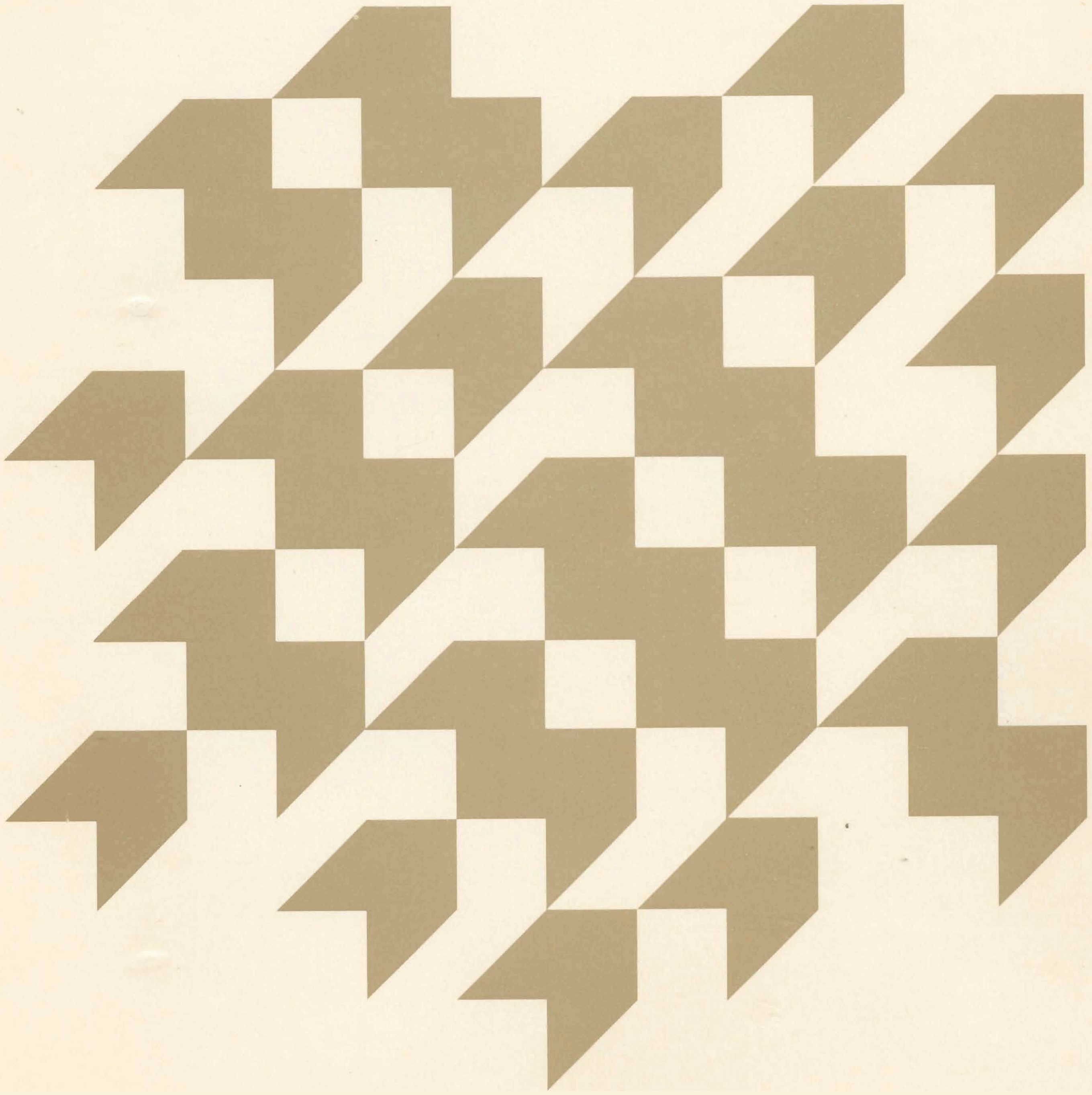
HEWLETT PACKARD

Symbolic Debug/1000

User's Manual



Symbolic Debug/1000

User's Manual

HEWLETT
PACKARD

HEWLETT-PACKARD COMPANY
Computer Languages Lab
11000 Wolfe Road
Cupertino, California 95014

MANUAL PART NO. 92860-90001
Printed in U.S.A. May 1983
E0583

PRINTING HISTORY

The Printing History below identifies the Edition of this Manual and any Updates that are included. Periodically, Update packages are distributed which contain replacement pages to be merged into the manual, including an updated copy of this Printing History page. Also, the update may contain write-in instructions.

Each reprinting of this manual will incorporate all past Updates, however, no new information will be added. Thus, the reprinted copy will be identical in content to prior printings of the same edition with its user-inserted update information. New editions of this manual will contain new information, as well as all Updates.

To determine what software manual edition and update is compatible with your current software revision code, refer to the appropriate Software Numbering Catalog, Software Product Catalog, or Diagnostic Configurator Manual.

First Edition	Aug 1982	
Update 1	May 1983	
Reprint	May 1983	Update 1 incorporated

Preface

This manual is a tutorial guide for the Symbolic Debug/1000 program. It is assumed that you know the rules of the programming language being used and know how to edit, compile, and load programs on an RTE system.

Chapter 1 outlines what the Symbolic Debug/1000 program can do. This chapter describes the features of the debugger and where it fits into the program development process. It also introduces some of the terms to be used later in the manual, e.g., breakpoint, current line, etc.

Chapter 2 describes the steps necessary to prepare a program for debugging. Programs to be debugged must be compiled and loaded with the symbolic debug option. Then Debug can be run to start the debugging session.

Chapter 3 describes the debugging rules and how to use the Debug program. Various examples are used to illustrate the common Debug functions.

Chapter 4 contains detailed descriptions of all the Debug commands. This chapter may be used as a reference section.

Chapter 5 describes profiling which is a special feature of the Debug program. This chapter also includes descriptions of the profiling commands.

Chapter 6 contains a list of unusual error conditions and their explanations. It also describes the corrective actions for some common problems.

Chapter 7 provides a discussion of advanced debugging topics. It includes some detailed descriptions of how Debug works. This chapter is for the more experienced users only.

Appendix A contains a list of error messages and explanations.

Table of Contents

Chapter 4 Debug Commands

Chapter 5 Profiling

Chapter 6 Troubleshooting Debug

Chapter 1
Debug Program Overview

Introduction

The Hewlett-Packard 92860A Symbolic Debug/1000 (Debug) program is a development tool that helps find bugs in your programs. It is an interactive, symbolic debugger for FORTRAN and Macro/1000 programs on RTE-6/VM and RTE-A based HP 1000 systems. It provides an alternative to using WRITE statements at various points in a program to debug the program. The output of these write statements might be as follows:

```
Arrived at subroutine SUBR
The value of variable XYZ at line 67 is  1.23E+6
```

This form of debugging is not interactive, meaning that you cannot change any information as the program runs. Non-interactive debugging works for most programs, but takes a great deal of time. When a bug is suspected in a program, you have to put in new WRITE statements at the suspected location, recompile, load, and execute the program.

The Debug program is an interactive debugger. It monitors the program while it runs and allows you to halt the program and look at data values, or to step from statement to statement in the program. You can interact with Debug to tell it to perform functions such as "execute program until it gets to line 652" or "display the contents of variable XYZ". Debug can start and stop a program during its execution, so that you may interact with the program at any particular state, allowing you to find bugs quickly.

Debug is a "symbolic debugger". It recognizes symbols used in the program being debugged as well as line numbers. For example, a FORTRAN statement such as

```
total = SIN(newnum+23.4)/finalscore
```

contains the following symbols: total, newnum, and finalscore. Debug allows you to use these symbols in debugging the program without specifying the memory addresses for the symbols.

Debug is normally used at the end of a program development cycle to get the program working. The program development steps are shown below.

1. Plan, outline, flowchart the program.
2. Edit the source program.
3. Run the appropriate compiler/assembler on your source.
4. If there are any compiler errors, go to step 2.
5. Load the program with the relocating loader (LINK).
6. Run Debug on your program.
7. If there are bugs in the code, go to step 2.

Operating Environment

Debug/1000 can be loaded and executed in the RTE-6/VM or the RTE-A operating system. The loading procedure and generation considerations are contained in the Symbolic Debug/1000 Configuration Guide (part no. 92860-90002).

In the RTE-6/VM operating system, the LINK relocatable loader must be loaded ino the system. Only LINK can produce the files Debug needs; therefore, LINK must be used to load programs to be debugged.

RTE-6/VM and RTE-A operating systems use different file name conventions. The names of files created and used by Debug under both operating systems are given throughout this manual. RTE-6/VM does not support code and data separation (CDS), so CDS information applies only to RTE-A.

The software revision of the operating system, LINK, FORTRAN 77, Macro/1000, and Debug must be 2226 or later.

Debug also requires a 262X or 264X display terminal. It will not work with other types of terminals.

Debug Capabilities

This section will familiarize you with the terms used later in this manual. A brief summary of the Debug capabilities is given below. Only major features are described in this section. Detailed descriptions of the Debug capabilities are contained in Chapters 4 and 5 of this manual.

Breakpoints
Proceed to next breakpoint
Display variables
Modify variables
Single stepping
Profiling
Clear breakpoints
List source code
Trace variables at breakpoints without stopping

Breakpoints

Breakpoint is a key feature of Debug that halts the execution of a program at a particular statement in the program. A breakpoint can be set at a location where a bug is suspected. During the debugging session, Debug will stop the program and display a screenful of source code to allow you to examine the program and enter other Debug commands to continue debugging.

A more complex and powerful type of breakpoint is called a conditional breakpoint. The conditional breakpoint takes effect only when certain variables in your program reach certain values. For example, you might instruct Debug to set a breakpoint at line 54 but stop program execution only if variable XYZ is zero. In this way, Debug monitors the data value XYZ as well as the instruction at line 54 and halts the program when XYZ is zero.

Proceed Until Breakpoint

Once a debug session has been initiated, Debug starts your program and displays a screenful of lines. At this point, there are no breakpoints or any other Debug commands. You can set breakpoints and enter any commands needed. When you are ready to run the program, enter the proceed command. This command allows the program to run until a breakpoint is reached or until completion.

Display Variables

When a program is halted at a breakpoint, you can examine any variable in the program with the Debug display command. Without Debug, some form of a WRITE statement is required to display a variable value. However, with Debug, there is no need to compile, load and run the program again each time you need to display another variable. The Debug display command can be used as often as desired to display variables.

Modify Variables

Variables in a program can be modified in the debugging session. You can instruct Debug to change the value of variables. When your program resumes execution, the new value will be in effect.

Single Step

Debug can also execute one line of code at a time. This allows you to observe your program "in slow motion" to check if it is executing as you intended.

List Source Code

Debug displays a block of your source code at the top of the terminal screen during the debug session. This block is typically 15 lines but can be modified with the View command. The display is adjusted as needed as the Debug commands are executed and shows where the program is being halted by a current line marker (>). At the beginning of the debug session, the current line is the first executable line. All subsequent source code listings are adjusted so that the current line is positioned at the center of the block.

Profiling and Other Capabilities

Profiling is another major feature of Debug. It shows the relative execution time of subroutines in a program. This helps in improving the execution time in applications where it is critical. Profiling is described in Chapter 5 of this manual.

In addition to the major features previously described in this chapter, Debug includes commands that perform the following:

* clear breakpoints.
* trace variables at breakpoints in the source code without stopping.
* show the names of the subroutines that called the current subroutine.
* examine contents of registers.
* change the size of source code display

All the Debug commands are explained in Chapter 4 of this manual. The common features of Debug are illustrated in Chapter 3. Debug also includes other special capabilities for the experienced users. These are discussed in Chapter 7, Advanced Topics.

Chapter 2
Program Preparation

This chapter describes how to start a debugging session. The program to be debugged must be compiled and loaded with the symbolic debug option. It then must be included in the Debug runstring. Debug must already be loaded in the system. The loading instruction for Debug is given in the Symbolic Debug/1000 Configuration Guide (part number 92860-90002).

Compiler Options

To use Debug to run a program, you must furnish Debug with the information needed to control program execution. This is accomplished by means of the compiler symbolic debug option S. The debug option may be included in the source program control statement, see examples shown below.

FORTRAN 77 Example:

```
FTN77,S
      PROGRAM ADDUP(4,99)
      INTEGER DCB(144)
        :
        :
```

Macro/1000 Example:

```
MACRO,S
      NAM ADDUP,4,99
DCB   BSS 144
        :
        :
```

If the ASMB control statement (old Assembler compatibility mode in Macro) is used, include "ASMB,S" as the control statement.

The symbolic debug option may also be added in the runstring of the compiler or the Macro Assembler. For example:

```
:RU,FTN7X,&MYPRG,-,-,,S

:RU,MACRO,&MYPRG,,,,S
```

When the debug option is specified, the language processor adds symbol table information to the output relocatable file. The symbol table information contains names, types, and locations of all of the symbols used in the program, as well as source file names, line numbers, and their locations.

No extra code is added to a program when it is compiled with the symbolic debug option. This option does not change the way the compiler generates code. It only includes the symbol table information in the relocatable output. Execution of the program is not affected by this option.

All programs to be debugged with Debug must be compiled with the symbolic debug option. However, Debug will still work even if only part of a program is compiled with the debug option. For example, the main is compiled with and the subroutine is compiled without the symbolic debug option. In this case, Debug can run the program but will not be able to control the subroutine execution.

Loader Options

Any program to be run with Debug must be loaded with LINK. LINK must be used with the debug option DE. The debug option can be entered in the LINK runstring as a +DE parameter. It can also be entered interactively or in a loader command file with the LINK DE command. For example, to load a sample program (ADDUP) with the debug option:

```
:RU,LINK,#ADDUP,+DE
```

The +DE parameter instructs LINK to prepare the program for debugging, in addition to loading the program specified in the loader command file #ADDUP. The +DE parameter may appear anywhere in the runstring.

When the +DE parameter is given, LINK produces two output files. The first is the usual type 6 program file. The other is the debug information file, containing symbol table and line number information. Under RTE-6/VM, the debug information file is given the name of the type 6 file produced, with an at sign (@) before the first character. If the type 6 file produced is called ADDUP, the debug information file is called @ADDUP. If there is an existing debug information file named @ADDUP, it is overwritten. If there is a file named @ADDUP that is not a debug information file, it is not overwritten. LINK displays an error message and Debug cannot run properly for ADDUP, because there is no debug information file.

Under RTE-A, the debug information file is given the name of the type 6 file produced, plus a .DBG extension. If the type 6 file is named ADDUP, the debug information file is named ADDUP.DBG As with RTE-6/VM, LINK will only overwite a debug information file with the same name.

If you do not specify the debug option, LINK will not produce a debug file. If an old debug file exists, the information in this file may not be applicable. This can cause much confusion. If the debug file for a program does not exist, Debug cannot be used to run the program. Therefore, it is recommended that you should always specify the LINK debug option when loading a program. This ensures that a current debug file is available for debugging.

The debug option should be specified even if you do not intend to run it with Debug. Because Debug does not add any code to your program, the program will run exactly the same way, either with or without the debug option. Even for programs that have already been debugged, you may still load them with the debug option; this way you will be ready for any new problems that may occur. The only reason not to use the debug option is to reduce the number of LINK output files and the size of the FORTRAN or Macro/1000 relocatables.

If a bug shows up in a program which was not loaded with the debug option, the program can be reloaded with the debug option and then debugged. Reloading the program will not change or eliminate the bug.

When loading a program to be debugged, make sure that it occupies a partition no larger than needed for that program. When Debug is executing, the program is locked in memory. Using a large partition unnecessarily may prevent another user from executing a large program.

Debug Runstring

To start the debugging session once a program is loaded, run Debug with the program name included in the runstring. The Debug runstring is shown below.

```
:RU,DEBUG,<pname>[:IH][,P1[,P2[,P3,...]]]
```

where:	<pname>	= program name
	:IH	= optional Debug parameter that inhibits copying. The program to be debugged must be restored to the system with the FMGR RP command and then removed at the end of the debug session. Default is to allow Debug to make a copy of pname and run that copy.
	P1,P2,P3...	= optional program parameters for the program being debugged.

Note that the RU entry is optional; you may use the implied run feature of the operating system. The syntax given above follows the convention described in the next chapter.

Chapter 3
Using Debug

Introduction

This chapter is a tutorial guide for beginning Debug users. Before using Debug, you should be familiar with the manual conventions and the Debug rules and limitations. A summary of debugging commands is also provided.

Manual Conventions

Throughout this manual, many examples are given to illustrate Debug operations. In the examples and command syntax descriptions, the following conventions are used:

1. The user entries are underlined.

2. Comments are shown indented to the right of the the sample screen displays to clarify a Debug message or an operator entry.

3. Optional parameters or operands are given within square brackets. For example:

   ```
   I file1 [file2]
   ```

   indicates that file2 is optional. It can be omitted and the Debug default rules will be followed. The Debug default rules are given in the Debug Syntax section and the appropriate command descriptions.

4. The term current line indicates that the line is to be executed next if program execution is allowed to proceed. This line is displayed with a current line marker ">".

5. Angle bracketed parameters are variables to be supplied by the user, e.g., <prog name>, <partition size>, etc.

Debug Syntax

The following is a brief description of the Debug syntax. This includes the common default rules and definition of terms used throughout the remainder of this manual. Detailed syntax descriptions are given in the individual command descriptions in Chapter 4 and Chapter 5.

1. Debug delimiter

   A space is the delimiter in the Debug command entry used to separate command and operands.

   A slash (/) is used to qualify an operand, to indicate the subroutine where it resides. Another slash may be used to qualify a subroutine, showing the segment of the program where the subroutine resides. For example, in the entry "D CRN/SUB1/SEG1", CRN resides in subroutine SUB1 which is in SEG1.

2. Debug prompts

   The Debug program prompt is:

   ```
   DEB85> _
   ```

   The two digit number is the terminal LU where Debug was scheduled. This number will be different for different terminals.

   When Debug is in the single-stepping mode, the prompt changes to:

   ```
   Step>> _
   ```

   When Debug is in the profiling mode, the prompt changes to:

   ```
   Overview> _
   ```

   The terminal cursor is positioned one space after the prompt. The cursor is shown in all examples where Debug is expecting an entry.

3. Defaults

   The default conditions associated with Debug commands are contained in the command description. However, Debug displays messages under certain situations such as an illegal entry and requests a yes/no answer. The default entry is given in the display in square brackets. Press the return key to specify the default entry.

For example, the following message is displayed when unable to single-step through a line (36):

```
Can't single step. Proceed to 37?[Y]
```

Pressing the return key is the same as entering Yes. Detailed explanations of the error messages are contained in the appropriate error message description in Appendix A.

4. Screen Display — As Debug executes, a screenful of source code is displayed. The display is memory locked if the terminal has that capability. For terminals that do not have that capability or for a 2621A terminal, Debug simulates memory lock by maintaining a record of the lines displayed and deletes only the top lines in the listing as the displayed lines exceed the terminal memory size.

5. Location — Location is a Debug command operand that can be any of the following:

   line number
   FORTRAN label number
   absolute address
   variable name (useful only with Macro routines)
   register name

6. Variable — Variable is a Debug command operand. It is a variable name that may include a subscript and can be a data item whose display type can be modified.

7. Value — Value is a Debug command operand that is a numerical constant as defined in FORTRAN or Macro/1000.

Debug Limitations

Programs to be debugged must be compiled and loaded with the symbolic debug option. Programs or subroutines not compiled and loaded with the symbolic option cannot be controlled by Debug. Therefore, only user written codes can be fully controlled by Debug. System library routines can be included in the source code but they are subject to item (1) of the list of limitations given below.

1. Some FORTRAN I/O subroutines cannot be single stepped. They must be skipped. Debug outputs a message to inform the user to skip to the next line.

2. Debug does not support privileged code. Any program or subroutine that goes privileged should not be debugged. The system may be halted during the debug session if a breakpoint is set inside privileged code.

3. EMA programs can be run with Debug but not VMA programs. If a VMA program (or subroutine) is run with Debug, a message is displayed to inform the user that variables in both EMA and VMA cannot be displayed or modified.

Debug Command Summary

A summary of the Debug commands and the command syntax is listed below. The usage of some of these commands is illustrated in this chapter. Detailed descriptions of all Debug commands are contained in Chapter 4 and Chapter 5.

Command	Description
Break B Location	Sets a breakpoint at location specified.
Clear C Location	Clears all breakpoints or the breakpoint set at the location specified.
Display D Variables	Displays variables.
Exit E	Aborts your program and exits Debug.
Goto G Location	Allows your program to proceed from location specified.
Help ? [command]	Displays a brief summary of the Debug commands
Include I f1 [f2]	Executes a set of commands from file f1 and optionally logs the output to file f2.
List L [Location]	Lists a screenful of source code in your program.
Modify M Variable Value	Modifies the value of variable.
Proceed P [Location]	Allows your program to proceed to the next breakpoint or specified line of source code. A temporary breakpoint is set at Location if it is supplied.
Step S [Into]	Steps to the next line of source code or optionally steps into the next subroutine.
Trace T Location	Shows location executed without stopping program.

View V number	Changes the number of source lines displayed on screen.
Where W	Shows the callers of the current subroutine.
Command stack /	Displays a list of Debug commands entered. Allows any of the commands to be selected and executed.

Start Debug Session

To start the debug session once your program is loaded, enter the following:

```
:RU,DEBUG,<prog name>
```

Debug displays the main and segment names as it builds the symbol table. Debug then displays a block of the source code and prompts for a command with:

```
DEB85> _
```

The cursor is positioned as shown above. At this point, Debug is expecting instructions. Any Debug commands can be entered. The program is under Debug control but execution will not occur until the Proceed command is entered.

The general format for entering instructions to Debug is:

```
DEB85> Command operand1 operand2 ... operandn
```

Command	is a Debug command. It can be spelled out or abbreviated to the first character.
operand1 through operandn	are the legal operands for the command entered. Refer to the command summary given previously or the appropriate command description for the operands allowed.

Debug accepts lower case inputs. However, all commands and operands entered in lower case are mapped to upper case letters before Debug reads them. This means that case-folding is in effect, both lower and upper case letters are the same to Debug.

Debug commands and operands are separated by a space, not a comma as with the File Manager.

Before proceeding to perform any of the Debug functions described in this chapter, you need to be very familiar with the Debug scoping rules and how to specify locations.

Scoping Rules

Many programs use the same variable names in more than one subroutine. For example, you may have the variable Temp in many subroutines. Each one is actually a different variable. To uniquely identify Temp, you have to tell Debug the scope of variable Temp. The scope of a variable name is the section of source code in which the variable is defined. For example, in program ADDUP, shown in the following Sample Source Code section, the scope of variable CRN is subroutine INIT. That is, the variable CRN may only be referenced in subroutine INIT. If the FORTRAN statement "CRN = YY" were to appear in subroutine CleanUp or in the main body of the program, it would not affect CRN in INIT.

Line numbers are also subject to scoping rules. Only the line numbers for the subroutine you are stopped in now can be specified without the subroutine name. All others must be qualified by subroutine name, e.g., 200/SUB3.

Scoping rules apply to Debug as you display variables or set breakpoints. For example, if your program is currently in subroutine INIT and you enter "D CRN", Debug will look for a variable in INIT to display. If it doesn't find a variable INIT in the current subroutine, it searches for INIT as a global variable common to the current subroutine. Debug follows the rules given below. It goes through each of these steps when you enter a display variable command or set a breakpoint. If it finds the variable or location specified during any of these steps, it displays the value or uses the location specified. Otherwise it goes on to the next step until the variable is found or outputs a message at the end of the sequence.

Scoping rules for symbols (variables or subroutine names) appearing as operands are as follows:

1. Determine if the symbol is local to the subroutine you are in now.

2. For a segmented program, determine if the symbol is global to the current segment.

3. Determine if the symbol is a global symbol to the main.

4. For a segmented program, determine if the symbol is a global symbol to any other segments (segments are searched in the order that LINK loaded them).

5. If this name is still not found, an error condition exists and Debug outputs a message and returns to the program prompt state. The variable may actually exist in a module not compiled with the debug option.

To specify symbols external to the current subroutine, the following rules apply:

1. If the subroutine is in the current segment or the main, append the subroutine name to the variable.

   Example:

   ```
   DEB85> B #10/SUBX          breaks at FORTRAN statement 10
                              in subroutine SUBX
   ```

2. If the subroutine is in another segment, append the subroutine name and the segment name.

   Example:

   ```
   DEB85> B #10/SUBX/SEG3     breaks at statement 10 in
                              subroutine SUBX of segment SEG3
   ```

   The segment name is required only if there are other subroutines named SUBX found in more than one segment.

Specifying Locations

As previously mentioned, a location may be a line number, a variable name, a FORTRAN label number, or an absolute address. Line or label numbers, or variable names may be further qualified to include the name of the subroutine in which they appear.

To specify a line number, enter the number after the Debug command. For example:

```
DEB85> B 234                  sets a breakpoint at line 234 of
                              the current subroutine.
```

To specify a variable name, enter the variable name. For example:

```
DEB85> D Temperature          displays the value of variable
                              Temperature.
```

Note that FORTRAN truncates variable names after 16 characters. Variable names must be limited to 16 characters or less.

To specify a FORTRAN label number, precede the number with a pound sign (#). For example:

```
DEB85> B #10                  sets a breakpoint at label 10 of
                              a current FORTRAN module.
```

To specify an octal absolute address as a location, enter the address followed by the letter B. For example:

```
DEB85> D 12042B               displays the contents of octal
                              location 12042.
```

To specify an absolute address as a decimal number, append the letter A to it. For example:

```
DEB85> D 5154A                displays the contents of decimal
                              location 5154.
```

Variable names and line or statement numbers, when appearing as shown above, apply only to the main program or the subroutine that you are currently examining (where Debug halts the program). To specify them in other subroutines, you must observe the scoping rules described in the Scoping Rules section. For CDS programs, absolute addresses refer only to the data segment, and addresses below 1024 (decimal) refer to the stack area.

Sample Source Code for Debug Examples

The following sections contain examples illustrating some of the Debug commands. It also describes the various steps in the debugging process. A sample program (shown below) is used to demonstrate the Help, Break, Proceed, Display, Modify, Clear, command stack, and Exit commands. This program opens a file named NUMBER, which resides on disc cartridge named YY, reads a series of ASCII numbers from the file, converts each of them to integer, and prints out the total. It is assumed that program ADDUP has been compiled with the debug option and loaded with the LINK loader.

```
01 ftn7x,s
02 $files 0,1
03        program addup(4,99), try out symbolic debug
04        implicit none
05
06        integer*2 FinalTotal,ios,number
07
08 c      This program reads a set of numbers from a file
09 c      and adds them up.
10
11        call init
12
13        FinalTotal = 0
14        do while (.true.)
```

```
        read(100,*,end=99,iostat=ios) number
        if (ios.ne.0) then
          write(1,*) 'Error #',ios
          stop
        else
        FinalTotal = FinalTotal + number
        endif
      end do

99    write(1,*) 'The final total is: ', FinalTotal

      call cleanup

      end


      subroutine init
      implicit none

      integer*2 ios
      character*6 crn

      crn = 'YY'

      open(100,file='NUMBER::'//crn,iostat=ios)
      if (ios.ne.0) then
        write(1,*) 'Can''t open, error #',ios
        stop
      endif

      return
      end


      subroutine cleanup
      implicit none

      close(100)

      return
      end
```

Note that the control statement in this FTN7X program contains the "S" option which is required for debugging. When this program is loaded, the LINK debug option "+DE" must be specified.

In the example shown, the file called NUMBER exists on disc cartridge YY and contains the following ASCII numbers:

```
12
34
99
```

When the Debug runstring is entered, Debug prepares the program for execution. The source file is displayed at the top half of your terminal screen and the desired commands can be entered in the bottom half. The source code is displayed in a terminal memory lock mode. For terminals that do not have this capability, Debug simulates memory lock by maintaining a record of the number of lines displayed and scrolling only the Debug display and operator entries, leaving the source code undisturbed. Your screen should be similar to the display shown below.

```
      04      implicit none
      05
      06      integer*2 FinalTotal,ios,number
      07
      08 c    This program reads a set of numbers from a file
      09 c    and adds them up.
      10
   >  11      call init
      12
      13      FinalTotal = 0
      14      do while (.true.)
      15          read(100,*,end=99,iostat=ios) number
      16          if (ios.ne.0) then
      17              write(1,*) 'Error #',ios
      18              stop

  DEB85> _
```

The prompt "DEB85>" indicates that Debug is ready to accept any debugging commands. Note that in the program listing, the arrow always points to the line of source code to be executed. Debug positions the source file so that the arrow is always near the middle of the lines displayed.

Setting a Breakpoint and Processing

The Debug commands for setting and proceeding to a breakpoint are B and P respectively. To set a breakpoint at line 15 and proceed to that line, enter the following:

```
DEB85> b 15
Breakpoint set at 15/ADDUP
DEB85> p
```

Debug executes the program and halts it at line 15. The top half of the terminal would scroll up and the arrow would point to line 15.

```
    08 c    This program reads a set of numbers from a file
    09 c    and adds them up.
    10
    11      call init
    12
    13      FinalTotal = 0
    14      do while (.true.)
  > 15          read(100,*,end=99,iostat=ios) number
    16          if (ios.ne.0) then
    17              write(1,*) 'Error #',ios
    18              stop
    19          else
    20          FinalTotal = FinalTotal + number
    21          endif
    22      end do
```

```
DEB85> b 15
Breakpoint set at 15/ADDUP
DEB85> p
DEB85> _
```

At this point in the execution of the program, subroutine INIT has been called, and lines 13 and 14 have been executed. The arrow is placed on line 15 to indicate that it will be the next line to be executed. Here you might want to examine the current contents of FinalTotal, using the Display (D) command.

Setting Conditional Breakpoints

A conditional breakpoint is a breakpoint that has an associated variable and a condition to test that variable. Debug tests the variable and allows the breakpoint to take effect only if the condition is true. Actually, the program stops each time it reaches that line, but Debug restarts the program if the condition is not true.

The following example sets a conditional breakpoint at line 20. Debug stops program execution only if the variable FinalTotal exceeds 34. The screen display when the program is halted at line 20 is shown below.

```
    13      FinalTotal = 0
    14      do while (.true.)
    15          read(100,*,end=99,iostat=ios) number
    16          if (ios.ne.0) then
    17              write(1,*) 'Error #',ios
    18              stop
    19          else
  > 20          FinalTotal = FinalTotal + number
    21          endif
    22      end do
    23
    24 99   write(1,*) 'The final total is: ', FinalTotal
    25
    26      call cleanup
    27
```

```
DEB85> b 20 FinalTotal > 34       Conditional breakpoint set at line 20.
Breakpoint set at 20/ADDUP
DEB85> p                          Starts program execution.
At 20/ADDUP  FINALTOTAL > 34      Debug halted program at line 20.
DEB85> _
```

Note that the DO loop in program ADDUP has been executed twice before the value of FinalTotal exceeds 34.

The conditional operators are:

=	equal
<>	not equal
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to

Debug does a substantial amount of processing each time the program reaches a location with a conditional breakpoint attached, whether or not the condition is true. Therefore, if you use conditional breakpoints at locations that are executed many times, execution may be slow. Refer to the Break command description in Chapter 4 for more information on conditional breakpoints.

At this point, you can examine the value of FinalTotal by using the Display (D) command.

Examine Contents of a Location

After setting a conditional breakpoint and executing the program, the contents of FinalTotal can be examined as follow:

```
DEB85> d FinalTotal
FINALTOTAL = 46
DEB85> _
```

More than one variable may be specified. Debug displays the value of each variable requested and prompts for another Debug command.

Displaying and Clearing Breakpoints

To display all breakpoints including conditional breakpoint, enter the Break command without any operand.

```
DEB85> B
   15/ADDUP
   20/ADDUP  FINALTOTAL > 34
DEB85> _
```

The list of breakpoints are displayed in the order of entry. Each line number is displayed with the program or subroutine where the line resides.

The Clear (C) command clears a breakpoint previously set. You must specify the line where the breakpoint was set. Enter the P command to run the program to the next breakpoint or to completion. Using the sample program ADDUP, to clear the breakpoint set at line 15:

```
DEB85> c 15
Cleared 15/ADDUP
```

To clear all the breakpoints previously set, enter the following:

```
DEB85> c *
All clear
DEB85> _
```

Use of the Step Command

The Step (S) command is used to step from one line of executable source code to the next. After the S command is entered, Debug interprets all subsequent carriage returns as step commands until another Debug command is entered. To remind you that carriage return is interpreted as a step command, Debug changes the prompt to "Step>>".

Only executable lines can be single-stepped. While in the single step mode, lines that do not produce executable code and lines that cannot be single-stepped are skipped by Debug. Some system library routines may contain code not compiled and linked with the symbolic debug option and are too complicated for Debug to single step. In this case, the following message is displayed:

```
Can't single step. Proceed to 16/ADDUP?[Y] Y
```

The above message is displayed because line 15 of program ADDUP contains a system routine that cannot be single-stepped. Debug expects a yes or no answer after displaying the above message. Either a Y or carriage return causes Debug to set a temporary breakpoint at line 16 and then executes the program up to line 16. If line 15 does a branch to a line other than line 16 or if you are not sure that line 16 is the next line to be executed, then you must answer no to the above prompt. Entering N causes Debug to display the following:

```
Unknown module
Step>> _
```

Any other entry will cause the "Can't single step" message to be repeated. After the No answer is entered, another Debug command can be entered to return from the single-step mode.

As the program is being single stepped, the source code display is scrolled down with the line marker pointed to the next line to be executed. Assuming the sample program is halted at line 15, the complete sequence of the single-step mode is as shown below.

```
DEB85> S
Can't single step. Proceed to 16/ADDUP? [Y]Y    Debug moves to line 16.
Step>> cr
Step>> _
```

At this point, the carriage return caused program execution to proceed to line 20, indicated by the current line marker at that line. The lines skipped are the ones that are not executed because the IF condition is false. To continue single stepping:

```
Step>> cr                        Debug moves to line 22.  Line 21 is
                                 skipped (no executable code).
Step>> cr                        Debug moves to line 15.
Step>> cr
Can't single step. Proceed to 16/ADDUP? [Y]sfs  Mistake in entry.
Can't single step. Proceed to 16/ADDUP? [Y]n
Module unknown
Step>> _
```

At this point, any Debug command can be entered to return from the single-step mode. Debug will execute the command and return to the program prompt state. Any illegal Debug command or a mistake in the entry will cause Debug to output a message and return to the program prompt state.

Exiting Debug

To exit Debug, use the Exit (E) command. This command also aborts your program. Debug saves the symbol table before terminating. Your program will not execute to completion. You may use this command any time you are through with the debugging session. Upon normal termination, control is returned to the operating system file manager.

```
DEB85> e
:
```

Step Through a Subroutine

To step through a subroutine, the program being debugged must be halted at the line that calls the subroutine. In the sample program ADDUP, line 11 calls subroutine INIT. After starting the debug session, Debug stops the program at the first executable line which is line 11. The sample display and terminal dialog are as follows:

```
    04      implicit none
    05
    06      integer*2 FinalTotal,ios,number
    07
    08 c    This program reads a set of numbers from a file
    09 c    and adds them up.
    10
  > 11      call init
    12
    13      FinalTotal = 0
    14      do while (.true.)
    15          read(100,*,end=99,iostat=ios) number
    16          if (ios.ne.0) then
    17              write(1,*) 'Error #',ios
    18              stop

  DEB85> _
```

The source code displayed consists of lines 4 through 18 with the line marker pointing to line 11. Entering the S command will move the pointer to line 13. At this point, all the lines in INIT have been executed. The last screen display and the Debug dialog are shown below.

```
    06      integer*2 FinalTotal,ios,number
    07
    08 c    This program reads a set of numbers from a file
    09 c    and adds them up.
    10
    11      call init
    12
  > 13      FinalTotal = 0
    14      do while (.true.)
    15          read(100,*,end=99,iostat=ios) number
    16          if (ios.ne.0) then
    17              write(1,*) 'Error #',ios
    18              stop
    19          else
    20          FinalTotal = FinalTotal + number

  DEB85> s
  Step>> _
```

Step Into a Subroutine

To step into the subroutine INIT, enter the Step-Into command with the program halted at line 11.

Exit the above debug session with the Exit command and start another session with the sample program ADDUP. Enter the Step Into (S I) command. The source code displayed will change to:

```
     30
     31         subroutine init
     32         implicit none
     33
     34         integer*2 ios
     35         character*6 crn
     36
   > 37         crn = 'YY'
     38
     39         open(100,file='NUMBER::'//crn,iostat=ios)
     40         if (ios.ne.0) then
     41             write(1,*) 'Can''t open, error #',ios
     42             stop
     43         endif
     44

   DEB85> S I
   At 37/INIT
   Step>> _
```

After the Step Into (S I) command is entered, the current line marker points to the first executable line of subroutine INIT which is line 37. You can single step through this subroutine. When you step through the RETURN statement, Debug returns to the main program, at line 13.

To continue with the sample program (halted at line 13 of the main), set a breakpoint at line 26 and proceed execution of ADDUP to that line.

```
Step>> b 26
Breakpoint set at 26/ADDUP
DEB85> p
```

When you arrive at line 26, the value of FinalTotal should be 145. To find out, display it by entering "D FinalTotal".

```
DEB85> D FinalTotal
FinalTotal = 145
```

The value of variable FinalTotal can be modified by the Modify (M) command.

Use of the Modify Command

The modify (M) command changes the value of a variable in the program. For example, to change the variable FinalTotal in sample program ADDUP to 1234 and display the new value:

```
DEB85> m FinalTotal 1234
FINALTOTAL: 145 => 1234
DEB85> d FinalTotal
FinalTotal = 1234
```

Subroutine Access

A subroutine can be accessed by appending a slash and the subroutine name in the location parameter in the Break, Display, Goto, Proceed, or Modify command. For example, to display the variable IERR in a subroutine called TAC, the entry is as follows:

```
DEB85> D IERR/TAC
```

In the sample program ADDUP, to set a breakpoint at the first executable line of subroutine CleanUp, enter the following.

```
DEB85> B CleanUp
Breakpoint set at 52/CLEANUP
DEB85> _
```

To set a breakpoint at a particular executable line in a subroutine, enter the line number, a slash, and the subroutine name. For example, to set a breakpoint at line 39 of subroutine INIT, enter:

```
DEB85> B 39/INIT
Breakpoint set at 39/INIT
DEB85> _
```

To find out the current subroutine that Debug is executing and the names of the routines that called it, use the Where (W) command. If there were any parameters passed, they would be displayed also. For example, assuming that the above breakpoint is set while the sample program is stopped at the first line of the program:

```
DEB85> p             executes program; halts at line 39.
DEB85> W
At 39/INIT
Callers:
     11/ADDUP
DEB85> _
```

Alternate Variable Display

Debug maintains certain information about each variable in your program. It knows whether a variable is type integer, type real, logical, etc. Whenever you display it, the variable is formatted using that type as default. To override the type, you can specify the display type by using the special display override directives shown below. There are other directives used for special variable displays and these are described in the Display command description contained in Chapter 4 of this manual.

```
:A - Assembly language
:B - binary integer
:C - character
:D - double precision real number (4 words)
:I - decimal integer (1 word)
:J - long decimal integer (2 words)
:L - logical (true/false)
:O - octal integer
:R - real (2 words)
:X - extended precision real (3 words)
:Z - hexadecimal integer
```

You may also override the default length of a variable by appending a number to the name. For example, to display the contents of a ten-element array INBUFF, you can enter:

```
DEB85> D INBUFF:10
```

This displays all 10 values of the array. Both the length and type overrides may be used in combination, to accomplish outputting variables in a number of different formats. Using the sample program ADDUP, display variable CRN when the program is halted at line 39. Then override the display type as follows.

```
DEB85> d crn/init
CRN/INIT = YY
DEB85> d crn/init:o
CRN/INIT:O = 54531b
DEB85> d crn/init:i
CRN/INIT:I = 22873
DEB85> crn/init:b
CRN/INIT:B = 0101100101011001
DEB85> _
```

Accessing Registers

For Macro/1000 programs with a MACRO control statement, the A and B registers are accessed by using the names "A" and "B". In this mode, the assembler defines the symbols "A" and "B" as registers.

For Macro/1000 programs with an ASMB control statement or FORTRAN 77 programs, the registers must be defined before they can be accessed. You can specify the A and B registers as locations (A) and (B) in Debug. The parentheses are required in specifying the register as a location.

The X, Y, E, O, Q, and Z registers are assigned the special names (X), (Y), (E), (O), (Q), and (Z). For example:

```
DEB85> D (Y)            displays the Y register

DEB85> D (A)            displays the A register in a FORTRAN or
                        ASMB program

DEB85> D A              displays the A register in a Macro/1000
                        program

DEB85> M (X) 10B        modifies the X register to 10 octal.

DEB85> D (Q)            displays the Q register in a CDS program.

DEB85> D (Z)            displays the Z register in a CDS program.
```

Chapter 4
Debug Commands

This chapter describes the commands used for program debugging. Profiling commands are contained in Chapter 5 of this manual. Commands described in this chapter are given in alphabetical order.

Break

Purpose: Sets a breakpoint, a conditional breakpoint, or displays current breakpoints.

```
Syntax:  B                                      (Displays all breakpoints)
      or B Location                             (Sets a breakpoint)
      or B Location Variable operator Value     (Sets a conditional breakpoint)
```

Location	is a line number, variable name, absolute address, or FORTRAN label number. Variable names are useful only in Macro programs where labels have symbolic names. The display type override directives are not valid for a location variable name.
Variable	is the name of a variable that is tested against the value specified. It is used only in setting a conditional breakpoint.
operator	is any of the following comparison operators:
Value	is a numerical constant. The data type is determined by the data type of the variable to the left of the operator.

Comparison operators (for operator):

=	equal to
<>	not equal to
>	greater than
<	less than
<=	less than or equal to
>=	greater than or equal to

Description:

This command sets a breakpoint in the program being debugged. The breakpoint can be either unconditional (normal case) or conditional.

A conditional breakpoint is a location in your program that has a variable and a condition attached to it. The program breaks each time it reaches that location. Debug returns control to the user if the condition is met. Otherwise, Debug restarts the program. For example,

```
DEBUG> B 23 xyz >= 88      breaks at line number 23 only if, before
                           line 23 is executed, variable XYZ is greater
                           than or equal to the value 88.
```

It is important that the scoping rules be observed. The line (or variable name) specified must not be ambiguous. If it is not in the current section of the source code, it must be qualified by appending a slash and the subroutine name. For example:

```
DEB85> B 23/SUBR XYZ >= 88
Breakpoint set at 23/SUBR
```

If a variable name is specified without any qualifier, it is assumed that it can be found in both the current subroutine and in SUBR. For the sake of clarity, it is recommended that variables to be tested at conditional breakpoints be fully qualified.

```
DEBUG> B 23/SUBR XYZ/SUBR >= 88
```

To display the list of current breakpoints that are set, enter the Break command without any operands. A list of breakpoints will be displayed in the order in which they are entered. For example:

```
DEB85> B
Breakpoints:
  12/MAIN
  234/SUB1
  22/SUB2/SEG3
```

A maximum number of 50 breakpoints is allowed. Conditional breakpoints require substantial amount of time for processing; appropriately 50 conditional breakpoints can be processed in one second. Therefore, conditional breakpoint should not be set at lines that are executed repetitively unless execution time is not critical.

For Macro/1000 code in certain situations, the line where a breakpoint is set may be the line following the line specified. When setting breakpoints at an entry point, Debug assumes that you arrive at the memory location by means of a JSB instruction. Because of this, Debug sets the breakpoint at the instruction FOLLOWING the entry point. For example, the symbol SUBR appears in an ENT statement and it stands for memory location 21042; all JSB instructions would write the return address at location 21042 and continue execution at address 21043. In anticipation of this, Debug sets a breakpoint at location 21043. This means that if the program arrives at location 21042 by means of a JMP instruction or a skip, or by any other means, it will not break until location 21043 is executed.

If line continuation on a statement is used in Macro/1000, the line number that Debug knows for that statement is the last line of the statement.

Clear

Purpose: Clears breakpoints or tracepoints that have been set.

Syntax:
```
   C Location
or C *
```

Location	is the line number, variable name, absolute address, or FORTRAN label number where the breakpoint was set.
*	Specifying the location as an asterisk clears all of the breakpoints and tracepoints currently defined. Example:

```
DEB85> c 15
Breakpoint cleared at 15/ADDUP
DEB85> c *
All clear                     This message will be displayed even
                              when there are no breakpoints.
```

Display

Purpose: Displays the values of variables or a line of code.

Syntax: D Variable [Variable ... Variable]

Variable	is a line number, variable name, a FORTRAN label number, or an absolute address. A variable name must be 16 characters or less. An absolute address may be displayed in octal or decimal by appending B or A to the address specified.

A number of different Variables may be specified. As many as can fit in one line of the command can be entered.

An at sign (@) preceding the Variable indicates a first level indirect address.

A single quote (') preceding Variable indicates that the Variable is used as a byte pointer.

Description:

The type of the variable display may be overridden by appending a colon (:) and an override directive to the Variable. The directives are listed below. Certain directives can be entered with a number to further specify the display, see examples shown below.

:A - Assembly language
:B - binary integer
:C - character
:D - double precision real number (4 words)
:I - decimal integer (1 word)
:J - long decimal integer (2 words)
:L - logical (true/false)
:O - octal integer
:R - real (2 words)
:X - extended precision real (3 words)
:Z - hexadecimal integer

```
DEB85> d xyz:o2          displays 2 octal numbers starting with xyz;
  XYZ:O2 = 44151b   0b   maximum of 127 display count can be specified.

DEB85> d xyz:c2          displays 2 characters; up to 78 characters
  XYZ:C2 = Hi            can be specified.

DEB85> d XYZ:O           displays XYZ as an octal number.
 XYZ:O = 44151b          shows the value of variable XYZ.

DEB85> D @XYZ:o          uses XYZ as an indirect pointer.
 @XYZ:O = 24567b         shows the value at address 44151B.
```

Multiple levels of indirect addresses may be specified by preceding the variable with more "@" signs. For example:

```
DEB85> D @@XYZ           displays the contents of memory
                         location 24567B.
```

Variables can also be used as byte addresses. Preceding the variable with a single quote (') instructs Debug to read the variable as a byte address and display the data of the location that it references. To display a word pointed to by the byte address contained in variable XYZ, enter the following:

```
    DEB85> d 'XYZ
```

The memory locations for a particular variable may be specified in the Display, Modify, Break, or Trace command. This is done by appending a colon (:) and a number between 1 and 127 to the variable name. This number may appear before or after any of the keywords described above, or may appear alone. For example:

```
    DEB85> d xyz              displays memory location of xyz
      XYZ = 18537

    DEB85> d xyz:5            displays xyz and the next 4 values (5 total)
    XYZ:5 = 18537   0   -26620    5230    7314
```

Debug remembers the operands that were entered on the previous Display command, and if the command appears with no operand, it displays the variable previously specified. For example:

```
    DEB85> d CRN
    CRN/INIT = YY
    DEB85> D
    CRN/INIT = YY
```

Variables that appear in segments other than the one that is currently in memory cannot be displayed. An exception to this rule is the SAVE variables as defined in the FORTRAN or Macro/1000 progams.

Exit

Purpose: Terminates Debug and the program being debugged.

Syntax: E

Description:

Debug terminates the program being debugged without displaying message regardless of the state it is in. The symbol table information is saved in file @<program name>.

Goto

Purpose: Moves Debug to another line in the program being debugged other than indicated by the current line marker.

Syntax: G Location

Location	is a line number, variable name, FORTRAN label number, or absolute address. The display type override directives cannot be used with the variable name. This operand is useful only in Macro/1000 programs.

Description:

The location operand must be in the segment that your program is executing. The program will not execute which means that the lines between the last current line and the present current line have not been executed. Therefore, use this command with caution to avoid confusion. It is not a recommended practice to go into the middle of a DO loop or into another subroutine.

The scoping rules and restrictions for variables and line numbers also apply to this command. The location operand should be qualified with the subroutine and/or segment name if it is not in the current subroutine. A line that does not produce executable code or contains code not compiled and linked with the symbolic debug option cannot be specified in the G command.

Include

Purpose: Instructs Debug to read a set of Debug commands from a command file and list to an output file.

Syntax: I f1 [f2]

f1	is the name of a file containing one or more Debug command, not including another I command.

f2	is an optional output file name. If specified, all Debug outputs will be stored in this file as well as being displayed at the terminal.
	If specified, Debug creates f2 if it does not exist. Default is no output file.

Description:

Nesting of the I command is not allowed.

After opening f1, Debug executes the commands from that file until the end of file is reached. At that point, Debug returns to the interactive mode, displaying the DEBUG> prompt. There should be one Debug command per line in the file. If an error occurs, Debug reports the error and reads the next line from the file.

To specify only a log file, enter the following:

```
    DEB85> I 1 f2
```

This entry specifies that input is from the terminal and file f2 is the log file. To terminate logging in this case, enter a CNTL-D (pressing the CNTL and D keys simultaneously).

If f2 is specified, only messages output by Debug are logged into this file; there is no program output. This is useful for acquiring extensive logs of trace information, which can be helpful when tracking a subtle problem.

Help

Purpose: Displays a short summary of Debug commands.

Syntax:

```
?

or ?  command
```

Description:

The help command displays a short summary of the Debug commands available. When entered with a command (or a single character), a brief description of that command is displayed.

List

Purpose: List a number of lines of source code.

Syntax:

```
   L [Location]
or L [line number[file name]]
or L [-[line count]]
or L [.]
```

Default is to display the next block of lines with a two-line overlap.

Location	is a line number, variable name, FORTRAN lable number, or absolute address.
line number [file name]	lists the specified line number from the program being debugged (default file name) or from the optional file named.
-[line count]	is a negative number that specifies a number of lines backward from the present block of lines displayed. For example, if lines 4 through 18 are displayed, entering L -3 will display lines 1 through 15. Default (only the minus sign entered) is the previous block of lines displayed.
. (period)	displays a block of lines centered on the current line (the line to be executed next).

Description:

The L command lists a number of lines. When a breakpoint is encountered, Debug positions the source file of the subroutine being debugged on the screen in such a way that the line about to be executed appears in the middle of the display. This line is called the current line and is identified with a marker (>). The list command displays a new block of lines according to the operand specified, with the specified line positioned either at or near the center of the block. The default number of lines displayed is 15 or a number changed by the View command.

With the List command, the rules for specifying locations are not strictly enforced. Debug allows you to specify any line number within the source file, including comment lines. If the location specified contains an entry point name and no line number, then the first executable line of the module is displayed at or near the center of the block of lines listed.

The L command can be used without any operand to scan through the source code.

The second form of the List command syntax (with line number and file name specified) is the most general form. It can be used to list source files, or any text files. If the file name is omitted, the source code of the program being debugged is listed. To list another file, a line number, followed by a space, followed by a file name is required. Once a file name has been specified, all subsequent list commands, except one with the period (.) operand, displays the optional file specified.

The next two forms of the List command syntax are used to list different parts of the file that currently appear on the screen. The negative line count allows you to go back a number of lines or to the previous screenful. For example, "L -20" lists a screenful of lines starting 20 lines above where the screen currently is positioned.

The last form of the List command syntax ("L .") is a way to reset the screen back to show the lines around the line you are about to execute. This command is used after you have scanned through the file or other files. It can also be used to refresh the screen in the event that your program does some I/O to the screen.

Note that the L command does not change where the program is stopped. Therefore, the Debug scoping rules do not apply to this command.

A special feature of Debug is to divert the input and output to a terminal other than the one from which a program is run. It is implemented with a runstring directive. The syntax is:

```
:RU,DEBUG,+L:n,[<other parameters>]
```

where +L:n must be the first parameter and
n is the terminal logical unit number.

This feature is used for debugging interactive programs that require frequent terminal I/O and utilize many of the terminal special features such as graphic displays.

Modify

Purpose:	Alters the value of a variable.	
Syntax:	M Variable Value	
	Variable	is the variable name to be modified.
	Value	is the new value. It must be a numerical constant.

Description:

Debug interprets the value as being the same type as the variable data type. For example, if the command "M X 23" is entered, and X is a 4-word floating point number, the operand 23 is formatted as 4-word floating point before it is placed into the memory location for X.

The type of the variable may be overridden by using one of the type-override characters (B, C, I, L, O, Z, R, X, D, or J) described with the D command. For example:

```
DEBUG> d X              X is a 4-word floating point number
  X = 234.453           The current value of X displayed
DEBUG M X:C 'hello'     The operand is interpreted as a character
                        string.  The quotes must be present.
```

In this example, the number of characters modified is the character string. The remaining memory locations are left as zeros in the 4-word floating point number.

Proceed

Purpose: Transfers control to the program being debugged until the next breakpoint is encountered, the program terminates, or a program violation such as an MP or DM error occurs.

Syntax: [number] P [Location]

number	is an optional incremental value from 1 to 32767 that specifies the number of breakpoints Debug should execute before returning control to the user. The space between this operand and the P command is not required. Default is 1.
Location	is an optional operand that defines a temporary breakpoint. This means that Debug sets a breakpoint there, transfers control to your program, stops at that breakpoint, and then clears that breakpoint upon return to Debug.

Description:

The optional number parameter is most useful in loops, where a break is desired after a number of loop iterations. For example:

```
DEB85> 40P              proceeds for 40 breakpoints.
```

If the location operand is specified, it must be a line with executable code. Otherwise, Debug displays the following message:

```
DEB85> p 23
P 23
  ^
Bad line number
DEB85> _
```

Step

Purpose: Allows line-by-line execution of the program being debugged.

Syntax: [number] S [keyword]

number	is an incremental value indicating how many lines to execute before returning control to the user. Default is 1. The space between Number and S is optional.
keyword	is either:
	I for stepping into a subroutine
or	T [Variables] for tracing variables

Description:

This command transfers control to the user program only for execution of the current line or number of lines specified by the incremental number parameter. Then control is returned to the user to allow examination of the results of that line.

In the case where the current line to be executed contains calls to subroutines or functions that the user would like to debug, the keyword may be specified (I). This instructs Debug to step into the subroutine called on that line. By default, the subroutine call would be stepped over, and control returned to the user on the line following the call. For example, assume that a program is currently positioned at the line shown below.

```
     :
> 234  Call Subr (x, y, z)
     :
```

Entering the S command would execute subroutine SUBR, and return control to line 235. If there were any breakpoints in subroutine SUBR, the single step command would cause Debug to break at those breakpoints. Entering "S I" would transfer control to the first executable line of code in subroutine SUBR.

The "Step Into" command requires that the subroutine to be executed be compiled with the debug option turned on (refer to the compiler option section in Chapter 2). If it was not compiled with debug option, Debug steps over the subroutine call, ignoring the keyword option.

If the subroutine does not use a standard call sequence and contains debug information, then Debug will always step into the subroutine.

Note that Debug displays a new prompt (Step>>) when the Step command is entered. Subsequently, a carriage return is interpreted as the Step command until any other command is entered.

Another option for the Step command allows you to trace the values of a list of variables after every step. This is done with the keyword option T (trace). For example, "S T var1 var2" instructs Debug to display the values of the variables var1 and var2 after every single step. This is done when you are in single step mode, and Debug stops displaying these variables when you move out of the single step mode. Entering "S T" (with no variable list) returns you to the previous variable list entered.

Trace

Purpose:	Sets a tracepoint in the program to be debugged.
Syntax:	Trace Location [Variables]

Location	is where the tracepoint is set. It follows the location specification rules.
Variables	specifies the optional list of variables to display when the tracepoint is executed.

Description:

A tracepoint is a special breakpoint that does not cause control to be returned to the user. Instead, when encountering a tracepoint, Debug simply indicates that the tracepoint has been executed by displaying the line number where a tracepoint was set, and allows the program to continue. Using the tracepoint command, you can follow the execution of the program without having too many interruptions.

The trace command also allows an optional list of variables to be displayed when the tracepoint is executed. If the T command appears with no operand, a list of the current tracepoints that have been set is displayed. Since tracepoints are really special breakpoints, a total of fifty breakpoints and tracepoints are allowed.

Examples:

Set a tracepoint at line 24 of current subroutine.

```
DEB85> T 24
  Tracepoint set at 24/ADDUP
DEB85> P
  At 24/ADDUP   This message is displayed every time 24 is executed.
```

Set a tracepoint at line 20 and watch variable FinalTotal.

```
DEB85> T 20 FinalTotal
  Tracepoint set at 20/ADDUP
DEB85> T        Displays the current tracepoints that are set
Tracepoints:
  20/ADDUP  Trace:  FINALTOTAL
DEB85> _
```

As with the Break command, the scope of the variables that appear in the Trace command must be accessible to Debug both when the tracepoint is set and when the tracepoint is executed. You may specify as many variables as you can enter in one command line.

View

Purpose:	Changes the number of lines of source code displayed on the terminal.
Syntax:	V [number]

number — is a number between 3 and 19. Default is 15 lines. The number of lines displayed will be changed to an odd number so that the line specified can be displayed at the center of the block of lines.

Where

Purpose: Shows current line, subroutine, and segment of the program being debugged. Also shows the subroutine call chain executed before that point.

Syntax: W

Description:

There are no operands for this command.

The output resulting from this command includes several lines displaying the following information. One line showing the line number, subroutine name, and segment name of the routine your program is currently executing. The next line displays the subroutine that called the current one, and the values of the parameters that were passed. Whenever applicable, there is another line for the caller of that subroutine, and the parameters, and lines for the caller of the caller, etc. The output stops when the main program is reached, or the main subroutine of a segment is reached.

The values of parameters displayed by the Where command represent those at the time that the command is entered, NOT at the time the subroutine was called. If the parameter value has changed since the subroutine call occurred, the NEW value is displayed. The parameters may not be in the same order as the source code. The names are the names used by the subroutine called.

When your program is stopped in subroutines that were not compiled with debug option turned on, or in RTE system library routines, the Where command does not show you the module name. Instead the octal address of where your program is stopped is displayed. The callers of this unknown subroutine are not displayed.

The following example illustrates the use of the Where command. It assumes that the main Prog called StartUp that called OpenFiles which in turn called FileError. Assume that the program is currently stopped in subroutine FileError. The result of the Where command is shown below.

```
At 35/FileError
Callers:
  340/OpenFiles:   DCB = 0     ErrorFlag = false
  122/StartUp:    CommandBuffer = 12345
  34/Prog
```

The variables DCB and ErrorFlag are parameters passed from OpenFiles to FileError, and CommandBuffer is a parameter passed from Prog to StartUp.

/ (Command Stack)

Purpose: Displays Debug commands entered.
Allows any one to be selected, modified with the terminal local editing keys, and executed.

Syntax: / [n]

n	is an optional number indicating the number of commands to be displayed. Default is up to 12 commands.

Description:

The last 12 unique commands are listed in the order of entry. The cursor is positioned at a blank line at the bottom of the stack. The terminal cursor vertical positioning keys can be used to select the command desired. The line selected may be modified. Pressing the return key executes that command line. Positioning the cursor at a blank line and pressing the return key returns to the Debug program prompt state.

The command stack is saved in the debug information file when Debug is exited. If multiple users are debugging the same program, the command stack that is restored at the start of the debug session is the last one saved.

Chapter 5 Profiling

This chapter describes the profiling capability of Debug/1000. A brief explanation of a program profiler is provided at the beginning of this chapter followed by a description of the Debug profiling features. A detailed description of the Debug profiling commands is also provided here.

What is a Program Profiler?

A program profiler is a tool used to identify what parts of your program are taking the most time, to help you make your program run faster. A symbolic debug profiler knows the subroutine names in the program. Therefore a histogram of which subroutines are taking the most time can be displayed. These features are available with the Symbolic Debug/1000 program.

Debug Profiling Features

The Debug profiling capability is a separate mode from the debug mode. You can initiate a debugging session and switch into the profiling mode. However, after the profiling mode has started, you cannot return to the debug mode. You must initiate another debug session for further debugging. The default operating mode is for debugging. To switch to the profiling mode, the Overview command is used.

Debug, while in the profiling mode, provides the following features:

Plot a histogram of subroutines
Plot a histogram of a subroutine
Log profiling data to a specified file
Display lines of source code
Terminate profiling and exit Debug

The Debug profiling commands that provide the features mentioned above are the Histogram (H), Include (I), List (L), and Exit (E) commands. The rules for entering profiling commands are the same as the Debug commands previously described. Only the four profiling commands can be used in profiling mode, no other Debug commands are allowed.

In performing profiling functions, Debug alternately runs and waits at 10 msec intervals and samples the program location counter for your program at these intervals. Since it attributes the entire 10 msec time frame to your program, it is essential that no other programs take up this time. Therefore, you should use only a system that is not busy.

Sample Profiling Session

A sample session in RTE-6/VM is given to familiarize you with profiling. In this example, it is assumed that all modules of the program have been compiled and linked with the debug option. A normal debug session is started as previously described. The Overview command initiates the profiling session. It instructs Debug to gather profiling data about the program by running the program to completion. If there are breakpoints set, they are automatically cleared. The following messages are displayed on the screen:

```
DEB85> O
Putting data in file =MYP85
All clear                         (All breakpoints cleared)
Taking data now
  (Any program output is listed here)
Finished taking data
Overview> _
```

The file =MYP85 is created by Debug to hold profiling data. This file may be kept at the end of a profiling session and used later to obtain more plots. The file name is derived by putting an equal sign in front of the program name. If a file with the same name exists, Debug asks for permission to purge the file. If you answer no, Debug terminates so that you may rename the existing file. A yes answer purges the existing file.

The message "Taking data now" indicates that the program is running and Debug is monitoring it.

During program execution, Debug is taking samples of the program location counter every 10 msec. This data, along with some information regarding segments, is kept in the data file. Debug counts the time that the program spends waiting on I/O, and the time spent executing.

When the program completes, Debug outputs the message "Finished taking data" and prompts for the next command. A different prompt is displayed and only the profiling commands are in effect.

The H command provides information about where the program is spending its time. To continue the sample profiling session:

```
Overview> h
Processing profile data

Routine               Amount   Histogram
-------               ------   ---------
GIMME                    8%  ****************
ATOL                     7%  ***************
GETCH                    5%  **********
WHERESLAVE               5%  **********
RPEEK                    4%  *********
IN                       4%  ********
ADROF                    4%  ********
CLEARB                   1%  **
MVW                      1%  **
LTOA                     1%  **
SETB                     1%  **
DOPOK                    1%  **
DEST                     1%  *
GTCLK                    1%  *
WINDX                    1%  *
SSTEP/DEBU1              1%  *
MBT                      1%  *
WHERE                    1%  *
Other (known code)       5%  **********
Other (unknown code)    18%  ************************************
I/O suspend             11%  **********************
Wait state              20%  *****************************************
System noise             1%  *

Overview>
```

In the sample histogram, the subroutines that spent the most amount of the total time are plotted with the busiest at the top. The percentage is rounded up to the nearest whole number which should be used only for comparison. Any of the subroutines that took up less than .5 % of the execution time are not listed by name but are summed up and listed under "Other (known code)". These added up to 5% in the example plot. Time spent in subroutines which have been compiled with ASMB or FTN4 or have been modified by OLDRE appear in the "Other (unknown code)" line. Most of the sytem library routines fall into this category. The I/O suspend category includes time spent by the program waiting on unbuffered I/O devices, e.g., discs. The wait state is time spent while waiting on another program or on buffered I/O device suspend or other waiting conditions. These states are different for RTE-A because there are different types and more wait states. System noise occurs whenever Debug samples the program and finds that it is preempted by the operating system.

To plot a histogram of subroutine ATOL in the sample program:

```
Overview? h atol
Profile for module ATOL

      7% of total time spent here

  Line No.            Amount  Histogram
  --------            ------  ---------
     764                2%  ****
     773                1%  *
     778                1%  *
     783                1%  **
     786                1%  **
     790                1%  *
     802                1%  **
     811                8%  *****************
     812               10%  **********************
     813                7%  ****************
     821                8%  *****************
     822               17%  *************************************
     824                9%  *******************
     827               13%  ****************************
     840                1%  *
     843                1%  *
     848               18%  ****************************************
     852                1%  *
     853                1%  **
     868                1%  *
```

In this plot, any line number that took up any execution time is listed, along the corresponding percent time. Comment lines and lines without any data samples are not listed.

Description of Profiling Commands

The Debug profiling commands are:

Overview	Initiates a profiling session
Histogram	Plots a histogram
Include	Logs listing to a file
Exit	Terminates profiling and Debug
List	Lists source code

These commands are described in the following sections.

Overview

Purpose:	Initiates the profiling session.
Syntax:	O [file name]

file name	specifies the file that contains the profiling data to be accessed by Debug. If omitted, Debug creates a file using the program name preceded with the equal (=) sign.

Description:

This command instructs Debug to begin taking profiling data on your program.

If the optional file descriptor is specified, Debug will not create a new data file, nor will it run your program to collect new data. It will open the file to obtain the profiling data.

Histogram

Purpose:	Plots a histogram.
Syntax:	H [subroutine name]

subroutine name	specifies that the code in the subroutine specified is to be examined and plotted. Default is to plot all the routines in the program.

Description:

The H command processes and examines the data taken as a result of the Overview command, either from the file created by Debug or that specified in the command string.

If no subroutine name is specified, all routines taking up more than .5% of total execution time will be plotted, starting from the busiest. All routines taking less than .5% of the total time will be grouped in a collective category, "Other (known code)". Samples which occur in subroutines such as library routines that do not have Debug information are grouped into a category indicated as "Other (unknown code)".

Include

Purpose: Instructs Debug to read a set of profiling commands from a command file and to list to an output file.

Syntax: I f1 [f2]

f1	is the name of a file containing one or more profiling commands, not including another I command.
f2	is an optional output file name. If specified, all Debug outputs will be stored in this file as well as being displayed at the terminal. If specified, Debug creates f2 if it does not exist. Default is no output file.

Description:

Nesting of the I command is not allowed.

After opening f1, Debug executes the commands from that file until the end of file is reached. At that point, Debug returns to the interactive mode, displaying the Overview> prompt. There should be one profiling command per line in the file. If an error occurs, Debug reports the error and reads the next line from the file.

To specify only a log file, enter the following:

```
     Overview> I 1 f2
```

This entry specifies that input is from the terminal and file f2 is the log file. To terminate logging in this case, enter a CNTL-D (pressing the CNTL and D keys simultaneously).

If f2 is specified, only messages output by Debug are logged into this file; there is no program output. This is useful for acquiring extensive logs of trace information.

Exit

Purpose: Terminates profiling session and exits Debug.

Syntax: E

Desciption:

The Exit command stops Debug. It is the same as the E command in the normal (debugging) operation.

List

The list command in the profiling mode is similar to the list command in the debugging mode. The only difference is that in the profiling mode, the terminal memory lock feature is turned off. Refer to Chapter 4 for details.

Chapter 6
Troubleshooting Debug

This chapter is intended to help you with some of the common problems in the debugging session and to explain the error conditions. It describes the symptoms, the probable causes, and whenever possible, the corrective actions. A complete list of Debug error messages is given in Appendix A of this manual.

Start Up Errors

The following sections describe the errors that can occur between the time you run Debug and the time Debug begins execution.

Compiling Errors

At the start of a debugging session, any one of the following messages may be displayed due to a compiling error:

```
Can't find main program's debug info
Was program linked with debug option?
Can't find starting line
```

All of these errors occur because Debug is unable to find the debug information for the main module of your program. Either the starting address was missing, or there is no entry point for the main module.

First, make sure that the main program was compiled with the debug option.

If you have compiled the main with the debug option and your program is written in Macro, you may need to add an ENT statement to the main module. For every Macro main program, the transfer address is specified on the END statement of that module. For Debug to work, there must be an entry point to that label.

Scratch File Error

The scratch file error message is:

```
Debug: Error in creating scratch file
```

This error happens when Debug initializes and an FMP error occurs while trying to build a scratch file containing the names of the symbols in your program. Debug creates this file on the same disc cartridge as the one your executable (type-6) file was placed. If a type-6 file is on LU 2 or LU 3, only the system manager can run Debug on these files. It is recommended that you pack that disc cartridge and try again. Debug requires 96 blocks for a type-1 scratch file, and will require one or more extents of the file for programs over 3000 lines or more.

Also during startup, Debug may display a file error on the debug information file. Under RTE-6/VM, you might see the following error message:

```
File not found:  @TEST
```

Under RTE-A, the debug information file is identified by the .DBG extension, so the error appears as:

```
File not found:  TEST.DBG
```

The debug information file is on the same disc cartridge as the type 6 executable program file. The debug information file contains symbol names and line number information, and is required for debugging. If you get this error message, the file has been purged. Reload the program with the debug option to create a debug information file.

Symbol Table Error

The "Symbol table overflow" message occurs when Debug undergoes initialization but finds that your program contains too many symbols for its symbol table space. It is recommended that you select the modules in your program that may not need debugging and recompile them without the debug option. The purpose is to reduce the number of symbols that Debug maintains for your program.

The symbols in this message refer to the local variables, statement numbers, procedure names, line numbers, and segment names used in your program. If you find that your program has too many symbols for Debug, but you still want to recompile with Debug option turned on, there are two alternatives.

One common source of symbol table overflow is declaring many FORTRAN COMMON blocks in modules that do not use them. Debug has to keep track of these excess symbols (separate copies for each subroutine, since the names may be different), whether or not the program uses them. If you have this problem, reduce the COMMON declarations in your subroutines to include only those COMMON blocks needed.

Another cause of symbol table overflow is the existence of numerous copies of a subroutine in many segments. These subroutines could be moved to the main if allowed by the size constraints.

MP or DM Violations

If your program aborts with the following message:

```
     Violation at xxx
```

where xxx is a memory location, your program has caused a memory protect (MP) or dynamic memory (DM) violation. Debug was aware of this condition and has prevented the operating system from throwing away your program. At this point in the debugging session, you may not proceed and execute your program without modifying the offending instruction. But you are free to examine memory locations and list lines of code in an effort to find out why your program caused the memory violation. If Debug gives you a line number and a subroutine name for the place the violation has occurred, then this line of code contains the violating instructions. If, however, Debug gives you an octal address of the violation, it occurred in some code that was not compiled with the Debug option. At this point, it is necessary to consult a load map to find out what subroutine the violation occurred in.

Erroneous Current Line Indication

The current line that will be executed if the program continues is indicated in the Debug display with a marker ">". If the current line marker points to a line other than the one your program is currently at, the source file has been changed since it was last loaded. The program must be recompiled and reloaded with LINK.

Abnormal Debug Program Abort

If Debug itself aborts for any reason, the event should be reported to your system manager. Program clean up is necessary. Debug has made a copy of your program, usually under the name XXXnn where XXX is the first 3 characters of the program name and nn is the terminal LU number of your terminal. At this point, that program is currently in memory, and locked there, waiting for Debug to release it. Because the aborted Debug cannot release this program, you must release it with the File Manager "OF,XXXnn,8" command.

Chapter 7
Advanced Topics

This chapter provides information about certain program displays, error conditions, and how Debug works. The information is for the more advanced Debug users.

Debug/Test Program in Separate Partitions

If Debug runs in a separate partition from the test program, how can it display variables in the program code space?

Debug maps in pages in the test program code that contain the needed memory locations. This is similar to how an EMA program maps in various locations in the EMA area where access is required.

Unusual Displays

During initialization, Debug displays the following message:

```
  Type carriage return
```

This only lasts for a split second. This is because Debug uses the split screen feature of the terminal to put source code at the top of the screen and commands at the bottom and it has to know what kind of terminal is being used. Debug performs a terminal status request, sending a special sequence of characters to the terminal and the response tells Debug the type of terminal being used. Debug displays "Type carriage return" on the screen, and waits for a response. If it gets an answer from the terminal, it erases the screen. Otherwise, the user must enter the carriage return. After a brief timeout, if the carriage return has not been received, Debug displays:

```
  Only handle 264x and 262x terminals
```

Sometimes when Debug has to refresh the source code part of the 2645 screen (e.g., when a new subroutine is entered), some lines disappear from the bottom window of the screen. What is causing this?

This happens only when the 2645 terminal has completely filled up its memory for lines to be displayed on the screen. The 2645 terminal implements the memory lock feature by requiring that space be allocated for new lines (such as the ones needed when you enter a new subroutine) at the BOTTOM of the screen. This means that some of the most recently typed lines will disappear from your screen. To prevent this, erase the screen yourself (home cursor, clear display keys) whenever there is nothing of interest on it, and you anticipate a large screen display.

Memory Locking a Program to be Debugged

Why must a program being debugged be memory locked? Every time you interact with Debug, (e.g., entering a command, or arriving at a breakpoint in your code), Debug has to access memory locations in your program. Therefore, the program must be in memory and not swapped out. To assure that this is always true, Debug memory locks the program being debugged.

Break at a Variable or Display a Line Number

Why is it possible to break at a variable or display a line number? This feature is used primarily by Macro users. In Macro, local variables or line numbers can contain either data or machine instructions. Debug cannot tell the difference. The ability to manipulate a Location is necessary in Macro. This feature is also available in FORTRAN but not as useful as in Macro.

How Does Single Stepping Work?

Debug examines each of the instructions in the line of code to be single stepped and sets breakpoints at all of the possible memory locations where that line could return control to. It then allows the program to execute. When control is returned to your program, Debug clears those breakpoints.

Can't Single Step

When Debug is unable to single step, it displays "Can't single step". This message is displayed typically when you are single stepping over RTE library routines that go privileged. Debug cannot follow programs that turn off interrupts since it relies on the RTE operating system to interact with your program. This also may happen when you are single stepping over subroutines that do not use the standard (.ENTR) calling sequence. The following section further discusses single stepping problems.

Variation in Single Stepping Speed

When Debug performs single step operations at various times, there is a difference in speed. This happens typically when you are single stepping over subroutines that are in RTE libraries that do not use the standard RTE calling sequence. In these cases, Debug does not know the return addresses for these subroutines, and must perform an emulation of their code to find out. The result is a reduction in speed.

How are Breakpoints Implemented?

When "B 123" is entered, Debug calculates the address for line 123 in your code space and copies the instruction that is currently there into one of the Debug buffers. It then replaces that instruction with a HLT instruction that causes a memory protect violation. When your program executes the HLT, RTE checks a debug bit in the program's ID segment. If the bit is set, control is returned to Debug. For segmented programs, breakpoints in other segments are saved and re-installed when that particular segment is brought into memory.

Privileged Mode Limitations

Single stepping and setting breakpoints in privileged subroutines are not allowed. Interaction with the test program requires the services of the RTE operating system. EXEC calls from Debug schedule the test program and RTE returns control back to Debug when a breakpoint is encountered. Privileged code does not allow RTE to intervene when breakpoints are encountered. Since breakpoints are implemented as HLT instructions, they will halt the CPU in the privileged mode. Therefore, there is no single stepping or break in the privileged mode.

Erratic Returns While Single-Stepping in FORTRAN 77

Single-stepping in FORTRAN 77 sometimes results in erratic returns. This is because the FORTRAN 77 compiler optimizes some RETURN statements, making them GOTO statements. In these cases, you have to enter another single stepping command to leave the subroutine.

Debugging Interactive Programs

If your program does much I/O with the terminal or uses many of the terminal special features, Debug may interfere with program execution. There is a special Debug command that allows you to debug from a terminal other than the one where your program is running. This command is "+L:n" which is entered in the Debug runstring. It must be the first parameter in the runstring. Parameter n must be a terminal LU number. For example:

```
     :RU,DEBUG,+L:5,MYPRG,P1,P2,P3
```

The runstring parameter "+L:5" directs Debug to send its terminal outputs to and receive its terminal inputs from LU 5.

Debugging a Program Scheduled by Another Program

This section describes how to debug a program scheduled with wait (using EXEC 23) by another program.

As an example, a program called MYSON is to be scheduled with an EXEC call with parameters P1, P2, and P3. Normally, the call is:

```
      CALL EXEC (23,5hMYSON,P1,P2,P3)
```

To debug program MYSON, change the call to:

```
      CALL EXEC (23,5hDEBUG,P1,P2,P3,0,0,+14hRU,DEBUG,MYSON,-14)
```

In this case, Debug must be restored (RP'd) into memory. Debug ensures that any RMPAR parameters are passed from MYSON to the calling program. Also, Debug ensures that the values of modified runstrings are returned.

This method of debugging a scheduled program always requires that you interact with Debug before encountering the scheduled program, even though the calling program is not being debugged.

It may be desirable to run MYSON under Debug only if the calling program is also being debugged. In this way, MYSON does not have to be recompiled and linked just to change the EXEC call which schedules Debug or MYSON.

The RTE library utility BNGDB can be used to determine if the calling program is being debugged. This utility returns a logical result, true if the calling program is being debugged, and false if not. For example:

```
     :
   Logical BNGDB
     :
     :
   If (BNGDB()) then
       Call EXEC (23,5hMYSON,P1,P2,P3)
   else
       Call EXEC (23,5hDEBUG,P1,P2,P3,0,0,+14hRU,DEBUG,MYSON,-14)
   endif
     :
```

Using the example code shown above, you can debug MYSON only if the calling program is also being debugged.

Appendix A
Error Messages

This appendix provides a listing of Debug error messages in alphabetical order of the first word. Some error messages are self explanatory. There are no explanations for these messages.

Address out of range

The address specified is not in the range used by the program. May occur when the program beging debugged is not in memory.

Arrays not legal here

Bad conditional breakpoint or tracepoint:

The breakpoint or tracepoint is not correct or the variables specified cannot be found.

Bad line number

The line specified is not appropriate for this command. The line number must be within the current subroutine or must be qualified by specifying a subroutine name or segment name. It must be an executable line.

Bad Location

The address specified is not in the range used by the program.

Bad value

The value specified is out of range for this command.

Caller can't be found (wasn't compiled with debug option)

The Where command did not find any debug information for the caller of the subroutine. Either the caller was not compiled with the debug option or the return address is incorrect.

Caller not available

The Where command could not determine the call of this subroutine because of multiple entry points. Debug cannot determine which entry point was used.

Can only access one log file at a time

Can only profile 1000 lines of this subroutine

Can't display or modify VMA addresses

VMA is not supported by Debug. Use EMA instead.

Can't find starting line because of multiple entries

Can't single step. Please use breakpoints.

Debug is unable to single step this line. This may occur when there is privileged code in one of the subroutines called by this line or special micro-code has been encountered. Set a breakpoint at the next location. To get the program past this point, use a P command to get past this line.

Display count must be between 1 and 127

Duplicate entry points

There are duplicate entry points or module names found in the symbol table file for this program (file produced by LINK). Change the source to have unique names, recompile and relink.

Expecting =, <>, <=, >=, > or <

File name expected

fmp error

Illegal array reference

Illegal character

Illegal character in type override field

The legal characters are A, B, C, D, I, J, L, O, R, X and Z.

Illegal combination of EMA and byte addressing

Byte addressing (use of the ' character) cannot be used with an EMA address

Illegal use of byte address

Byte addresses (use of the ' character) cannot be used with indirection (the @ character).

Illegal use of EMA address

A breakpoint cannot be set at an EMA address nor can an EMA address be listed.

Integer expected

Single integer is expected.

Legal keywords are Into and Trace

Only 'Step Into' and 'Step Trace' are legal options for the step command

Legal register references are (A), (B), (X), (Y), (E), or (O)

Location must be a subroutine name

Location must be in current segment or main

Only variables in the main or currently loaded segment can be displayed or modified.

No breakpoint here

There is no breakpoint at the location specified to the Clear command. Specify a location with a breakpoint or use C * to clear all breakpoints.

No debug info for this module

Recompile the module using the Symbolic debug option and relink.

No INCLUDE nesting

An Include command cannot be given from an include file.

No more than 7 dimensions allowed on arrays

Not enough subscripts. Filling in lower bounds for those not given

More subscripts were declared for this array than were supplied. Debug will use the lower bound for those not given.

Only handle 264X and 262X terminals

Only 50 breakpoints allowed

This includes the breakpoints Debug automatically sets while single stepping.

Only 20 levels of nested procedure definitions is allowed

Only = and <> legal with this type

Only comparisons for equal and not equal are allowed for strings and complex data types.

Program ran to completion

Debug will terminate itself when this condition occurs.

Program was aborted

The program was aborted by an OFF command. Debug will terminate.

Schedule error

The progam given in the runstring or run command could not be scheduled.

Segment DEBU0 missing

Debug is not loaded correctly. Reload using Link.

Segment DEBU1 missing

Reload Debug using LINK.

Segment DEBU2 missing

Reload Debug using LINK.

Segment DEBU3 missing

Reload Debug using LINK.

Segment DEBU4 missing

Reload Debug using LINK.

Segmenting/Debug internal errors detected

Report the problem to your HP representative.

Subscript must be an integer

Only single integer array subscripts are supported.

Symbol not found

Symbol table overflow

Too many subroutines were compiled with the Debug option. Try recompiling some routines without the Debug option.

This command is illegal in overview mode

The legal commands in overview mode are: Histogram, List, Include, View and Exit

This command is legal only in overview mode

The Histogram command is legal only after the Overview command is given.

Timing error. Do you have a quiet system?

The Overview command found the program in an unexpected state. Usually occurs when other programs are running while the overview data is being collected.

Too many segments

The program has more segments than Debug can handle. Current limit is 60.

Too many subscripts given. Ignoring extra ones

More subscripts were specified then were declared for this array. Debug ignores the extra ones.

Unknown command

The first character does not match any of the Debug commands. This applies to the first two characters as well.

Unknown data type

The specified type is not known to Debug.

Value doesn't match variable's type

The value given is different from the type of the variable on the left.

Value too big

Value too small

Variable must be in same segment as breakpoint

Conditional breakpoint and tracepoints require that the variable accessed be in the main or the same segment as the breakpoint or tracepoint.

2nd address before 1st

Index

READER COMMENT SHEET

Symbolic Debug/1000
User's Manual

92860-90001 May 1983

Update No. ________
(If Applicable)

We welcome your evaluation of this manual. Your comments and suggestions help us improve our publications. Please use additional pages if necessary.

FROM:

Name	______________________
Company	______________________
Address	______________________
	______________________
Phone No. ______________	**Ext.** ________

FOLD FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 141 CUPERTINO, CA.

— POSTAGE WILL BE PAID BY —

Hewlett-Packard Company
Computer Languages Lab
11000 Wolfe Road
Cupertino, California 95014
ATTN: Technical Publications

FOLD FOLD

HEWLETT
PACKARD

MANUAL PART NO. 92860-90001
Printed in U.S.A. May 1983
E0583

HEWLETT-PACKARD COMPANY
Computer Languages Lab
11000 Wolfe Road
Cupertino, California 95014