execution error.

4. Press the CHAVAL function key. When you are prompted, enter:

   divisor = 5.0

   The value of divisor is changed to 5.

5. Press the SEEVAL function key. When you are prompted, enter:

   divisor

   The following text is displayed in the Output window:

   divisor = 5.00000E+0000

   The change to DIVISOR is verified.

6. Press the GOTO function key. The execution arrow points to the DIVISION statement, so program execution resumes with this statement.

7. Press the STEP1 function key two times. The DIVISION statement is executed. Therefore, the GOTO and CHAVAL functions can be used in concert to recover from execution errors. However, to correct execution errors permanently, you must exit Debug, edit the program, and recompile it.

8. Press the STEP1 function key again. The result of the DIVISION statement is displayed in the Output window.

9. Press the STEP1 function key two more times. EXAMPLE_PAS ends and the following message is displayed in the Output window:

   Debug: The status at termination was: Normal.

10. Press the QUIT function key. Exit Debug.

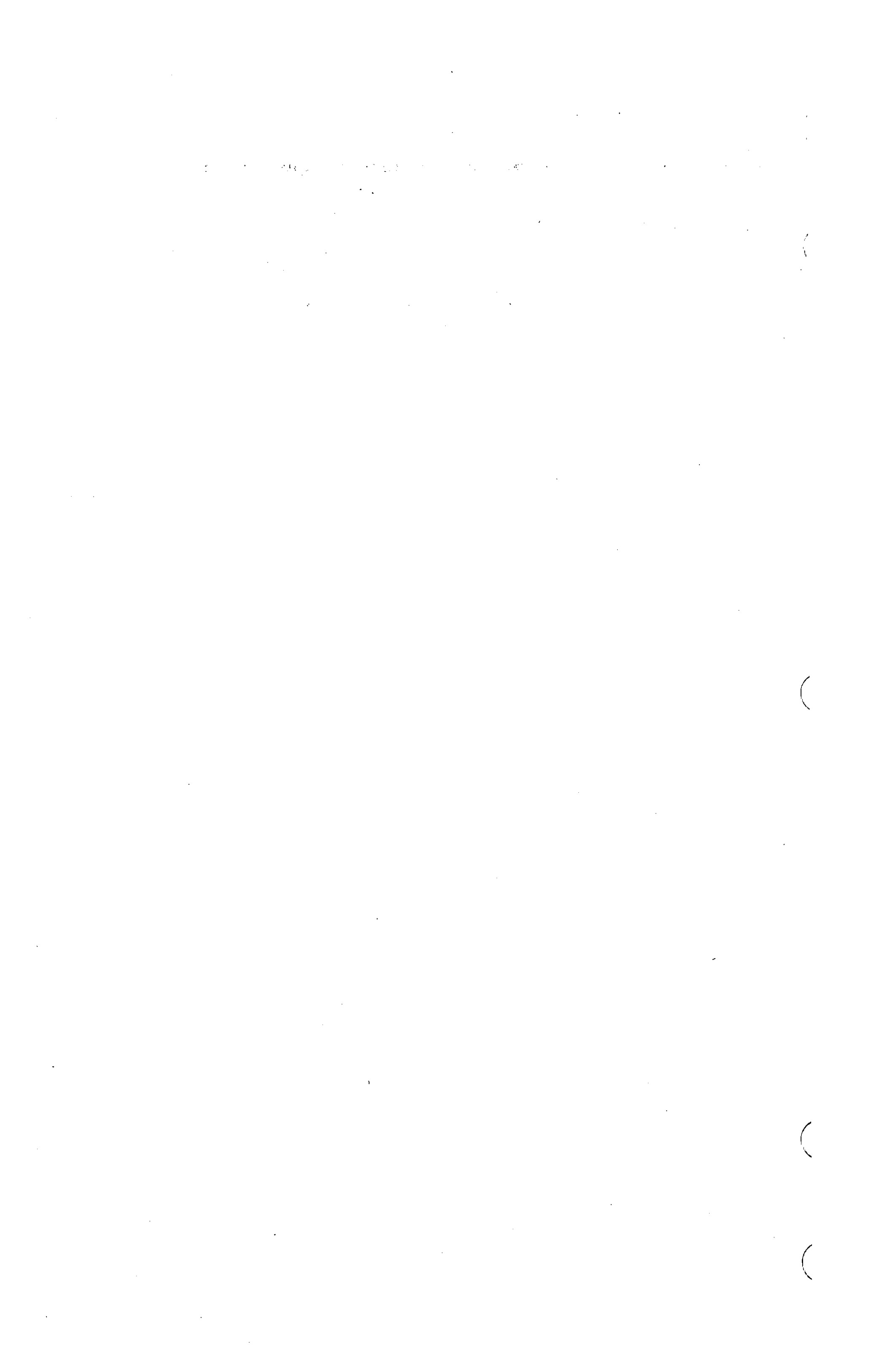
Now that you have concluded this example, you should be able to begin using screen mode Debug to debug your Pascal programs.

Appendixes

A - Glossary A-1

B - Related Manuals B-1

C - ASCII Character Set C-1



Glossary A

A

Active Call Chain

List of calls that led to the current procedure.

B

BASIC

An elementary programming language whose name is an acronym for Beginner's All-Purpose Symbolic Instruction Code.

Batch Debugging

Debugging when the user has no direct control of debugging during program execution. Contrast with Interactive Debugging.

Batch Mode

An execution mode in which a job is submitted and processed as a unit with no intervention from the user. Compare with Interactive Mode.

Beginning-of-Information (BOI)

The file boundary that marks the beginning of the file.

Binary Object Program

An executable machine language program that is produced from a source.

Binding

The process of combining modules to form a new load module.

BOI

See Beginning-of-Information

Bound Module

A load module formed by binding other modules.

Break

The primary mechanism for Debug to gain control from an executing program. A break specifies an event and an address range such that when the event occurs within the address range, Debug takes control.

C

C

A high-level language for programming a computer.

COBOL

COmmon Business Oriented Language. A business language for programming a computer.

Compile

To translate a program written in a high-level language into a machine language program that can be loaded and executed.

Condition Handler

A procedure called when an exception condition occurs. Condition handler processing occurs after Debug processing if Debug mode is on. The procedure is called only if it has been established as the condition handler for the condition type and the condition occurs within its scope.

Current Line

The line on which the cursor is positioned. If the cursor is on the subcommand line, the current line is the line on which the cursor was positioned when you press HOME.

Cursor

The pointer used by your terminal to indicate where you are positioned in the file.

CYBIL

CYBer Implementation Language. The implementation language of NOS/VE.

D

Debug

The NOS/VE command utility for tracing and correcting program errors.

Debug Mode

A program attribute that can be set in a program description or with the SCL command SET_PROGRAM_ATTRIBUTE.

Debug Session

The sequence of interactions that take place between the user and the Debug utility.

Debug Utility

A utility that provides source-code-level symbolic debugging for programs.

Default

The assumed value for a parameter when the parameter is not specified by the user.

E

End-of-Information (EOI)

The point at which the data in a file ends.

Entry Point

A location within a program unit or module that can be branched to from other program units or modules. Each entry point has a unique name.

EOI

See End-of-Information.

Exception Handler

A situation that, when detected by a procedure caller, indicates an abnormal completion of the called procedure.

Execution Ring

The level of hardware privilege assigned to a procedure while it is executing.

Execution Time

The time at which a compiled source program is executed. Also known as Run Time.

F

File

A collection of information referenced by name. A file is an autonomous collection of information that exists separately from the programs that read or write the file.

File Attribute

One of a set of characteristics of a file. The file attribute set defines the file structure and processing limitations.

File Connection

A condition established between one file called the subject file and one or more other files, each of which is called a target file. The file connection causes data access requests made on the subject file to be redirected to one or more target files.

FORTRAN

FORmula TRANslating System. An algebraic and logical language for programming a computer.

Function

A procedure that returns a value to the place in an expression where the procedure was called.

Function Key

A key on the terminal that, when pressed, performs a specified operation. The operation can be either defined by the software or built into the terminal.

Function Key Prompts

Labels displayed on your screen which describe the function of a programmable function key prompt.

I

Identifier

A name that denotes a quantity. An identifier can be a name that denotes a constant, type, variable, value, procedure, or function.

Interactive Debugging

Debugging when the user has direct control of the debugging process. Contrast with Batch Debugging.

Interactive Mode

A mode of execution where the user enters commands or data at the terminal during program execution. Contrast with Batch Mode.

K

Keyword

A parameter value that has special meaning in the context of a particular parameter.

L

Line Mode Debugging

Debugging by entering commands in response to prompts. Contrast with Screen Mode Debugging.

M

Machine Addressing

Use of actual machine addresses. Contrast with Symbolic Addressing.

Machine-Level Debugging

Debugging using machine-level terms such as machine addresses. A knowledge of machine architecture is required. Contrast with Symbolic Debugging.

Module

A unit of code. An object module is the unit of object code corresponding to a compilation unit. A load module is a unit of object code stored in an object library. When using Debug, module refers to a program, program unit, subroutine, procedure, module, block data, or function.

N

Name Call

A method of loading and executing a program in which you enter the name of the file containing the object program.

NOS/VE

Acronym for Network Operating System/Virtual Environment, an operating system for the host computer.

O

Object Code

The code that the compiler produces from your source code. Also called relocatable binary code.

Optimization

The manipulation of object code to reduce execution time.

P

Parameter

An item of information that is passed to or from a procedure or function.

Pascal

A high-level language for programming a computer.

Procedure

When using Debug, procedure refers to a program, function, block within a function, subroutine, procedure, block data, or statement function.

Process Virtual Address (PVA)

The virtual address known locally by a program (or process). It is converted to a system virtual address (SVA) that is known globally by the system. It consists of a ring number, a segment number, and a byte number. The segment number is used to form the active segment identifier in the system virtual address.

Q

Qualifier

A word used to uniquely reference a user-defined word or a special register. A qualifier can be a data-name, file-name, section-name, library-name, or report-name.

R

Recursion

The process of invoking a function from within that function. Recursion is closely related to mathematical induction.

Ring

Level of hardware protection given a file or segment. A file is protected from unauthorized access by tasks executing in higher rings. See also Execution Ring.

Ring Attribute

A file attribute whose value consists of three ring numbers, referred to as r1, r2, and r3. The ring numbers define the four ring brackets for the file as follows:

Read bracket is 1 through r2
Write bracket is 1 through r1
Execute bracket is r1 through r2
Call bracket is r2+1 through r3.

Run Time

The time at which a compiled source program is executed. Also known as Execution Time.

S

SCL Variable

The means of storing a value to be tested or displayed by an SCL statement.

Screen Mode Debugging

Debugging using a full-screen interface. Contrast with Line Mode Debugging.

Source Listing

A compiler-produced listing of the user's original source program.

Source Program

A program written in a high-level language such as BASIC, COBOL, C, FORTRAN, or Pascal.

Standard File

A file that provides a default file for use by job files and other files. The standard files are identified by the following names:

$COMMAND
$COMMAND_OF_CALLER
$ECHO
$ERRORS
$INPUT
$LIST
$OUTPUT
$RESPONSE

Status Variable

The variable in which the completion status of the command or procedure is returned.

Symbolic Addressing

Use of address in source program terms such as program names and line numbers. Contrast with Machine Addressing.

Symbolic Debugging

Debugging using source program terms such as line numbers and program names. Contrast with Machine-Level Debugging.

Syntax

Rules defining whether a statement is well formed.

T

Task

A task is a process within the job that executes independently of other tasks using its own address space. A task has the ability to access all local files attached to the job.

Terminal Definition File

The source file used in defining a terminal for use with a full-screen application.

Traceback

A list of procedure names within a program, beginning with the currently executing procedure, proceeding backward through the sequence of called procedures, and ending with the main program.

U

Utility

A NOS/VE processor consisting of routines that perform a specific operation.

V

Value

An expression or application value specified in a parameter list. Each value must match the defined kind of value for the parameter. Keywords, constants, and variable references are all values.

Variable

A named memory location that is allowed to store different values at different times during program execution.

Variable Name

A name that identifies a variable.

Related Manuals B

Table B-1 lists all manuals that are referenced in this manual or that contain background information. A complete list of NOS/VE manuals is given in the SCL Language Definition manual. If your site has installed the online manuals, you can find an abstract for each NOS/VE manual in the online System Information manual. To access this manual, enter:

```
explain
```

Ordering Printed Manuals

You can order Control Data manuals through Control Data sales offices or through:

Control Data Corporation
Literature and Distribution Services
308 North Dale Street
St. Paul, Minnesota 55103

Accessing Online Manuals

To access an online manual, log in to NOS/VE and specify the online manual title (listed in table B-1) on the EXPLAIN command. For example, to read the Debug online manual, enter:

```
explain manual=debug
```

Related Manuals

Table B-1. Related Manuals

Manual Title	Publication Number	Online Title
BASIC for NOS/VE Usage	60486313	BASIC
C/VE Reference Manual	60469830	C
COBOL for NOS/VE Usage	60486013	COBOL
CYBIL Language Definition Usage	60464113	
FORTRAN Version 1 for NOS/VE Language Definition Usage	60485913	
FORTRAN Version 1 for NOS/VE Quick Reference	L60485918	FORTRAN
FORTRAN Version 2 for NOS/VE Language Definition Usage	60487113	
FORTRAN Version 2 for NOS/VE Quick Reference	L60487118	VFORTRAN
Pascal for NOS/VE Usage	60485613	PASCAL
SCL for NOS/VE Object Code Management Usage	60464413	
SCL for NOS/VE Source Code Management Usage	60464313	
Terminal Definition for NOS/VE Usage	60464016	
Virtual State Hardware Reference Manual	60469680	

ASCII Character Set C

This appendix lists the ASCII character set (table C-1).

NOS/VE supports the American National Standards Institute (ANSI) standard ASCII character set (ANSI X3.4-1977). NOS/VE represents each 7-bit ASCII code in an 8-bit byte. The 7 bits are right-justified in each byte. For ASCII characters, the leftmost bit is always zero.

In addition to the 128 ASCII characters, NOS/VE allows use of the leftmost bit in an 8-bit byte for 256 characters. The use and interpretation of the additional 128 characters is user-defined.

Table C-1. ASCII Character Set

Decimal	ASCII Code (Hexadecimal)	Octal	Graphic or Mnemonic	Name or Meaning
000	00	000	NULL	Null
001	01	001	SOH	Start of heading
002	02	002	STX	Start of text
003	03	003	ETX	End of text
004	04	004	EOT	End of transmission
005	05	005	ENQ	Enquiry
006	06	006	ACK	Acknowledge
007	07	007	BEL	Bell
008	08	010	BS	Backspace
009	09	011	HT	Horizontal tabulation
010	0A	012	LF	Line feed
011	0B	013	VT	Vertical tabulation
012	0C	014	FF	Form feed
013	0D	015	CR	Carriage return
014	0E	016	SO	Shift out
015	0F	017	SI	Shift in
016	10	020	DLE	Data link escape
017	11	021	DC1	Device control 1
018	12	022	DC2	Device control 2
019	13	023	DC3	Device control 3
020	14	024	DC4	Device control 4
021	15	025	NAK	Negative acknowledge
022	16	026	SYN	Synchronous idle
023	17	027	ETB	End of transmission block
024	18	030	CAN	Cancel
025	19	031	EM	End of medium
026	1A	032	SUB	Substitute
027	1B	033	ESC	Escape
028	1C	034	FS	File separator
029	1D	035	GS	Group separator
030	1E	036	RS	Record separator
031	1F	037	US	Unit separator
032	20	040	SP	Space
033	21	041	!	Exclamation point
034	22	042	"	Quotation marks
035	23	043	#	Number sign

(Continued)

Table C-1. ASCII Character Set (Continued)

Decimal	ASCII Code (Hexadecimal)	Octal	Graphic or Mnemonic	Name or Meaning
036	24	044	$	Dollar sign
037	25	045	%	Percent sign
038	26	046	&	Ampersand
039	27	047	'	Apostrophe
040	28	050	(	Opening parenthesis
041	29	051	)	Closing parenthesis
042	2A	052	*	Asterisk
043	2B	053	+	Plus
044	2C	054	,	Comma
045	2D	055	-	Hyphen
046	2E	056	.	Period
047	2F	057	/	Slant
048	30	060	0	Zero
049	31	061	1	One
050	32	062	2	Two
051	33	063	3	Three
052	34	064	4	Four
053	35	065	5	Five
054	36	066	6	Six
055	37	067	7	Seven
056	38	070	8	Eight
057	39	071	9	Nine
058	3A	072	:	Colon
059	3B	073	;	Semicolon
060	3C	074	<	Less than
061	3D	075	=	Equal to
062	3E	076	>	Greater than
063	3F	077	?	Question mark
064	40	100	@	Commercial at
065	41	101	A	Uppercase A
066	42	102	B	Uppercase B
067	43	103	C	Uppercase C
068	44	104	D	Uppercase D
069	45	105	E	Uppercase E
070	46	106	F	Uppercase F
071	47	107	G	Uppercase G

(Continued)

Table C-1. ASCII Character Set (Continued)

Decimal	ASCII Code (Hexadecimal)	Octal	Graphic or Mnemonic	Name or Meaning
072	48	110	H	Uppercase H
073	49	111	I	Uppercase I
074	4A	112	J	Uppercase J
075	4B	113	K	Uppercase K
076	4C	114	L	Uppercase L
077	4D	115	M	Uppercase M
078	4E	116	N	Uppercase N
079	4F	117	O	Uppercase O
080	50	120	P	Uppercase P
081	51	121	Q	Uppercase Q
082	52	122	R	Uppercase R
083	53	123	S	Uppercase S
084	54	124	T	Uppercase T
085	55	125	U	Uppercase U
086	56	126	V	Uppercase V
087	57	127	W	Uppercase W
088	58	130	X	Uppercase X
089	59	131	Y	Uppercase Y
090	5A	132	Z	Uppercase Z
091	5B	133	[	Opening bracket
092	5C	134	\	Reverse slant
093	5D	135	]	Closing bracket
094	5E	136	^	Circumflex
095	5F	137	_	Underline
096	60	140	`	Grave accent
097	61	141	a	Lowercase a
098	62	142	b	Lowercase b
099	63	143	c	Lowercase c
100	64	144	d	Lowercase d
101	65	145	e	Lowercase e
102	66	146	f	Lowercase f
103	67	147	g	Lowercase g

(Continued)

Table C-1. ASCII Character Set (Continued)

Decimal	ASCII Code (Hexadecimal)	Octal	Graphic or Mnemonic	Name or Meaning
104	68	150	h	Lowercase h
105	69	151	i	Lowercase i
106	6A	152	j	Lowercase j
107	6B	153	k	Lowercase k
108	6C	154	l	Lowercase l
109	6D	155	m	Lowercase m
110	6E	156	n	Lowercase n
111	6F	157	o	Lowercase o
112	70	160	p	Lowercase p
113	71	161	q	Lowercase q
114	72	162	r	Lowercase r
115	73	163	s	Lowercase s
116	74	164	t	Lowercase t
117	75	165	u	Lowercase u
118	76	166	v	Lowercase v
119	77	167	w	Lowercase w
120	78	170	x	Lowercase x
121	79	171	y	Lowercase y
122	7A	172	z	Lowercase z
123	7B	173	{	Opening brace
124	7C	174	|	Vertical line
125	7D	175	}	Closing brace
126	7E	176	~	Tilde
127	7F	177	DEL	Delete

Index

A

Abort File 3-21
ABORT_FILE Parameter 3-21
About this manual 5
Accessing online manuals B-1
ACTIVATE_SCREEN Command 3-5; 5-4
Active call chain A-1
ACTS Command 3-5; 5-4
Address
    Bound modules 6-4
    Changing contents 5-11
    Displaying contents 5-39
    Machine 1-1; 2-6; A-6
    Module/procedure offset 6-5
    Module/section offset 6-5
    Ranges 3-7
    Referenced 6-1, 3
    Reported 6-1
    Symbolic A-9
    Tables 6-1
Aligning the window 4-38
Arithmetic
    Overflow break 3-11
    Significance is lost break 3-11
ASCII character set C-1
ATTENTION_CHARACTER parameter 2-5
Audience, manual 5

B

BACK function 4-34
BASIC
    Debug example 7-1
    Description A-1
Batch job debugging 1-6; A-1
Batch level debugging 1-6
Batch mode debugging 1-6; A-1
Beginning
    A Debug session 3-1
    Execution 3-10; 4-22; 5-66
Beginning-of-information A-1
Binary Object Program A-1
Binding
    Description A-1
    Modules 6-4
BKW function 4-36
BOI A-1

Bound Modules
    Addressing 6-4
    Description A-1
Breaks
    Address ranges 3-7
    Arithmetic overflow 3-11
    Arithmetic significance is lost 3-11
    Defaults 3-11
    Deferred 6-10
    Definition 3-7; A-2
    Deleting 3-7; 4-21
    Displaying 3-7
    Divide fault 3-12
    Exponent overflow 3-12
    Exponent underflow 3-12
    Floating-point indefinite 3-12
    Floating-point significance is lost 3-12
    Invalid business data processing data 3-12
    Multiple 6-11
    Pause 2-4; 6-6
    Setting 3-7; 4-19; 5-67
    Terminate 2-4; 6-6

C

C
    Debug example 7-14
    Description A-2
Call chain, tracing 5-33
Carriage return key 4-35
CHAD command 3-19, 20; 5-6
CHAFA command 3-5
CHAIS command 3-2
CHAM command 5-11
CHANGE_DEFAULTS command 3-19, 20; 5-6
CHANGE_FILE_ATTRIBUTE command 3-5
CHANGE_INTERACTION_STYLE command 3-2
CHANGE_MEMORY command 5-11
CHANGE_PROGRAM_VALUE command 3-13; 5-15
CHANGE_REGISTER command 5-24
CHANGE_TERMINAL_ATTRIBUTES command 2-3

Changing
Defaults 5-6
Displays 4-16
Execution location 3-10, 11; 4-27
Memory contents 5-11
Register values 5-24
Screen options 4-39
The Debug input file 3-17
The Debug output file 3-17, 20
The P register 3-10, 11
To line mode 3-5
To screen mode 3-5; 5-4
Variable values 1-1; 3-13; 4-30, 32; 5-15
Window displays 4-16
CHAPV command 3-13; 5-15
CHAR command 5-24
Character set C-1
CHATA command 2-3
CHAVAL function 3-13; 4-32
$CL function 5-92
Clearing a screen 4-38
$CM function 5-93
COBOL
Debug example 7-26
Description A-2
Command file 3-17, 21
Commands
In line mode 3-6; 5-1
In screen mode 3-6; 4-5
Communication network
CDCNET 2-5
NAM/CCP 2-5
Compiling
A BASIC program 2-6
A C program 2-6
A COBOL program 2-6
A CYBIL program 2-6
A FORTRAN Version 1 program 2-6
A FORTRAN Version 2 program 2-6
A Pascal program 2-6
A subprogram 2-6
DEBUG_AIDS parameter 2-6
Definition A-2
For symbolic debugging 2-6
-g option 2-6
OPTIMIZATION_LEVEL parameter 2-7
-R option 2-7

Component modules 6-4
Comprehensive debugging 6-1
Compress characters 4-39
Condition handlers 3-8; 6-9; A-2
Contents, table of 3
Conventions, manual 6
$CP function 5-94
$CPVA function 5-95
CREATE_FILE_CONNECTION command 3-20, 26
CREATE_OBJECT_LIBRARY command 6-4
CREATE_PROGRAM_DESCRIPTION command 3-1, 2, 4
CREFC command 3-20, 26
CREOL command 6-4
CREPD command 3-1, 2, 4
$CURRENT_MODULE function 5-93
Current line
    Description A-2
    Number 5-92
$CURRENT_LINE function 5-92
$CURRENT_PROCEDURE function 5-94
$CURRENT_PVA function 5-95
Cursor
    Description A-2
    Moving 4-35
CYBIL
    Debug example 7-40
    Description A-2
    Runtime error 6-8

D

DB/ prompt 1-5; 3-4
DEAS function 2-1; 3-5; 4-41
Debug
    Beginning a session 3-1
    Command file 3-17, 21
    Commands 3-6; 5-4; 5-1
    Control 3-1
    Ending a session 3-16; 4-41; 5-65
    Examples 7-1
    Features 1-1
    Functions, line mode 5-91
    Functions, screen mode 4-1, 4, 14
    Input file 3-17
    Introduction 1-1
    Line mode 1-5; 5-1
    Mode 3-1, 2, 4; A-3
    Optimization 6-7
    Output 3-17, 20
    Output file 3-17, 20
    Performance 6-7
    Procedure A-7
    Prompt 1-5, 3-4
    Rings 6-10
    Screen mode 1-3; 4-1
    Session 1-2; 3-4; A-3
    Tables 2-6
    Utility 1-1; A-3
DEBUG_AIDS parameter 2-6
Debugging
    A BASIC program 7-1
    A C program 7-14
    A COBOL program 7-26
    A CYBIL program 7-40
    A CYBIL runtime error 6-8
    A FORTRAN Version 1 program 7-55
    A FORTRAN Version 2 program 7-55
    A Pascal program 7-68
    A terminated program 6-8
    Batch job 1-6
    Batch level 1-6
    Batch mode 1-6
    Comprehensive 6-1
    Condition handlers 6-8
    Environment 1-1; 5-36
    Getting started 2-1
    Interactive 1-3; A-5
    Interrupt processing 6-6
    Line mode 1-5; 3-4; 5-1
    Machine-level 2-6; A-6
    Multi-task 6-11
    Optimized code 6-7
    Screen mode 1-3; 3-2; 4-1
    Symbolic 1-1; 2-6
    Using function keys 2-3; 4-4

DEBUG_INPUT parameter 1-6; 3-17
DEBUG_MODE parameter 3-1
DEBUG_OUTPUT parameter 3-17, 20
Default
    Breaks 3-11
    Changing 5-6
    Description A-3
Deferred breaks 6-10
DELB command 3-7; 5-28
DELBRK function 3-7; 4-21
DELETE_BREAK command 3-7; 5-28
Deleting breaks 3-7; 4-21; 5-28
DISB command 3-7; 5-30
DISC command 5-33
DISCI command 3-14, 15
DISDE command 5-36
DISFA command 3-5
DISFI command 3-14, 15
DISM command 5-39
DISPLAY_BREAK command 3-7; 5-30
DISPLAY_CALL command 5-33
DISPLAY_COMMAND_INFORMATION command 3-14, 15
DISPLAY_DEBUGGING_ENVIRONMENT command 5-36
DISPLAY_FILE_ATTRIBUTE command 3-5
DISPLAY_FUNCTION_INFORMATION command 3-14, 15
Displaying
    Breaks 3-7; 5-30
    Debugging environment 5-36
    Memory contents 5-39
    Registers 5-58, 100
    Source code 4-16
    Stack frame 5-61
    Variable values 3-13; 4-30; 5-46, 97
DISPLAY_MEMORY command 5-39
DISPLAY_PROGRAM_VALUE command 3-13; 5-46
Displays
    Changing 4-16
    Trace 4-10, 18
    Zoom-in 3-2; 4-6, 16
    Zoom-out 3-2; 4-8, 17
DISPLAY_REGISTER command 5-58
DISPLAY_STACK_FRAME command 5-61
DISPLAY_VALUE Command 1-2; 3-20
DISPV command 3-13; 5-46
DISR command 5-58
DISSF command 5-61
DISV command 1-2; 3-20
Divide fault break 3-12
Dividing windows 4-38
DOWN function 4-35

E

Echo Debug commands 1-2
End-of-information A-3
Ending a Debug session 3-16; 4-41; 5-65
Enlarge compressed characters 4-39
Entry point A-3
Environment
  Debugging 1-1; 5-36
  Multi-ring 6-11
  Multi-task 6-11
  Programming 3-2
EOI A-3
Error
  Runtime 3-8, 21
  Execution 3-8, 21
Examples
  Debugging a BASIC program 7-1
  Debugging a C program 7-14
  Debugging a COBOL program 7-26
  Debugging a CYBIL program 7-40
  Debugging a FORTRAN Version 1 program 7-55
  Debugging a FORTRAN Version 2 program 7-55
  Debugging a Pascal program 7-68
Exception handler A-4
EXECUTE_TASK command 3-1, 2, 4
Execution
  Beginning 3-10; 4-22; 5-66
  Changing location of 3-10; 4-27
  Error 3-8, 21
  Interruption 4-19; 5-67; 6-6
  Resuming 3-10; 4-22; 5-66
  Ring A-4
  Suspending 3-7; 4-19; 5-67
  Time A-4
  Under Debug control 3-1
EXET command 3-1, 2, 4
EXPLAIN command 3-14
Exponent
  Overflow break 3-12
  Underflow break 3-12

F

File
    Abort 3-21
    Attribute 3-5; A-4
    Connections 1-2; 3-26; A-4
    Debug input 3-17
    Debug output 3-17, 20
    Definition A-4
    $LOCAL.COMMAND 3-17
    Of Debug commands 1-1; 3-17, 21
    $OUTPUT 3-20
    Standard A-9
    $SYSTEM.TDU.TERMINAL_DEFINITIONS 2-3
FILE_PROCESSOR attribute 3-5
FIRST function 4-34
Floating-point indefinite break 3-12
Floating-point significance is lost break 3-12
FORTRAN Version 1
    Debug example 7-55
    Description A-4
FORTRAN Version 2
    Debug example 7-55
    Description A-4
Full-screen
    Characteristics 2-2
    Definition 2-3
    Interface 2-2
    Layout 4-2
Function
    Descriptions 1-3; 2-3; 4-4, 14, 18; A-5
    Keys 1-3; 2-3; 3-6; 4-2, 4; A-5
    Line mode 5-91
Function key prompts 4-2, 4; A-5
Functions 1-3; 2-3; 4-18; A-5
FWD function 4-36

G

-g option 2-6
GOTO function 1-4; 3-10; 4-27

H

Handlers, condition 3-8; 6-9
HELP
    Command 3-14
    Function 3-14; 4-10, 16
    Information 3-14; 4-10, 16
    Online 1-3; 3-14; 4-10, 16
    Window 3-14; 4-10, 16
History, manual 2
Home
    Key 4-35
    Line 1-3; 2-3; 3-6; 4-5, 35
HSPEED function 3-10; 4-22

I

Identifier A-5
In case of trouble 7
INCLUDE_FILE command 1-2; 3-19
Input file 3-17
Interactive
    Debug session 1-3
    Debugging A-5
    Mode A-5
    Terminal 2-1
Interrupt
    Processing 6-6
    Pause break 6-6
    Terminate break 6-6
    Nearly exhausted resource 6-6
Invalid business data processing data break 3-12

K

KEYS
    Function 4-12, 18
    Window 4-12, 18
Keyword A-5

L

LAST function 4-34
LEFT function 4-38
Line Mode Debug
    Capabilities 1-5
    Commands 5-1
    Definition A-5
    Ending a session 3-16; 5-65
    Example 1-5
    Prompt 1-5; 3-4
    Session 3-4
    Starting a session 3-4
    Switching to 3-5; 4-40
    Terminal set up 2-1
Load module 6-4
$LOCAL.COMMAND 3-17
LOCATE function 4-36
Locating text 4-36
LSPEED function 4-25

M

Machine-level
    Addresses 1-1; 2-6
    Debugging 2-6; A-6
    Process Virtual Address [PVA] 2-6
Manual
    Audience 5
    Conventions 5
    History 2
    Online 3-14; B-1
    Ordering B-1
    Organization 5
    Related B-1
$MEM function 5-96
Memory
    Changing 5-11
    Contents 5-96
    Displaying 5-39
$MEMORY function 5-96
Modules
    Addressing 6-4
    Binding 6-4
    Block referencing 6-5
    Bound 6-4
    Component 6-4
    Description A-6
    Display 4-17
    Procedure offset addressing 6-5
    Section offset addressing 6-5
Moving
    Cursor 4-35
    Pages 4-36
    Screens 4-36

MSPEED function 3-10; 4-25
Multiple breaks 6-11
Multi-ring environment 6-11
Multi-task debugging 6-11

N

Name call 3-4; A-6
NARROW function 4-39
NEXT key 4-35
NOS/VE
    Description A-6
    Dual State With 2550's 2-5
    Dual State With CDCNET 2-5
    Standalone 2-5
$NULL 3-21

O

Object code
    Description A-6
    Optimization 2-7; 6-7; A-6
Online
    HELP 1-3; 3-14
    Manual 3-14; B-1
OPTIMIZATION_LEVEL parameter 2-7
Optimizing
    A BASIC program 2-7
    A C program 2-7
    A COBOL program 2-7
    A CYBIL program 2-7
    A FORTRAN Version 1 program 2-7
    A FORTRAN Version 2 program 2-7
    A Pascal program 2-7
    Description A-6
    Level for use with Debug 2-7; 6-7
    Object code 2-7; 6-7
    OPTIMIZATION_LEVEL parameter 2-7
    Performance 6-7
    -R option 2-7
Options window 4-13, 40
OPTS function 4-13, 40
Ordering printed manuals B-1
Organization, manual 5
Output
    file 3-17, 20
    window 4-10

P

P Register 3-11
Pages, moving 4-36
Parameter A-7
Pascal
    Debug example 7-68
    Description A-7
Pause break 2-4; 6-6
Performance, Debug 6-7
PPE (See Professional Programming Environment)
Procedure
    Debug A-7
    SCL 1-1
Process Virtual Address [PVA]
    Description 2-6; A-7
    Value 5-95
Professional Programming Environment 3-2
Program
    Compilation 2-6
    Error 3-7, 8, 21
    Execution 3-10; 5-66
    Terminated 6-8
$PROGRAM_VALUE function 5-97
Programming Environment 3-2
Prompt, Debug 1-5, 3-4
$PV function 5-97
PVA
    Description 2-6; A-7
    Value 5-95

Q

Qualifier A-7
QUI command 3-16; 5-65
QUIT
    Command 3-16; 5-65
    Function 3-16; 4-41

R

-R option 2-7
Recursion A-8
Referenced addresses 6-1, 3
Refreshing the screen 4-38
REFRSH function 4-38
$REG function 5-100
Register
    Changing 5-24
    Displaying 5-58, 100
    P 3-11

$REGISTER function 5-100
Related manuals B-1
Reported addresses 6-1
RESC command 2-4
Restoring a Source window 4-34
RESUME_COMMAND 2-4
Resuming execution 3-10; 4-22; 5-66
RETURN key 4-35
RIGHT function 4-39
Ring
    Attribute A-8
    Debug 6-10
    Deferred breaks 6-10
    Description A-8
    Multiple breaks 6-11
    Multi-ring environment 6-11
RUN command 3-10, 11; 5-66
Runtime error 3-7, 8, 21; A-8

S

SCL
    CHANGE_FILE_ATTRIBUTE command 3-5
    CHANGE_INTERACTION_STYLE command 3-2
    CHANGE_TERMINAL_ATTRIBUTES command 2-3
    CREATE_FILE_CONNECTION command 3-20, 26
    CREATE_OBJECT_LIBRARY command 6-4
    CREATE_PROGRAM_DESCRIPTION command 3-1, 2, 4
    DISPLAY_COMMAND_PARAMETER command 3-14, 15
    DISPLAY_VALUE command 3-20
    EXECUTE_TASK command 3-1, 2, 4
    EXPLAIN command 3-14
    Features 1-2
    INCLUDE_FILE command 3-19
    SET_DEBUG_RING command 6-10
    SET_PROGRAM_ATTRIBUTES command 3-1, 2, 4
    SET_TERMINAL_ATTRIBUTES command 2-3
    Variable A-8
Screen
    Clearing 4-38
    Moving 4-36
    Options 4-40
    Refreshing 4-38
    Splitting 4-38
    Tailoring 4-38
Screen layout, screen mode Debug 4-2
Screen mode Debug
    Capabilities 1-3
    Communication 1-3; 4-4
    Description A-8
    Displays 4-6

Screen mode Debug (Contd)
    Ending a session 3-16; 4-41
    Entering commands 4-5
    Example 1-4
    Functions 4-1, 4, 14
    Locating text 4-36
    Restrictions 1-4
    Screen layout 4-2
    Session 3-2
    Starting a session 3-2
    Switching to 3-5
    Tailoring a screen 4-38
    Terminal set up 2-1
    Windows 4-6
SEEVAL function 3-13; 4-30
Session, Debug 1-2; 3-1
SETB command 3-7; 5-67
SET_BREAK command 3-7; 5-67
SETBRK function 3-7; 4-19
SET_DEBUG_RING command 6-10
SETDR command 6-10
SETPA command 3-1, 2, 4
SET_PROGRAM_ATTRIBUTES command 3-1, 2, 4
SETSM command 3-9; 5-85
SET_STEP_MODE command 3-9; 5-85
SETTA command 2-3
SET_TERMINAL_ATTRIBUTES command 2-3
Setting Breaks 3-7; 4-19; 5-67
Source
    Listing A-8
    Program A-8
    Window 4-6
SPLIT function 4-38
Splitting screens 4-38
Stack frame, displaying 5-61
Standard file A-9
Status variable A-9
Step mode
    Setting 3-7, 9; 4-26; 5-85
    Stepping through lines 3-7, 9; 4-26; 5-85
    Stepping through procedures 3-7, 9; 4-25; 5-85
STEP1 function 3-9; 4-26
STEPN function 3-9; 4-26
STOP Statement 3-4
Submitting comments 6
Subprogram traceback list 1-1; 5-33
Suspending execution 3-7; 4-19
Switching
    To line mode Debug 3-5; 4-40
    To screen mode Debug 3-5; 5-4
Symbolic
    Addressing A-9
    Capabilities 1-1
    Debugging 1-1; 2-6; A-9
Syntax A-9

T

Table of contents 3
Tables,
    Address 6-1
    Debug 2-6
Tailoring a Debug screen 4-38
Task A-9
Terminal
    Definition file A-9
    Definitions 2-3
    Full-screen characteristics 2-2
    Full-screen definition 2-3
    Full-screen interface 2-2
    Screen layout 4-2
    Set up for line mode 2-1
    Set up for screen mode 2-2
    Style 3-2
TERMINAL_MODEL parameter 2-3
Terminate break 2-4; 6-6
TERMINATE_COMMAND 2-4
Terminated program 6-8
Termination breaks
    Arithmetic overflow 3-11
    Arithmetic significance is lost 3-11
    Defaults 3-11
    Divide fault 3-8, 12
    Exponent overflow 3-8, 12
    Exponent underflow 3-12
    Floating-point indefinite 3-12
    Floating-point significance is lost 3-12
    Invalid business data processing data 3-12
    Pause 2-4; 6-6
    Terminate 2-4; 6-6
TRACE
    Display 4-10, 18
    Function 4-10, 18
Traceback list 1-1; 5-33; A-9

U

UP function 4-35
User breaks
    Activating 2-4
    ATTENTION_CHARACTER parameter 2-5
    CDCNET command 2-5
    NAM/CCP command 2-5
    Pause_break 2-4
    RESUME_COMMAND 2-4
    Terminate_break 2-4
    TERMINATE_COMMAND 2-4
    User_break_1 2-4
    User_break_2 2-4
Utility A-10

V

Value A-10
Variable A-10
Variable name A-10
Variable values
    Changing 3-13; 4-30, 32; 5-15
    Displaying 3-13; 4-40; 5-46, 97
Viking 721 1-4

W

WIDE function 4-39
Window
    Aligning 4-34
    Description 4-6
    Displays 4-6
    Dividing 4-38
    Help 4-10
    Keys 4-12
    Locating information 4-34
    Options 4-13
    Output 4-10
    Source 4-6

Z

Zoom-in display 3-2; 4-6, 16
Zoom-out display 3-2; 4-8, 17
ZMIN function 3-2; 4-6, 16
ZMOUT function 4-8, 17

Debug for NOS/VE Usage 60488213 B

We would like your comments on this manual. While writing it, we made some assumptions about who would use it and how it would be used. Your comments will help us improve this manual. Please take a few minutes to reply.

Who Are You?

___ Manager
___ Systems Analyst or Programmer
___ Applications Programmer
___ Operator
___ Other ____________________

How Do You Use This Manual?

___ As an Overview
___ To learn the Product/System
___ For Comprehensive Reference
___ For Quick Look-up

Which Are Helpful to You?

___ Command and Function Summaries
___ Related Manuals Appendix
___ Online Quick Reference
___ Other ____________

What programming languages do you use? ____

How Do You Like This Manual? Check those that apply.

Yes	Somewhat	No	
___	___	___	Is the manual easy to read (print size, page layout, and so on)?
___	___	___	Is it easy to understand?
___	___	___	Is the order of topics logical?
___	___	___	Are there enough examples?
___	___	___	Are the examples helpful? (___ Too simple ___ Too complex)
___	___	___	Is the technical information accurate?
___	___	___	Can you easily find what you want?
___	___	___	Do the illustrations help you?
___	___	___	Does the manual tell you what you need to know about the topic?

Comments? If applicable, note page number and paragraph.

Would you like a reply? ___ Yes ___ No

Continue on other side

From:

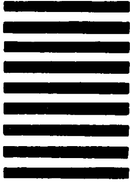
Name ______________________________ Company ______________________________

Address ______________________________ Date ____________________

______________________________ Phone No. ____________________

____

Please send program listing and output if applicable to your comment.



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 8241 MINNEAPOLIS, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CONTROL DATA

Technology and Publications Division

Mail Stop: SVL104

P.O. Box 3492

Sunnyvale, California 94088-3492

OLD FOLD

Comments (continued from other side)

Quick Index

Alphabetical List of Debug Screen Mode Functions

BACK 4-34
BKW 4-36
CHAVAL 4-32
DEAS 4-40
DELBRK 4-21
DOWN 4-35
FIRST 4-34
FWD 4-36
GOTO 4-27
HELP 4-16
HOME 4-35
HSPEED 4-22
KEYS 4-18
LAST 4-34
LEFT 4-38
LOCATE 4-37
LSPEED 4-25
MSPEED 4-25
NARROW 4-39
NEXT 4-35
OPTS 4-39
QUIT 4-40
REFRSH 4-38
RIGHT 4-39
SEEVAL 4-30
SETBRK 4-19
SPLIT 4-38
STEP1 4-26
STEPN 4-26
TRACE 4-18
UP 4-35
WIDE 4-39
ZMIN 4-16
ZMOUT 4-17

Alphabetical List of Debug Line Mode Commands

ACTIVATE_SCREEN........ 5-4
CHANGE_DEFAULTS........ 5-6
CHANGE_MEMORY.......... 5-11
CHANGE_PROGRAM_VALUE... 5-15
CHANGE_REGISTER........ 5-24
DELETE_BREAK........... 5-28
DISPLAY_BREAK.......... 5-30
DISPLAY_CALLS.......... 5-33
DISPLAY_DEBUGGING_ENVIRONMENT 5-36
DISPLAY_MEMORY......... 5-39
DISPLAY_PROGRAM_VALUE.. 5-46
DISPLAY_REGISTER....... 5-58
DISPLAY_STACK_FRAME.... 5-61
QUIT................... 5-65
RUN 5-66
SET_BREAK.............. 5-67
SET_STEP_MODE.......... 5-85

Alphabetical List of Debug Line Mode Functions

$CURRENT_LINE.......... 5-92
$CURRENT_MODULE........ 5-93
$CURRENT_PROCEDURE..... 5-94
$CURRENT_PVA........... 5-95
$MEMORY................ 5-96
$PROGRAM_VALUE......... 5-97
$REGISTER.............. 5-100

CONTROL DATA