DOMAIN Language Level Debugger Reference

Order No. 001525
Revision 04

Apollo Computer Inc.
330 Billerica Road
Chelmsford, MA 01824

First Printing: April, 1982
Latest Revision: January, 1987

This document was produced using the Interleaf Workstation Publishing Software (WPS). Interleaf and WPS are trademarks of Interleaf, Inc.

Preface

This manual describes the language level debugger, DEBUG. We've organized this manual as follows:

Chapter 1	Provides a brief overview and tutorial of DEBUG.
Chapter 2	Explains how to compile your source code so that it can be debugged, and how to invoke the debugger.
Chapter 3	Details all DEBUG commands.
Chapter 4	Explains some language dependencies. For example, what should a C programmer know when using DEBUG?
Appendix A	Supplies some helpful hints in Question and Answer format.
Appendix B	Explains how to debug an installed library.
Appendix C	Describes how optimization affects debugging.

Related Manuals

Use the following language manuals in conjunction with the debugger manual:

- *DOMAIN Pascal Language Reference* (000792)
- *DOMAIN FORTRAN Language Reference* (000530)
- *DOMAIN C Language Reference* (002093)

Problems, Questions, and Suggestions

We appreciate comments from the people who use our system. In order to make it easy for you to communicate with us, we provide the User Change Request (UCR) system for software-related comments, and the Reader's Response form for documentation comments. By using these formal channels you make it easy for us to respond to your comments.

You can get more information about how to submit a UCR by consulting the *DOMAIN System Command Reference*. Refer to the CRUCR (CREATE_USER_CHANGE_REQUEST) Shell command description. You can view the same description on-line by typing:

```
$ help crucr <RETURN>
```

For your documentation comments, we've included a Reader's Response form at the back of each manual.

Documentation Conventions

Unless otherwise noted in the text, this manual uses the following symbolic conventions:

boldface	Bold, uppercase words or characters in formats and command descriptions represent commands or keywords that you must use literally. Letters in uppercase

	boldface must be used, but letters in lowercase boldface are optional. For instance, consider **SIGnal**. Since the word is boldfaced, it is mandatory. The arrangement of uppercase and lowercase letters indicates that the word can be abbreviated to SIG.
nonboldface	Words that are neither boldfaced, nor italicized indicate a part of the expression that you must supply, but you do not supply it literally. For instance, consider pathname. You would not enter the word "pathname," you would enter a pathname instead.
italicized	Italicized words are optional arguments.
`output`	Typewriter font words in command examples represent literal system output.
`color`	Colored words indicate user input.
(comments)	In examples, comments are italicized and enclosed in parentheses.
< >	Angle brackets enclose the name of a key on the keyboard.
CTRL/Z	The notation CTRL/ followed by the name of a key indicates a control character sequence. You should hold down <CTRL> while pressing the key.
...	Horizontal ellipsis points indicate that the preceding item can be repeated one or more times.
. . .	Vertical ellipsis points mean that irrelevant parts of a figure or example have been omitted.

Summary of Technical Changes

We last revised this manual for SR9.0. Since then, we've reorganized the manual and made it look prettier. DEBUG software has changed in the following ways:

- -SRC is now on by default. The -NSRC option must be used to suppress source display.
- You no longer have to supply fully-qualified routine names or variable names (e.g., FOO\BAR\WALDO) when the routine or variable is visible from the current environment.
- You can now refer to line numbers in other routines in the current source file without prefixing the routine name.
- DEBUG marks breakpoint locations in the source display with a "!".
- New options for the ENVIRONMENT command allow you to walk up and down the call stack, and restore a previously defined user environment.
- A new SOURCE command allows you to directly specify the source file to be displayed. Please use this new SOURCE command instead of the old SDIR command.
- By using the new SOURCE command, you can display the source code of programs that were compiled with the -DB option.
- The DELETE -BREAKPOINT command accepts two new sub-options: -VA for deleting a breakpoint at a specific address, and -HERE for deleting the current breakpoint.
- If a program stops in Apollo library code. DEBUG no longer automatically sets the user environment to the last-called user routine. (The new -CALLER option of the ENVIRONMENT command makes it easy to walk back manually.)

- A new debugger variable named 'MAX_QUAL limits the number of qualifiers (which are enclosing routine–names) that prefix displayed routine and variable names.
- DEBUG displays unprintable characters in ASCII data as '<^x>' or '<xx>'. A new debugger variable named 'MAX_BAD_CHARS limits the number of such characters which are output in a string.
- Character string literals in commands may be delimited with either single or double quotes. Single and double quotes are interchangeable (but must be used in matching pairs).
- You can now set breakpoints on FORTRAN statement functions.
- DEBUG now correctly accesses FORTRAN arrays that have variable dimensions.
- By default, DEBUG displays C "char" variables (8–bit integers) in ASCII format and allows you to set them to character literals. You can set Pascal "char" variables to integer values.
- You can subscript all C pointers. (Prior to SR9.5, you could not subscript struct fields and array elements.) The "all" subscript [*] is not valid in general because the size of the implied array is unknown. But it can be used with pointers to chars for which the usual null–terminated string convention applies.
- You can now de–reference pointers to procedures and functions. The result is a character string containing the name of the routine.
- The SHELL command now accepts a shell command string as an argument.
- The –PROC command line option now accepts a process uid or a Unix pid as its argument.
- The source code position arrow is now turned off while the target program is running.
- You can now debug installed libraries, though some rather severe restrictions apply. A new –GLOB command line option lets you step into code in global address space.
- A new SIGNAL command simulates a fault at the current point of execution.
- Action lists for faults have been added. If you define a macro named 'FAULT_ACTION, DEBUG executes it when the program faults. The debugger variable 'FAULT_STATUS contains the fault status code.
- There is a new –VERBOSE option for HELP.
- DEBUG can now distinguish between unsigned and signed integer variables.
- We've fixed many bugs.

We've used changed bars to mark technical changes to DEBUG since SR9.0.

Contents

Illustrations

Tables

Chapter 1

Introduction to DEBUG

The DOMAIN Language Level Debugger (DEBUG) is a high-level language debugger. You use it to debug FORTRAN, Pascal, and C programs running on the AEGIS or the DOMAIN/IX operating systems. DEBUG supports the features found in most high-level language debuggers; that is, it lets you set breakpoints, jump through the program, and examine variables. But DEBUG also supports many other interesting features. Using DEBUG, you can

- Control program flow
  - Set breakpoints to suspend program execution at any statement.
  - Step the program one source statement at a time.
  - Change the order of execution.
  - Intercept or simulate program faults.
- Examine program status
  - Display, set, or change the value of a variable.
  - Describe the data type and storage allocation of a variable.
  - Trace the chain of calls that brought the program to its current state.
- Display program source code
  - Display the source code of the program in a separate window, with an automatically updated indication of the current point of execution.
  - Have full Display Manager access to the source file display for scrolling, text searching, etc.
- "Program" DEBUG
  - Define action-lists of commands to be automatically executed at breakpoints or program faults.
  - Define macros to abbreviate common sequences of commands.
  - Define startup files of commands to be automatically executed when DEBUG starts.
  - Specify conditional execution of commands.

1.1 The Debugging Process

The debugging process can be divided into the following three steps:

1. Compile the program so that DEBUG can use it. (For details, see Chapter 2.)
2. Invoke DEBUG. (Also see Chapter 2.)
3. Use DEBUG after you've invoked it. (For details, see Chapter 3.)

To accomplish step 1, compile with the -DB, -DBA, or -DBS option. For example:

```
$ ftn test.ftn -dba          $ pas test.pas -dba          $ cc test.c -dba
```

The simplest way to accomplish step 2 is to issue the command DEBUG followed by the name of the object file you want to debug, for example:

```
$ debug test.bin
```

Accomplishing step 3 is more complicated, so we provide the following short tutorial.

1.2 A Short Debugging Tutorial

The goal of a debugging session is to eliminate the errors in your source code. The debugger is a tool that helps you find errors by allowing you to selectively examine a program's variables while the program is running. Once you find the errors, you must edit the source code and recompile it.

A feature of all debuggers is that they let you set **breakpoints** through the program. A breakpoint is an order to temporarily halt program execution at a particular line. For instance, if you set a breakpoint at line 20, the program will run normally until it reaches the statement at line 20, and then it will halt. In DEBUG, you set breakpoints with a command called, conveniently enough, **BREAKPOINT**. While halted at a breakpoint, you can examine the values of variables or perform any other debugger function. In DEBUG, you use the **EXAMINE, PRINT,** or **ARGS** command to display the value of variables. You can set an unlimited number of breakpoints in the program. Debuggers also support a command that resumes program execution from the breakpoint; in DEBUG, this command is called **GO**.

The **STEP** command is an alternative to the combination of **BREAKPOINT** and **GO** commands. You use the **STEP** command to execute the program one statement at a time.

So, there you have it in a nutshell. You use the debugger to flip from breakpoint to breakpoint in order to discover where your program went wrong. Keep in mind that you cannot use DEBUG to change erroneous source code. You can only use DEBUG to detect the location of erroneous source code. When you find the error, you still have to go back and change your source code.

1.2.1 Three Sample Debugging Sessions

To help you get started, we provide Figures 1–1, 1–2, and 1–3, which show sample debugging sessions with FORTRAN, C, and Pascal programs. Notice how the debugging window in these figures is divided into the following three window panes:

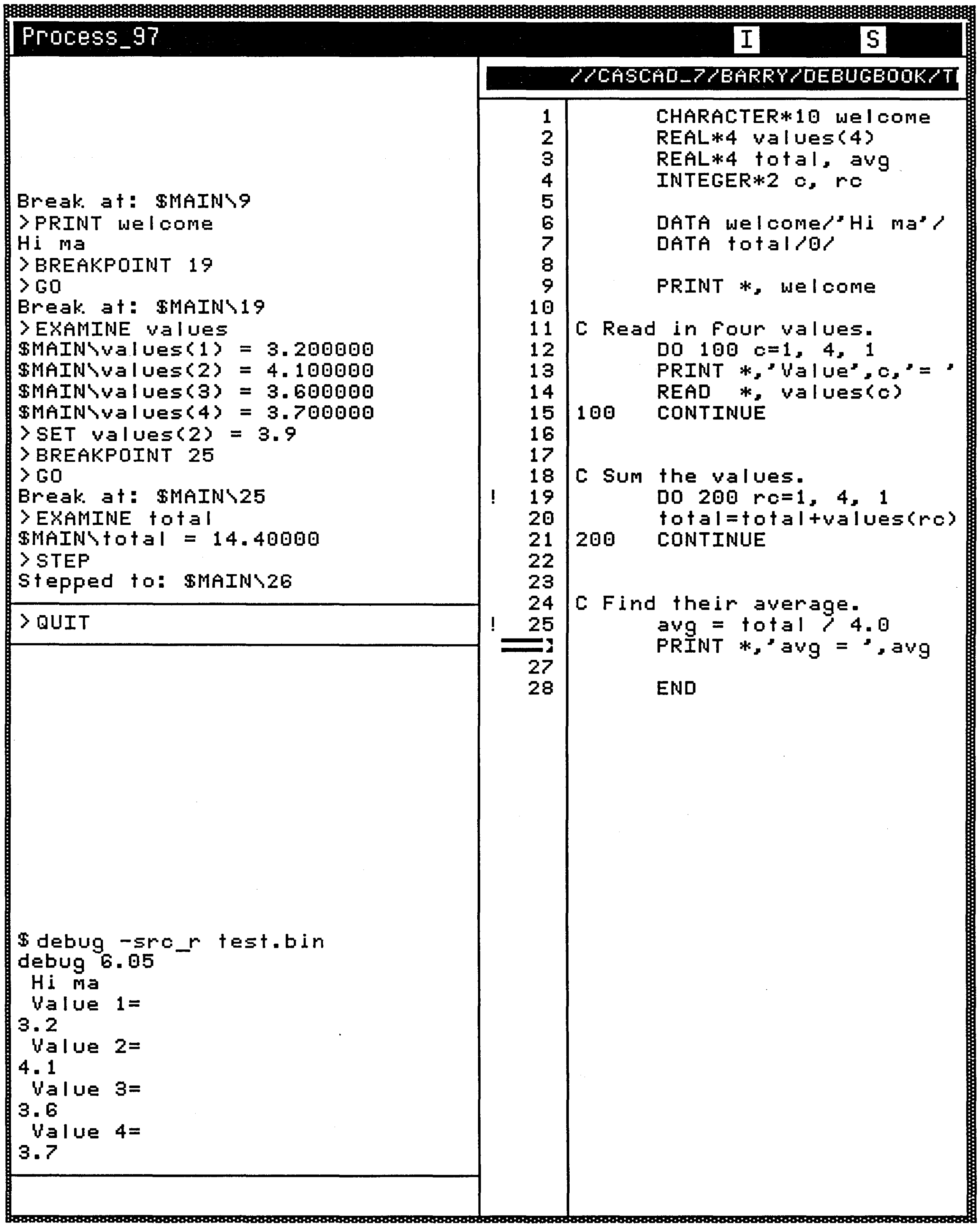
- The right window pane is a listing of the source code that we are debugging.
- The bottom left window pane is a transcript pad showing all input and output of the running program.
- The top left window pane is where we enter all our DEBUG commands, and DEBUG displays debugger output.

Let's now examine the top left window pane in greater detail. Here's the order of the debugging commands:

```
PRINT        Displays the value of the character array.
BREAKPOINT   Sets a breakpoint at the given line number.
```

GO	Advances the program from the start to the first breakpoint.
EXAMINE	Displays all four elements of the array.
SET	Changes the value of the second element of the array.
BREAKPOINT	Sets a second breakpoint at the given line number.
GO	Advances the program from the first breakpoint to the second.
EXAMINE	Displays the value of a simple variable.
STEP	Advances the program to the next statement.
QUIT	Ends the debugging session.

Look at the source display in the window pane on the right. Notice how each line of code is preceded by a line number. These line numbers are very useful for setting breakpoints. DEBUG places an exclamation point next to a line number where a breakpoint has been set. If you delete the breakpoint (with the **DELETE** command), the exclamation point disappears. Find the arrow (it's on line 26, 23, or 29 in the three figures). The arrow tells you where the location of the program counter; in other words, it marks your current position in the program.



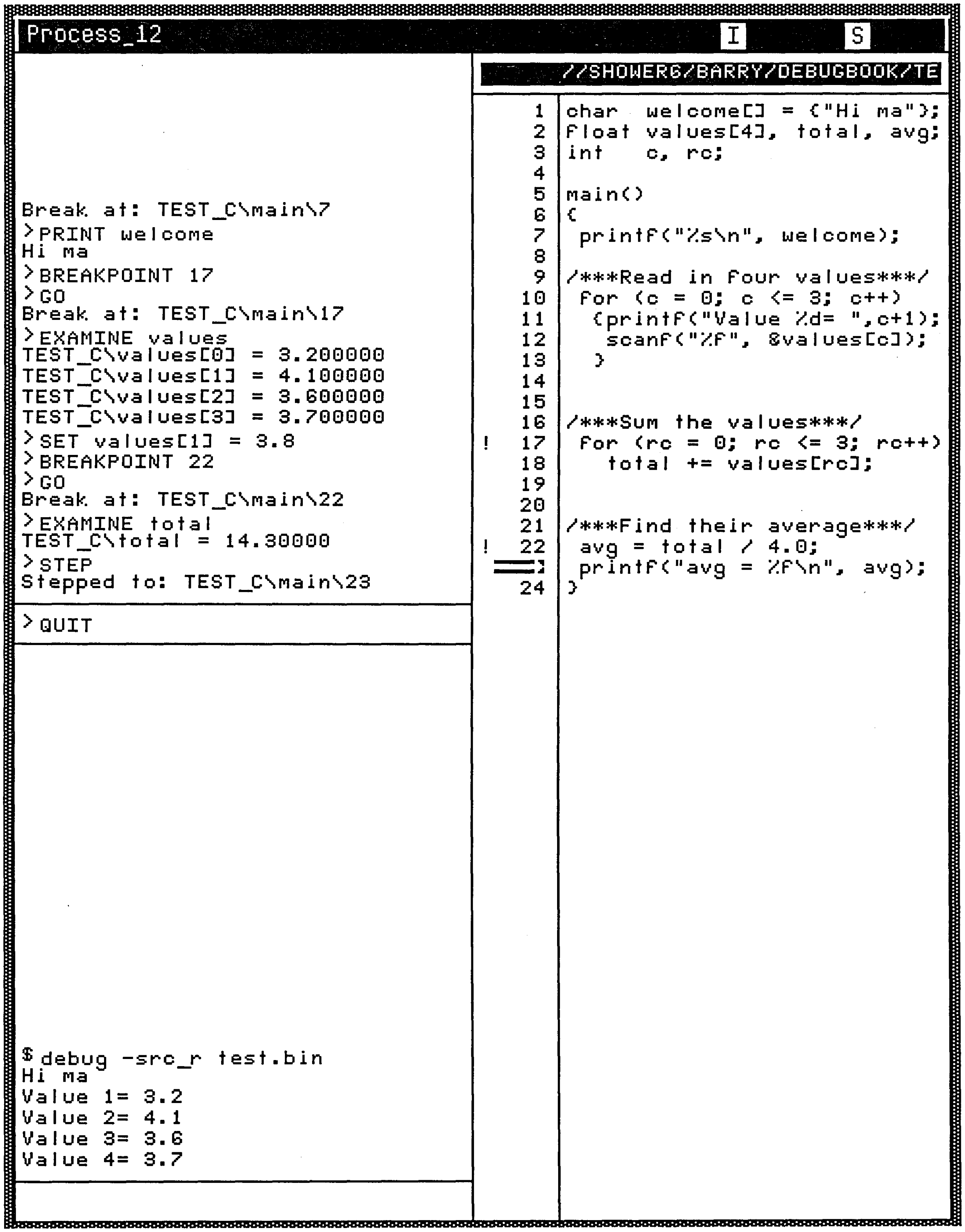
```
Break at: $MAIN\9
>PRINT welcome
Hi ma
>BREAKPOINT 19
>GO
Break at: $MAIN\19
>EXAMINE values
$MAIN\values(1) = 3.200000
$MAIN\values(2) = 4.100000
$MAIN\values(3) = 3.600000
$MAIN\values(4) = 3.700000
>SET values(2) = 3.9
>BREAKPOINT 25
>GO
Break at: $MAIN\25
>EXAMINE total
$MAIN\total = 14.40000
>STEP
Stepped to: $MAIN\26
```

```
>QUIT
```

```
$debug -src_r test.bin
debug 6.05
 Hi ma
 Value 1=
3.2
 Value 2=
4.1
 Value 3=
3.6
 Value 4=
3.7
```

```
     1         CHARACTER*10 welcome
     2         REAL*4 values(4)
     3         REAL*4 total, avg
     4         INTEGER*2 c, rc
     5
     6         DATA welcome/'Hi ma'/
     7         DATA total/0/
     8
     9         PRINT *, welcome
    10
    11   C Read in four values.
    12         DO 100 c=1, 4, 1
    13         PRINT *,'Value',c,'= '
    14         READ  *, values(c)
    15   100   CONTINUE
    16
    17
    18   C Sum the values.
!   19         DO 200 rc=1, 4, 1
    20         total=total+values(rc)
    21   200   CONTINUE
    22
    23
    24   C Find their average.
!   25         avg = total / 4.0
  ═══▶         PRINT *,'avg = ',avg
    27
    28         END
```

Figure 1-1. A DEBUG Session With a Simple FORTRAN Program



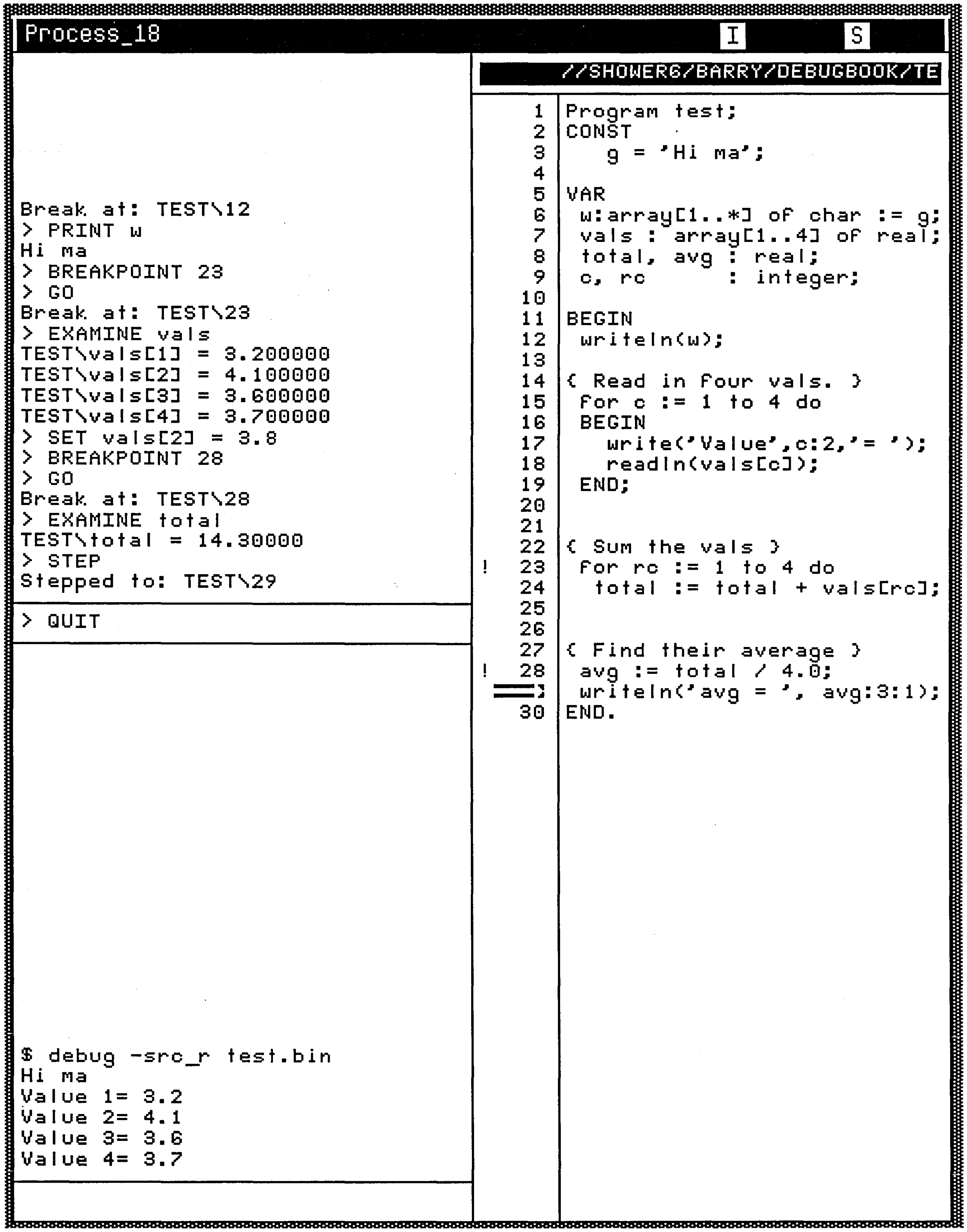
```
Break at: TEST_C\main\7
>PRINT welcome
Hi ma
>BREAKPOINT 17
>GO
Break at: TEST_C\main\17
>EXAMINE values
TEST_C\values[0] = 3.200000
TEST_C\values[1] = 4.100000
TEST_C\values[2] = 3.600000
TEST_C\values[3] = 3.700000
>SET values[1] = 3.8
>BREAKPOINT 22
>GO
Break at: TEST_C\main\22
>EXAMINE total
TEST_C\total = 14.30000
>STEP
Stepped to: TEST_C\main\23
```

```
>QUIT
```

```
$debug -src_r test.bin
Hi ma
Value 1= 3.2
Value 2= 4.1
Value 3= 3.6
Value 4= 3.7
```

```
     1  char  welcome[] = {"Hi ma"};
     2  float values[4], total, avg;
     3  int   c, rc;
     4
     5  main()
     6  {
     7   printf("%s\n", welcome);
     8
     9  /***Read in four values***/
    10   for (c = 0; c <= 3; c++)
    11    {printf("Value %d= ",c+1);
    12     scanf("%f", &values[c]);
    13    }
    14
    15
    16  /***Sum the values***/
!   17   for (rc = 0; rc <= 3; rc++)
    18      total += values[rc];
    19
    20
    21  /***Find their average***/
!   22   avg = total / 4.0;
  ===:   printf("avg = %f\n", avg);
    24  }
```

Figure 1–2. A DEBUG Session With a Simple C Program



```
Break at: TEST\12
> PRINT w
Hi ma
> BREAKPOINT 23
> GO
Break at: TEST\23
> EXAMINE vals
TEST\vals[1] = 3.200000
TEST\vals[2] = 4.100000
TEST\vals[3] = 3.600000
TEST\vals[4] = 3.700000
> SET vals[2] = 3.8
> BREAKPOINT 28
> GO
Break at: TEST\28
> EXAMINE total
TEST\total = 14.30000
> STEP
Stepped to: TEST\29
```

```
> QUIT
```

```
$ debug -src_r test.bin
Hi ma
Value 1= 3.2
Value 2= 4.1
Value 3= 3.6
Value 4= 3.7
```

```
     1  Program test;
     2  CONST
     3     g = 'Hi ma';
     4
     5  VAR
     6   w:array[1..*] of char := g;
     7   vals : array[1..4] of real;
     8   total, avg : real;
     9   c, rc      : integer;
    10
    11  BEGIN
    12   writeln(w);
    13
    14  { Read in four vals. }
    15   for c := 1 to 4 do
    16   BEGIN
    17     write('Value',c:2,'= ');
    18     readln(vals[c]);
    19   END;
    20
    21
    22  { Sum the vals }
!   23   for rc := 1 to 4 do
    24    total := total + vals[rc];
    25
    26
    27  { Find their average }
!   28   avg := total / 4.0;
  ==>    writeln('avg = ', avg:3:1);
    30  END.
```

Figure 1–3. A DEBUG Session With a Simple Pascal Program

Chapter 2

Invoking DEBUG

This chapter explains how to prepare a program so that it can be debugged, and how to invoke the debugger.

2.1 Preparing Programs For DEBUG

In order to use DEBUG to debug a program, you must first compile your source code with the correct compiler option. Each compiler (CC, PAS, and FTN) supports four command options that affect DEBUG's access to a program. Table 2–1 lists these options and summarizes the DEBUG access each option provides. A detailed description of each option follows the table.

Table 2–1. Compiler Options Affecting Debugging

Effects \ Options	–DB (default)	–DBA	–DBS	–NDB**
You can access the names of routines.	YES	YES	YES	NO
You can access the values or descriptions of variables.	NO	YES	YES	NO
You can access line numbers and set breakpoints.	YES	YES	YES	NO
DEBUG automatically displays the source code you are debugging.	NO*	YES	YES	NO
The compiler removes any optimizations that might interfere with debugging.	NO	YES	NO	NO
The compiler performs the level of optimization specified by the –OPT option.	YES	NO	YES	YES

* DEBUG only displays the source code that you request.
** You cannot debug a file compiled with –NDB.

2.1.1 The –DB Compiler Option

When you compile a program with the –DB option (the default), the compiler performs its normal optimizations and creates a line number table, but it does not create a symbol table.

The line number table permits you to set breakpoints or run a traceback. The line number table provides DEBUG with the names of the routines in the module, but does not provide information on the nesting structure. Therefore, if two routines have the same name, DEBUG cannot distinguish between them.

Because it does not create a symbol table, the –DB option provides no access to variables, constants, and labels. Therefore, the –DB option is usually of little value when you intend to use DEBUG.

This option does permit source display debugging (described in the "Source Code Display" listing of Chapter 3); however, DEBUG will only display the source code that you explicitly request to see. (The –DBS and –DBA options provide DEBUG with enough information to automatically display the proper source code.)

2.1.2 The –DBS and –DBA Compiler Options

For most debugging, you will want to compile with either the –DBS or –DBA option. As Table 2–1 shows, both options provide the exact same access to the debugger, the only difference is in how the compiler optimizes the code.

If you specify –DBA, the compiler eliminates all optimizations that could interfere with debugging. If you specify –DBS, the compiler allows any optimizations. These optimizations could interfere with debugging since the source code may not correspond exactly to the object code. On the other hand, –DBS ensures that the code you debug is the same as production code compiled with –DB or –NDB thereby increasing your confidence in the correctness of the final program.

–DBA makes debugging easier by eliminating compiler optimizations that interfere with a direct correspondence between the source and object programs. Which is better? It is hard to generalize, but here are two helpful hints:

- The difficulty of debugging a program compiled with –DBS depends greatly on the program itself. You may wish to start with –DBS and switch to –DBA only if optimization–related programs are too great.
- Optimization sometimes exposes latent program bugs. Debugging a program compiled with –DBA should not replace thorough testing of the optimized program.

Appendix C details the ways in which optimization can affect debugging.

2.1.3 The –NDB Compiler Option

The –NDB option causes the compiler to create as compact an object module as possible. The resulting object module will have no line number table or symbol table. DEBUG has no access to programs compiled with –NDB. In fact, –NDB also interferes with the TB, HPC, and DPAT utilities. Therefore, we recommend that you not use –NDB unless it is absolutely essential to reduce the program's size. Note that the extra DEBUG information added by –DB, –DBA, or –DBS does not affect the execution time of the program.

2.2 Invoking DEBUG

To invoke DEBUG, issue a command line that has one of the following two formats:

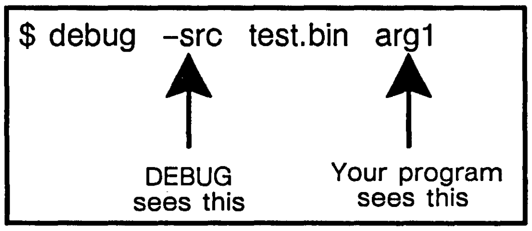
$ debug *debug_options* target_program *target_program_arguments*

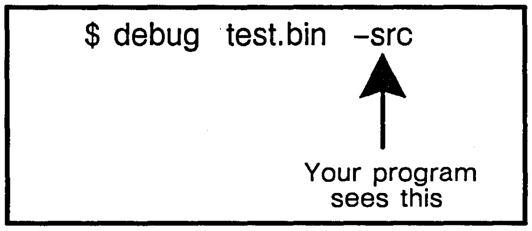
or

$ debug *debug_options* **-proc** process_name

Debug_options are one or more options described in the next section. The **target_program** is the object file that you want to debug. **Target_program_arguments** are any arguments you want to supply to the **target_program.**

You must specify all **debug_options** *before* you specify the target program name. DEBUG only scans for options up to the target program name. For example, consider how the system interprets the following two different command lines:





By default, when you invoke DEBUG (without specifying the -proc option), the following happens:

1. The system forks, creating a new process. DEBUG runs in the child process, which is named "DEBUG" (or "DEBUG.1" or "DEBUG.2", etc.).
2. DEBUG loads the target program into the original (parent) process. During program loading, DEBUG copies all read-only portions to read/write storage, so that you can set breakpoints.
3. DEBUG splits the window into three window panes -- one window pane displays the source code you are debugging, another window pane displays all DEBUG commands you make, and a third window pane holds all I/O of the program you are debugging.
4. DEBUG sets a breakpoint on the first executable statement in the target program.
5. DEBUG looks in your home directory for the pathname `user_data/startup_debug`. If DEBUG finds this pathname, DEBUG processes the file's contents as a sequence of DEBUG commands. If DEBUG does not find this pathname, no error occurs. DEBUG then looks in your working directory for filename `startup_debug`. If DEBUG finds this filename, DEBUG processes the file's contents as a sequence of DEBUG commands. If DEBUG does not find this filename, no error occurs.
6. DEBUG starts execution of the target program. The target program runs until just before the first executable user statement, where DEBUG sets a breakpoint. At that point, DEBUG issues its prompt character (>) and waits for a command.

NOTE: You cannot redirect or pipe the I/O of the target program. For example, the command line

```
$ DEBUG foo.bin >foo.output
```

redirects DEBUG's standard output stream, rather than `foo`'s because I/O redirection is a Shell function. If you need to redirect or pipe target program I/O, use the -PROC option.

2.2.1 DEBUG Options

The DEBUG command line options allow you to customize the operation and appearance of your debugging session. Table 2–2 summarizes these options and gives a brief description of each option's function.

Table 2–2. DEBUG Command Line Options

Option	Function
–nc	Prevents DEBUG from copying the target object file. DEBUG maps the object file so that you can write the breakpoint directly into the object file.
–proc proc_name	Enables DEBUG to debug the next program executed in a specified process.
–read pathname	Invokes a DEBUG command file with the specified pathname.
–set argument(s)	Sets one or more DEBUG variables at invocation.
–wp10 ... –wp90	Sets the size of the DEBUG window pane from 10% to 90% of the process transcript window.
–nwp	Causes DEBUG to use the same windows as your target program (no separate DEBUG windowpane). DEBUG's input and output mixes with your program's input and output.
–src	Creates a "source–display" window paneas well as a DEBUG windowpane and displays the source file as you debug your program. This option is on by default.
–nsrc	Suppresses creation of a source–display window pane.
–src_t	Forces DEBUG to put the source–display window pane at the top of the DEBUG window pane.
–src_r	Forces DEBUG to put the source–display window pane on the right of the DEBUG windowpane.
–sdir pathname	Allows you to specify one or more alternate pathnames for source files.
–glob	Enables DEBUG to enter routines in global address space.
–smap	Prints a brief section map of the target program loading operation.

The remainder of this section describes these options in more detail.

2.2.2 The –NC Option

The –NC (No Copy) option prevents DEBUG from copying the target object file. Instead, DEBUG maps the object file so that you can write breakpoints directly into the object. This option is useful when you are debugging a large program. The –NC option saves time because the system requires less time to map the target for writing than it does to copy the entire procedure code.

NOTE: Take care when using the –NC option, because it is possible to make the object file invalid. Normally, when you exit from a debugging session, DEBUG repairs the breakpoint locations in the object file, leaving it exactly as it was before you invoked DEBUG. However, if you are debugging a program that is crashing itself and/or the process, DEBUG may never regain control before exiting. In this case, the object file will have breakpoint instructions in its procedure code that you cannot remove.

If DEBUG does leave breakpoints in the object file, it sets a flag so that you know that the object file is bad. If you try to use the object file again, you receive the following error message:

```
object module has unremovable breakpoints; rebind (process manager/loader)
```

When you use the -NC option, it is a good practice to keep an original copy of the object file and copy it to a test file before invoking DEBUG.

2.2.3 The -PROC Option

DEBUG always runs in a separate process from the target program. By default, DEBUG runs in a child of the process in which it is originally invoked, and the target program runs in the parent. You use the -PROC option to specify a different process for the target program to run in. The format for the option is:

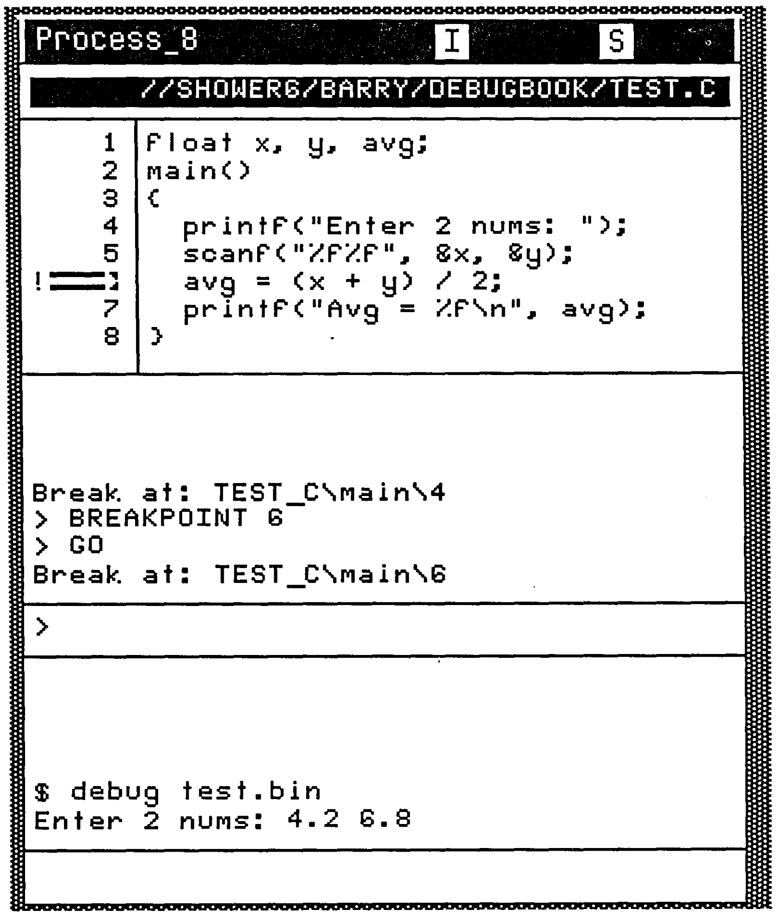
```
        process_name
-PROC   pid
        process_uid
```

You must specify exactly one of the three choices with -PROC.

When you use the -PROC option, you do not specify a target program; instead, DEBUG watches the specified target process (and its children) until it detects a new program being started (invoked or exec'ed). You are responsible for starting the target program in the named process. Usually you start the target program by invoking it on a command line, just as you would if you were not debugging it.

-PROC Example

If you do not specify the -PROC option, DEBUG and the target program run in the same window, as shown in Figure 2-1. In this example, DEBUG is a child process of Process_8.



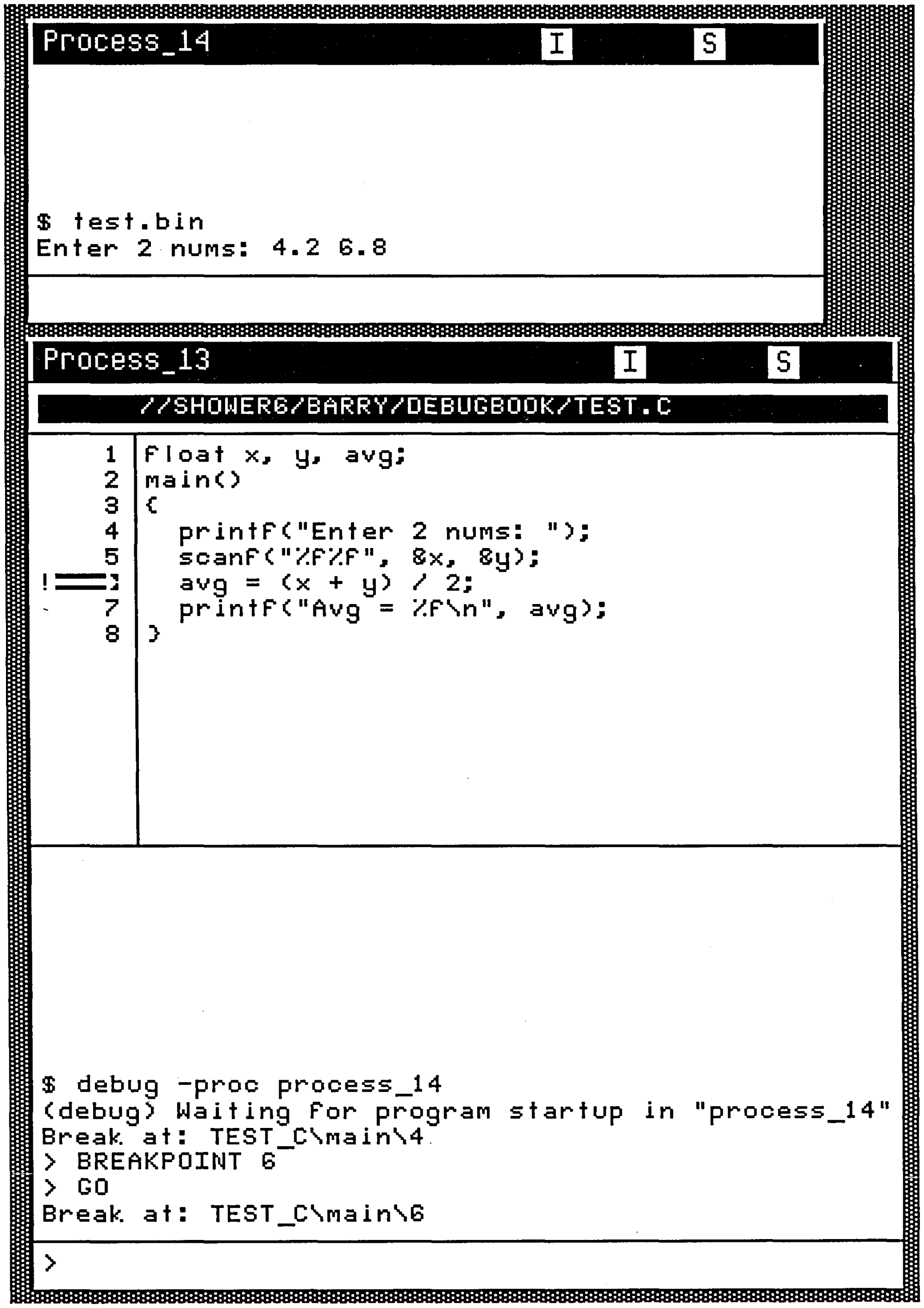
```
//SHOWER6/BARRY/DEBUGBOOK/TEST.C

    1 | float x, y, avg;
    2 | main()
    3 | {
    4 |   printf("Enter 2 nums: ");
    5 |   scanf("%f%f", &x, &y);
!     |   avg = (x + y) / 2;
    7 |   printf("Avg = %f\n", avg);
    8 | }
```

```
Break at: TEST_C\main\4
> BREAKPOINT 6
> GO
Break at: TEST_C\main\6
>
```

```
$ debug test.bin
Enter 2 nums: 4.2 6.8
```

Figure 2-1. Debugging Without the -PROC Switch

Sometimes, you may want to control the process in which the target and debugger run. Figure 2–2 shows how we used the –PROC option to run DEBUG in Process_13 and the target program in Process_14.



```
$ test.bin
Enter 2 nums: 4.2 6.8
```

```
    1  float x, y, avg;
    2  main()
    3  {
    4    printf("Enter 2 nums: ");
    5    scanf("%f%f", &x, &y);
 !===]   avg = (x + y) / 2;
    7    printf("Avg = %f\n", avg);
    8  }
```

```
$ debug -proc process_14
(debug) Waiting for program startup in "process_14"
Break at: TEST_C\main\4
> BREAKPOINT 6
> GO
Break at: TEST_C\main\6
>
```

Figure 2–2. Debugging With The –Proc Switch

Let's trace the steps we used to get the situation shown in Figure 2–2:

1. We created two shells. In our examples, the operating system named the shells Process_14 and Process_13. You'll likely get different names. We decided to run DEBUG in Process_13 and `test.bin` in Process_14.

2. We typed the following command in Process_13:

   ```
   $ debug -proc Process_14
   ```

   The system responded with the following prompt:

   ```
   (debug) Waiting for program startup in "Process_14"
   ```

3. We moved the cursor into Process_14 and entered the command:

   ```
   $ test.bin
   ```

4. We debugged `test.bin` by entering DEBUG commands from Process_13; program output appeared in Process_14.

Using –PROC to Debug Child Processes Spawned by a Program

Because DEBUG watches both the specified target process and its children for a program start, you can use DEBUG to debug child processes spawned by a program. The basic technique is the following:

1. Start the parent program, but don't start it under DEBUG.
2. After the program has started, but before it spawns the child process, start DEBUG specifying the parent process in a –PROC option. You may need to force the parent to pause in order to accomplish this.
3. Resume the parent process. When the child process is created and either invokes or execs a new program, DEBUG switches its attention to the child process and debugs the new program.

For programs that use a fork–exec sequence to spawn a child, you can use the following alternate technique:

1. Modify the code to force the child process to pause after the fork but before the exec.
2. When the child process pauses, start DEBUG on it.
3. Resume the child process. DEBUG will detect the exec as above.

Advantages of –PROC

Using the –PROC option has several advantages. The first advantage is that the –PROC option minimizes DEBUG's interference with the target process. The normal DEBUG invocation creates a new process in which to run the debugger (transparent to the user). However, at invocation, DEBUG momentarily runs in the same process as the target before continuing in the newly created process. This can alter memory contents and cause the target to run differently from a program running in a separate process, particularly if the program has unitialized variables or erroneously makes wild memory references. Therefore, if your program runs differently during a normally invoked DEBUG session than when you run it alone, try using the –PROC option.

The second advantage is that the –PROC option forces DEBUG to run in its own separate window. This enables you to use the Display Manager DQ (CTRL\Q) command unambiguously in either the debugger or the target window.

The third advantage is that the –PROC option gives the target program full use of a window.

2.2.4 The –READ Option

The –READ option causes DEBUG to read a specified file that contains DEBUG commands. The format for the option is:

-Read pathname

where pathname is the name of a file containing DEBUG commands. DEBUG executes the commands in the file immediately after it executes the commands in the default startup files (if they exist; see Chapter 1). The -READ option serves an identical purpose as the **READ** command (described in Chapter 3). By issuing a **READ** command as your very first command, you get the same results as if you had used the -READ option on your command line.

If the file contains any **GO** or **STEP** commands, DEBUG defers processing these commands until it has processed every other command. For example, in the following command file, the **EXAMINE** command executes before the **GO** command:

```
> SET `max_var_len = 66
> BREAKPOINT 262
> GO
> EXAMINE try
```

NOTE: Your DEBUG command line may contain no more than one -READ option. However, the file of DEBUG commands may itself contain **READ** commands.

2.2.5 The -SET Option

The -SET option sets the value of one or more debugger variables at invocation. For information on debugger variables, see the "Debugger Variables" listing in Chapter 3. Remember that debugger variables are somewhat different from program variables. After you invoke DEBUG, you can also set values with the **SET** command. (The option and the command are similar.) The option has three different formats:

FORMAT1:	**-SET** 'debugger_variable
FORMAT2:	**-SET** 'debugger_variable=initial_value
FORMAT3:	**-SET** " 'debugger_variable = initial_value "

In other words, specifying the name of the 'debugger_variable is mandatory, but specifying its value is optional. If you do not specify its value, DEBUG will prompt you for it. The only difference between FORMAT2 and FORMAT3 is blank space. If there are any blank spaces between the 'debugger_variable and the initial_value, then you must enclose both in quotation marks (either single or double). For example, compare the right and wrong ways to use -SET:

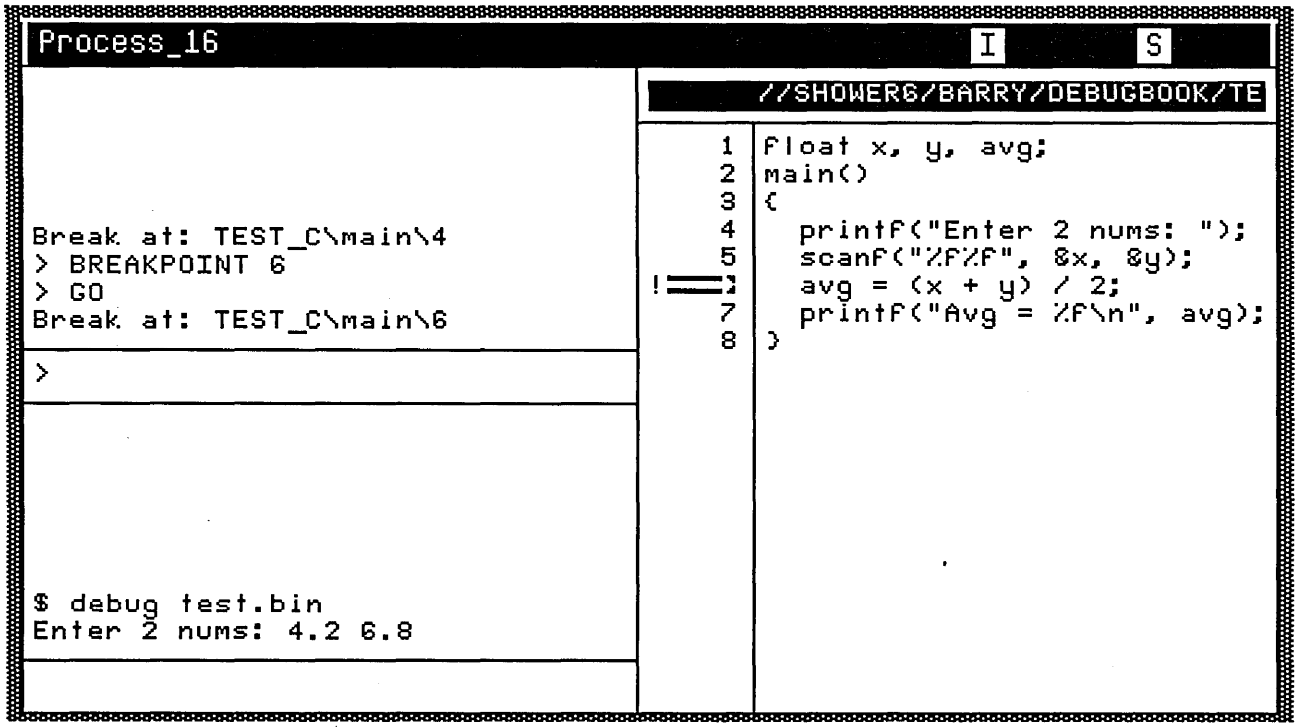
```
-SET `max_array_dim          (Right)
-SET `max_array_dim=8        (Right)
-SET ´max_array_dim = 8      (Wrong)
-SET "`max_array_dim = 8"    (Right)
```

For more information about the **SET** command, refer to Chapter 3.

2.2.6 Window pane Options: -WPnn and -NWP

By default, DEBUG divides the window into three window panes: the source display window pane, the DEBUG commands window pane, and the program I/O window pane. In this section, we concentrate on the DEBUG commands window pane and the program I/O window pane.

By default, the DEBUG commands window pane and the target program I/O window pane take up equal space as shown in Figure 2-3. You can scroll through the DEBUG commands transcript pad and the target program I/O transcript pad work just as you would through any other DOMAIN transcript pad. DEBUG uses the error input and error output streams for its I/O.



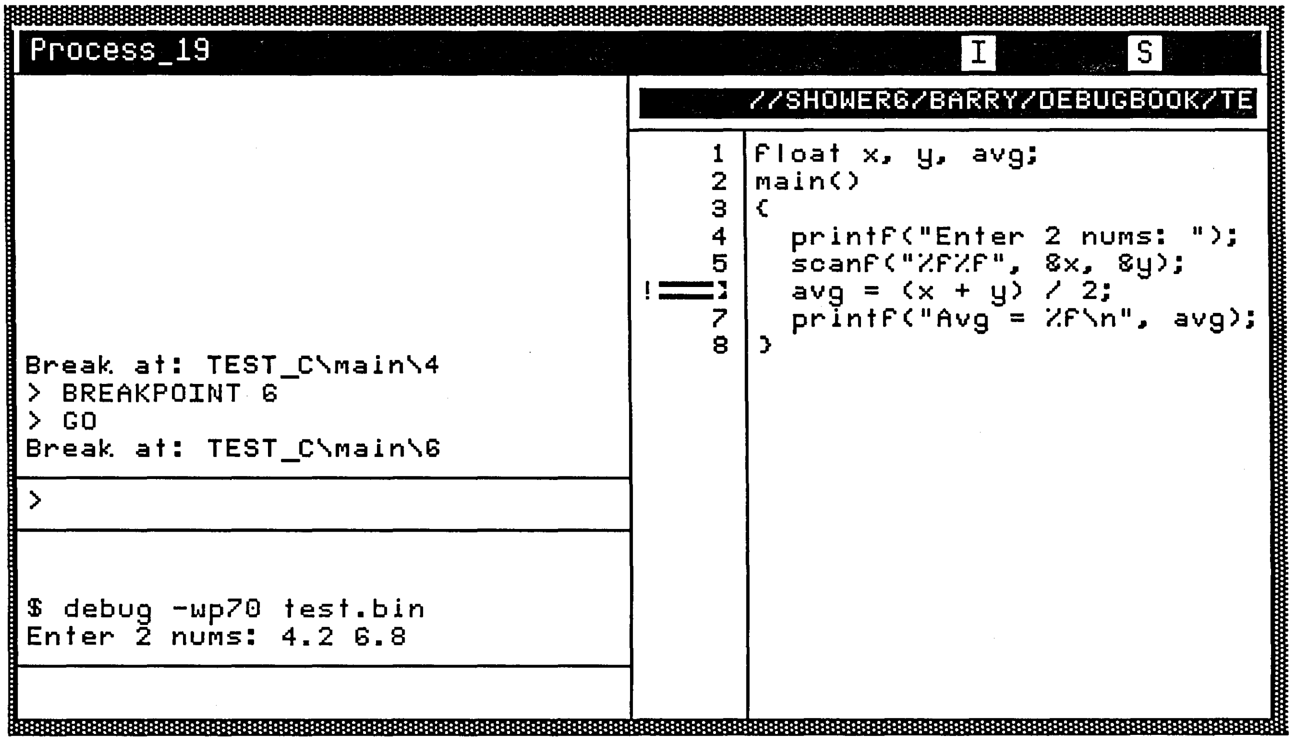
```
Process_16                                          I        S
                                  //SHOWERG/BARRY/DEBUGBOOK/TE
                                       float x, y, avg;
                                       main()
                                       {
Break at: TEST_C\main\4                  printf("Enter 2 nums: ");
> BREAKPOINT 6                           scanf("%f%f", &x, &y);
> GO                               !     avg = (x + y) / 2;
Break at: TEST_C\main\6                  printf("Avg = %f\n", avg);
                                       }
>

$ debug test.bin
Enter 2 nums: 4.2 6.8
```

Figure 2–3. Default Window Panes.

Use the –WPnn options to specify window panes of differing sizes. You can set the DEBUG window pane size in increments of 10 percent of the transcript window. The options are:

–WP10, –WP20, ... –WP90 = DEBUG window pane size of: 10%, 20%, ... 90%

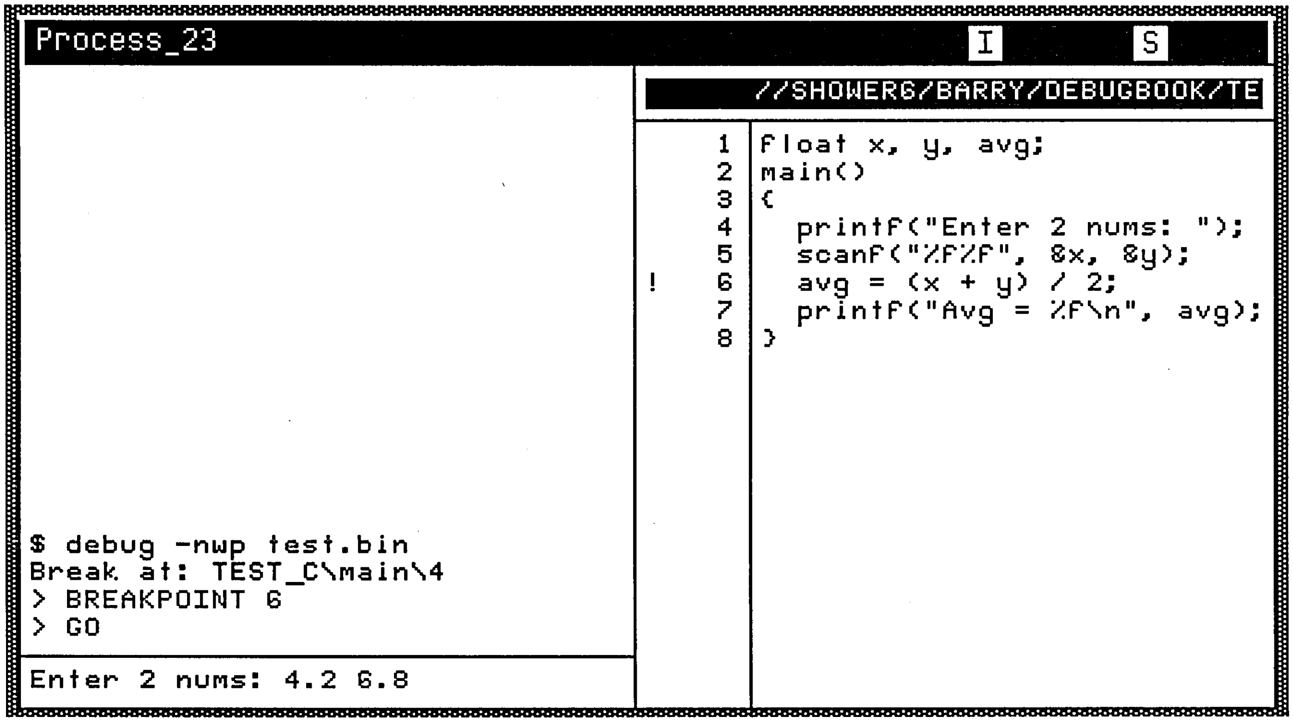
For example, Figure 2–4 demonstrates a DEBUG window pane size of 70%.



```
Process_19                                          I        S
                                  //SHOWERG/BARRY/DEBUGBOOK/TE
                                       float x, y, avg;
                                       main()
                                       {
                                         printf("Enter 2 nums: ");
                                         scanf("%f%f", &x, &y);
                                   !     avg = (x + y) / 2;
                                         printf("Avg = %f\n", avg);
                                       }
Break at: TEST_C\main\4
> BREAKPOINT 6
> GO
Break at: TEST_C\main\6
>

$ debug -wp70 test.bin
Enter 2 nums: 4.2 6.8
```

Figure 2–4. –WP70 Creates Nonequal Window Panes.

The -NWP option suppresses the creation of separate input and transcript pads; therefore, all DEBUG commands mix with program input and output as illustrated in Figure 2-5.



```
$ debug -nwp test.bin
Break at: TEST_C\main\4
> BREAKPOINT 6
> GO
```

```
Enter 2 nums: 4.2 6.8
```

```
   1  float x, y, avg;
   2  main()
   3  {
   4    printf("Enter 2 nums: ");
   5    scanf("%f%f", &x, &y);
!  6    avg = (x + y) / 2;
   7    printf("Avg = %f\n", avg);
   8  }
```

Figure 2-5. -NWP Combines the DEBUG Commands Window Pane and the Target Program I/O Window Pane.

The primary advantage of the -NWP option is that it causes the DEBUG commands you enter to be preserved when the DEBUG session ends. (If you do not specify -NWP, the DEBUG commands disappear.)

The primary disadvantage of -NWP, is that if you specify it, DEBUG may not be able to display anything if your program

- Creates a frame.
- Does direct graphics in a window. That's because direct graphics programs take control of the display and keyboard (resources that the Display Manager normally controls). If a breakpoint or fault occurs while you're debugging a program of this kind, DEBUG returns the display and keyboard to the Display Manager's control. The program regains control of these resources just before it resumes execution.
- Relies on exclusive use of STREAM_$ERRIN and STREAM_$ERROUT.

If you specify -NWP and -PROC on the same command line, DEBUG ignores the -NWP.

2.2.7 Source Display Options: -SRC, -NSRC, -SRC_T, and -SRC_R

The source display feature allows you to view your source code as you debug it. Use one of the DEBUG options -SRC, -SRC_T, or -SRC_R to trigger source display debugging. There are only minor differences among the three:

- If you specify -SRC_R, DEBUG puts the source display to the right of the other window panes. For example, see Figure 2-4.

- If you specify –SRC_T, DEBUG puts the source display on top of the other window panes. For example, see Figure 2–1.
- If you specify –SRC in a window that is wider than it is tall, DEBUG puts the source display to the right of the other window panes (i.e., like –SRC_R).
- If you specify –SRC in a window that is taller than it is wide, DEBUG puts the source display on top of the other window panes (i.e., like –SRC_T).

The –SRC option is the default. Use the –NSRC option to suppress source display.

If you specify –NWP with –SRC, –SRC_R, or –SRC_T, DEBUG divides the window into only two window panes (instead of three) and puts the source display to the right or on top.

2.2.8 The –SDIR Option

The –SDIR (Source Directories) option allows you to specify one or more alternate directories for the source filename(s) stored in the object file. If DEBUG cannot find a source file that has the pathname stored in the object file, it extracts the filename section of the pathname, combines it with the first pathname from the list created with –SDIR, and searches again. DEBUG continues to search through the list of pathnames until the search succeeds or until the end of the list. The format of –SDIR is:

–SDIR pathname ... *–SDIR pathname*

In other words, the key word –SDIR must be followed by a pathname. Furthermore, the command line can optionally contain more than one of these options.

DEBUG processes the pathnames from left to right across the command line to compose the SDIR list. You can also create or add to the SDIR list with the **SOURCE** command described in the "SOURCE" listing of Chapter 3.

2.2.9 The –GLOB Option

The –GLOB option is detailed in Appendix B. It should only be of concern for those programmers wishing to debug installed libraries.

2.2.10 The –SMAP Option

If you specify the –SMAP option, DEBUG prints a section load map at startup. For example, here is a sample load map:

```
Object Module "//SHOWER6/BARRY/DEBUGBOOK/TEST.BIN"
Section Map:
  # Location   Size    Name
  1 00008000 000000A8 PROCEDURE$
  2 000080A8 00000068 DATA$
  3 009080E8 00000104 DEBUG$
  4 00008110 00000004 AVG
  5 00008114 00000004 Y
  6 00008118 00000004 X
```

Chapter 3

Using DEBUG

In Chapter 2, we explained how to invoke DEBUG; here we explain how to use DEBUG once it has been invoked. This chapter starts with some important facts about using DEBUG; the remainder of this chapter is an encyclopedia of DEBUG commands.

If you are not familiar with DEBUG, we suggest that you read the tutorial introduction in Chapter 1, focus on sections 3.1, 3.2, and 3.3 of this chapter, and concentrate on the following listings later in this chapter:

- BREAKPOINT
- EXAMINE
- GO
- QUIT
- Routine-Name
- Variables
- STEP

3.1 Entering DEBUG Commands

DEBUG provides many ways to enter DEBUG commands. The simplest method is simply to enter one command at a time in response to the DEBUG prompt (>); for example:

```
> EXAMINE prufrock
> BREAKPOINT 17
> QUIT
```

A **command string** is two or more DEBUG commands separated by semicolons. A command string cannot exceed 512 characters. Here are three simple commands combined into a command string:

```
> EXAMINE prufrock; BREAKPOINT 17; PRINT carray
```

An **action-list** is a command or command string enclosed by a pair of brackets. Usually you do not enter an action-list by itself but as an argument to a more complex command such as **IF**. You can make an action-list span more than one line by not closing the brackets until the end of the action list. For example, the following action-list spreads across three lines:

```
> [EXAMINE prufrock;
_  BREAKPOINT 17;
_  PRINT carray]
>
```

Note that DEBUG prompts you with an underscore (_) until you include the closing brackets.

Another way to enter commands is to store them in a file and allow DEBUG to execute all the commands in the file. There are a number of ways to specify such files:

- If the `user_data` directory in your home directory contains a file named startup_debug, DEBUG reads and executes the commands in the file when you invoke DEBUG.
- If the working directory contains a file named `startup_debug`, DEBUG reads and executes the commands in the file when you invoke DEBUG.
- If you specify the -READ pathname option when you invoke DEBUG, DEBUG reads and executes the commands in pathname when you invoke DEBUG.
- If you specify the **READ** pathname command after you've invoked DEBUG, you can get DEBUG to read and execute the commands in pathname during your DEBUG session.

Finally, you can group one or more DEBUG commands into a macro by using the **MACRO** command. Then, if you want DEBUG to execute all these commands, you merely specify the name of the macro.

3.2 Overview of DEBUG's Features

This section provides an overview of DEBUG features.

3.2.1 Setting Breakpoints and Tracing

Use the **BREAKPOINT** command to set a breakpoint. If you just want to know that your program executed a certain statement, use the -TRACE option with **BREAKPOINT**. DEBUG does not stop the program when you use -TRACE. DEBUG permits you to specify an action-list for DEBUG to perform when it reaches the breakpoint.

3.2.2 Controlling Program Flow

DEBUG permits you to step through the program on a statement by statement basis or to go until the next breakpoint is reached. You can restart a stopped program from the statement where it last stopped or from anywhere else in the same routine. You can stop the execution of the current routine and, if the routine has a return value or can take a FORTRAN alternate exit, you can force the return parameters to whatever values you wish. The **GO** and **STEP** commands control program flow.

3.2.3 Operating on Variables

DEBUG can show the values, give the virtual address, describe the data types, and change the values of variables. It can perform these operations on all FORTRAN, Pascal, and C variables with one exception: Pascal FILE variables. You can only describe (with the **DESCRIBE** command) a Pascal FILE variable. You can, however, do anything with a file_variable^.

When you use DEBUG to examine the contents of a variable, you can display the data according to the variable type (for example, integer data displayed as integers) or in some other standard format (hexadecimal or ASCII, for example).

You can also use variables as terms in expressions that DEBUG evaluates.

Use the **PRINT** and **EXAMINE** commands to display a variable's value, and use the **SET** command to change a variable's value.

3.2.4 Displaying Traceback and Examining Routine Arguments

The DEBUG command **ARGS** (arguments) displays the arguments, if any, of a routine. The DEBUG command **TB** (traceback) traces the calling sequence that led to the current routine, and can optionally show the arguments of each routine in the calling sequence.

3.2.5 Creating Debugger Macros and Definitions

If you have commands that you often enter or text strings that you often use, you can define debugger names as shorthand for those strings or commands. This allows you, for example, to examine a group of variables repeatedly without typing all the variable names each time.

Use the **MACRO** command for creating command macros and the **DEFINE** command for creating text string definitions.

3.2.6 Setting Environments

In most instances, DEBUG automatically searches the proper environment (i.e., routine) when you ask to set a breakpoint, examine a variable, etc.; however, in some cases, you may want to explicitly name a different environment for DEBUG to look in. The two most common reasons for setting a different environment are:

- When the program is stopped in a routine, but you want to examine the value of a variable stored in another routine.
- When the program is stopped in a routine, but you want to set a breakpoint in another routine.

You control the environment with the **ENVIRONMENT** command.

3.2.7 Running a Shell From the Debugger

DEBUG allows you to run a Shell by invoking the **SHELL** command. This Shell is exactly like any DOMAIN operating system Shell: it allows you to run programs, execute Shell commands, and so on. Once you close the Shell, you return to debugging your program.

3.3 General DEBUG Information

Here, we discuss command abbreviations, case sensitivity, and debugger names.

3.3.1 Command Abbreviations

Most DEBUG commands can be abbreviated. For example, you can use any of the following to specify the **QUIT** command:

```
> Q
> QU
> QUI
> QUIT
```

But how can you find the legal abbreviations for each command? The "FORMAT" section of each command listing shows the name of each command in boldface. The letters in uppercase are required, and the letters in lowercase are optional. For example, the format for the QUIT command is displayed as **Quit** indicating that only the Q is required.

3.3.2 Case Sensitivity

The keywords in DEBUG commands are case insensitive. The names of procedures, functions, subroutines, and variables in Pascal and FORTRAN programs are also case insensitive. However, the names of functions and variables in C programs are case sensitive.

3.3.3 Debugger Names (of Variables, Macros, and Definitions)

You can use **debugger names** in definitions made with the **MACRO** and **DEFINE** commands, and as names for **debugger variables**. A debugger name has the following general format:

```
`identifier
```

The accent grave (`), often called a "tic", must be the first character. Following the accent grave, you must specify an identifier. The first character of the identifier must be a letter. Each succeeding character can be a letter, a digit, an underscore (_) or a dollar sign ($). The name can contain up to 32 characters.

3.4 The DEBUG Encyclopedia

The remainder of this chapter consists of an encyclopedia of DEBUG commands and concepts. We've organized the listings alphabetically. The following list summarizes the DEBUG commands:

ARGS	Displays the names and values of each argument in an active routine.
BREAKPOINT	Sets a breakpoint.
DEFINE	Defines an abbreviation for a frequently-used command or string.
DELETE	Deletes macros, definitions, breakpoints, or alternate source directories.
DESCRIBE	Describes the data types of one or more variables.
ENVIRONMENT	Displays or changes an environment.
EXAMINE	Displays the value(s) of one or more variables.
EXIT	Ends a debugger session.
GO	Begins or resumes execution of the target program.
HELP	Displays information about DEBUG commands.
IF	Evaluates a given expression and conditionally executes an action-list.
JUMP	Moves between commands in an action-list.
LIST	Lists information on macros, definitions, breakpoints, routines, program variables, DEBUG variables, and alternate source directories.
MACRO	Defines a sequence of DEBUG commands that you can then invoke with a single name.
PRINT	Displays the value(s) of one or more expressions.
QUIT	Ends a debugger session and terminates the target program.
READ	Executes a file of DEBUG commands.
SET	Assigns a new value to a program variable or debugger variable.
SHELL	Invokes a command Shell.
SIGNAL	Simulates a fault.
SOURCE	Specifies the source code that DEBUG displays.

STCODE	Displays the text of a system error code.
STEP	Executes the next statement in the target program.
TB	(Traceback) Displays the current call stack.
VA	Displays the virtual address of a variable, a routine, or the program counter.
#	Adds a comment line to the DEBUG transcript pad.

The following list summarizes the DEBUG conceptual listings:

Action–Lists	You use an action–list to group commands for easy access.
Arrays	When you want to examine or define a portion of an array you can turn to this listing for information on specifying all or part of an array.
Debugger–Variables	In addition to program variables from your source code, DEBUG understands two other kinds of variables: user–defined debugger variables and predefined debugger variables. Both are described here.
Expressions	Some DEBUG commands can take an expression as an argument. Here, we explain the rules for composing a legal DEBUG expression.
Faults	This listing explains how DEBUG handles faults and fault–handlers.
Pointer–Variables	This listing explains how to manipulate pointer variables during DEBUG sessions. (Also see Section 4.2.5 in Chapter 4.)
Routine–Name	In a program with more than one routine, it is important that you understand how to identify the proper routine; turn here for that understanding.
Source Code Display	This listing details the source code that DEBUG displays.
Statement–ID	This listing explains how to specify a particular statement in a program (so that you can set a breakpoint there, for instance).
Variables	Turn to this listing for information on specifying program variables during DEBUG sessions.

Action-Lists -- Use an action-list to group commands together for easy access.

FORMAT

An action-list is not a command, but a way of entering commands.

[*debug-command1; debug-command2; ...; debug-commandN*]

REQUIRED

None.

OPTIONAL

debug-command	Zero or more DEBUG commands separated by semicolons (;). See Section 3.1 for a description of DEBUG commands.

DESCRIPTION

An action-list is a way to group commands for easy access. Usually, action-lists are used as arguments to other commands. For instance, the **BREAKPOINT** command takes an action-list as an optional argument. However, you can also issue an action-list by itself, unattached to another command.

Enclose the entire action-list in square brackets. For example, here is a sample action-list:

```
[EXAMINE x; PRINT string; GO]
```

DEBUG prompts for more commands for the action-list as long as there are more left brackets than right brackets, and the total length of the command string does not exceed 512 characters. The prompt for more input is an underline character. For example, here is a command with an action-list that is spread over four lines. For clarity, we show the commands in the action-list on separate lines (DEBUG does not require this format).

```
> BREAKPOINT beach\  -DO [EXAMINE david;
_       PRINT 'Value should be:', mermaids
_       DESCRIBE waves, storm;
_       EXAMINE waves, storm]
```

DEBUG also uses the underline character as a prompt for more input when you use square brackets to define arrays and sets. Refer to the "Arrays" listing for details.

DEBUG does not check the contents of an action-list until you execute it. In other words, if you made some sort of syntax error within an action-list, DEBUG does not detect the error when you define the action-list, but detects the error when you execute the action-list.

GO and STEP Commands Within Action-Lists

If you place a **GO** command in an action-list, it will be the last command executed in the action-list. For instance, although the following action-list looks temptingly correct

```
> [BREAKPOINT 25; GO; EXAMINE x]
```

DEBUG only executes the **BREAKPOINT** and **GO** commands. DEBUG does not execute the **EXAMINE** because it appears after the **GO**. (You could put the **EXAMINE** command in an action-list associated with the breakpoint to get the desired result.)

A **STEP** command inside an action–list does not suppress the execution of the commands that follow **STEP**. In other words, a **STEP** command within an action–list works in the same way as a **STEP** command outside of an action–list.

ARGS -- Displays an active routine's arguments and argument values.

FORMAT

Args *routine-name*

REQUIRED ARGUMENTS

None.	If you do not specify a routine-name, DEBUG displays the arguments of the current routine.

OPTIONAL ARGUMENTS

routine-name	The name of an active routine whose arguments you want to investigate. (See the "Routine-Name" listing later in this encyclopedia for details about specifying a routine-name. Note that a routine-name can include an activation-number which identifies a specific activation of a recursive routine. Don't forget that a routine-name must end with a backslash \.)

DESCRIPTION

Use the **ARGS** command to display the name and value of each argument of the specified active routine. Note that **ARGS** only reports information about the routine's arguments (i.e., parameters); it does not report the routine's local variables. DEBUG does not necessarily display the arguments in the same order you defined them.

The optional *routine-name* must be the name of an active routine. The only routines that are active at any one time are

- The routine where DEBUG is currently stopped.
- The routine(s) that called this routine. (Therefore, the main routine of the program is always active.)

Before issuing **ARGS**, you may want to set the debugger variables **`max_qual**, **`max_var_len**, or **`max_array_dim**, which are detailed in the "Debugger Variables" listing later in this encyclopedia.

See also the -Args option of the **TB** command.

Note To FORTRAN Users

DEBUG does not know whether a a program entered a subroutine through the main or through an alternate entry point. When you issue an **ARGS** command, DEBUG always tries to display the arguments associated with the main entry. Thus, the data that DEBUG displays may be invalid if the program entered the routine through an alternate entry.

You can use **EXAMINE** or **SET** commands to access the arguments you know to be valid. If you use the **ARGS** command or attempt to **EXAMINE** or **SET** invalid arguments, you will get bad data and possibly an access violation or odd address fault.

EXAMPLES

Suppose you are stopped at a breakpoint somewhere in routine F. To study the arguments of this routine, issue the following command:

```
> ARGS
SAMPLEMOD\MATH\F\x = 5
SAMPLEMOD\MATH\F\y = 3.14
```

The results indicate that routine F supports two arguments, x and y, with values of 5 and 3.14, respectively. We could have produced the same results by issuing a routine-name, for instance:

```
> ARGS F\
SAMPLEMOD\MATH\F\x = 5
SAMPLEMOD\MATH\F\y = 3.14
```

Since routine F was called by routine MATH, we can also investigate the arguments of routine MATH as follows:

```
> ARGS MATH\
SAMPLEMOD\MATH\bitmap_number = 4
SAMPLEMOD\MATH\pixel_range = 1000
SAMPLEMOD\MATH\text_color = 2
```

Arrays -- Here, we explain how to specify arrays in commands that accept them as arguments.

DESCRIPTION

DEBUG supports several ways to specify an array or a portion of an array. Before you specify an array, you may want to read the "Debugger Variables" listing later in this encyclopedia. The variable **`max_array_dim** is particularly useful when specifying arrays.

Consider, for example a three-dimensional array (10 x 9 x 8) named `stearns`. You can refer to this variable or parts of this variable in any of the following ways:

- Simply give the array name, in which case any operation refers to the whole array (for example, `stearns`).
- Name an element explicitly (for example, `stearns(1,5,5)` or `stearns[1,5,5]`). (You can use parentheses or square brackets interchangeably, regardless of the source language.)
- Specify, in place of an array subscript, a range of the form:

  expression:expression

  In this case, the first expression must be less than or equal to the second, and both must evaluate to integers or the appropriate index type, for example:

```
stearns(1:5,6:9,5))
alphabet['a':'z']          (A Pascal array indexed by chars.)
```

- Replace an explicit reference to one or more array dimensions with an asterisk (*), which refers to all subscripts in that dimension. Omitting the dimension has the same effect, as long as you include all dimensions to the left, and do not include any dimensions to the right. For example, each of the following examples refer to the entire array:

```
stearns(1:10,1:9,1:8)
stearns[*,*,*]
stearns(*,*)
stearns[]
stearns
```

  The following are also equivalent to each other:

```
stearns(8)
stearns[8,*,*]
stearns(8:8,1:9,1:8)
```

Here are some more examples of legal array references:

`stearns(1:8)`	*(Refers to the first eight elements of the first dimension, and all elements of the other two dimensions.)*
`stearns(8,1:9)`	*(Refers to the eighth element of the first dimension and all elements of the second and third dimensions.)*
`stearns(*,8)`	*(Refers to all of the first and third dimensions and the eighth element of the second dimension.)*
`stearns(9,8)`	*(Refers to the ninth element of the first dimension, the eighth element of the second dimension, and all elements of the third dimension.)*

```
stearns(10,8,2:4)  (Refers to the tenth element of the first dimension
                     the eighth element of the second dimension, and the
                     second, third, and fourth elements of the third
                     dimension.)
```

You do not have to restrict yourself to the actual defined bounds of an array. If you try an operation with array subscripts greater than the array boundaries, the compiler performs the operation as if the array were actually that big, and, if you use the **SET** command, DEBUG alters the memory contents at that location. DEBUG does, however, warn you that you are referencing outside the array boundaries. (DEBUG does not issue a warning in the case of one–dimensional arrays with one element.)

Arrays and C Pointers

DEBUG permits you to access C pointer variables as arrays by specifying subscripts. The asterisk subscript is legal in this context only if the pointer points to a char, in which case the size of the array is determined by scanning for a null byte. See Section 4.2.5 of this manual for details.

Some More Examples

Consider a one–dimensional array of 120 characters. To view only the first six characters of the array, enter a command like the following:

```
> EXAMINE my_char_array[1:6]
TEST\my_char_array[1] = M
TEST\my_char_array[2] = a
TEST\my_char_array[3] = r
TEST\my_char_array[4] = i
TEST\my_char_array[5] = l
TEST\my_char_array[6] = y
```

To display an array of characters as a string, use the **PRINT** command.

The debugger variable **`max_array_dim** can be particularly useful for examining arrays. When you examine an array without specifying the dimensions, this variable determines how many array entries, per dimension, DEBUG examines. For example, consider the affect of **`max_array_dim** in the following example:

```
> EXAMINE carray
TEST\carray[1] = K
TEST\carray[2] = e
TEST\carray[3] = r
TEST\carray[4] = r
TEST\carray[5] = y
>
> SET `max_array_dim = 2
>
> EXAMINE carray
TEST\carray[1] = K
TEST\carray[2] = e
```

The **EXAMINE** command ignores **`max_array_dim** when you use the "*" or "expr:expr" subscripts; for example, even though **`max_array_dim** is 2 we still get the following results:

```
> EXAMINE carray[1:4]
TEST\carray[1] = K
TEST\carray[2] = e
TEST\carray[3] = r
TEST\carray[4] = r
```

BREAKPOINT -- Sets a breakpoint.

FORMAT

	Exactly one from this column.	None, any, or all from this column.
Breakpoint	*routine-name* *statement-id* *-Exit* *-Here* *-VA address* *-VA routine-name\offset*	*-Do action-list* *-Trace* *-After integer*

REQUIRED ARGUMENTS

The **BREAKPOINT** command must be followed by exactly one of the following:

routine-name Identifies where DEBUG will set a breakpoint. This must be either a routine-name by itself (see the "Routine-Name" listing of this encyclopedia for details) or a statement-ID (see the "Statement-ID" listing). If you specify a routine-name by itself, DEBUG sets the breakpoint just before the first executable statement in the named routine. Don't forget that a routine-name by itself must end with a backslash (\).

-Exit Sets a breakpoint on the exit of the current routine. (Note, each routine has only one physical exit even though it may have multiple RETURN statements. DEBUG sets the breakpoint on the physical exit only.)

-Here Sets a breakpoint at the current run environment location. Any user environment specification is ignored.

-VA Allows you to set a breakpoint on a virtual address. -VA must be followed by one of the following:

address	A virtual address. By default, DEBUG views the number you enter as a decimal value; you can, however, change the base to hexadecimal or octal. (See the "Expressions" listing for details on setting different bases.)
routine-name\offset	A routine-name followed by an integer offset (in bytes) from the beginning of the routine provides a virtual address.

If the virtual address does not correspond to the start of a source statement, DEBUG may have difficulty setting the environment when the program hits the breakpoint.

OPTIONAL ARGUMENTS

-Do action-list Causes DEBUG to execute the action-list when the break occurs. See the "Action-List" listing earlier in this encyclopedia for details.

-Trace Prevents the breakpoint from stopping program execution. Instead, DEBUG prints a message that identifies the tracepoint, and executes any action-list that you have given (see the -DO option). Target program execution continues.

–After integer DEBUG maintains a "hit" count for every breakpoint. The hit count measures how many times a statement is executed. (That is, the hit count is incremented each time the statement at that breakpoint is executed.) You must supply an integer argument to –After. When the hit count is less than the integer, the breakpoint is not activated. When the hit count equals the integer, the break is activated, and the hit count is reset to zero. If you never specify –After in a DEBUG session, the default value of the integer is 1.

The integer you supply will not change unless you explicitly change it with another –After. For example, if you specify –After 3, then the program breaks on the third, sixth, ninth, etc. times that the statement is executed.

DESCRIPTION

Use the **BREAKPOINT** command to set a breakpoint. The breakpoint always occurs immediately before your program executes the statement, exit, or routine where you set the break. For example, if you set a breakpoint at line 100, then the program halts just before it executes the statement at line 100. If you set a breakpoint at a routine, DEBUG executes the routine prolog (which establishes the addressing environment, allocates local variables, etc.), but halts before executing the first user statement of the routine.

If you specify an action–list with the breakpoint, DEBUG invokes the commands in the action–list. DEBUG then checks to see if the breakpoint is a trace. If it is a trace, DEBUG prints a message notifying you that the break occurred, and execution continues. If the breakpoint is not a trace, DEBUG requests a command.

Use the **LIST** command to learn the positions of all active breakpoints. Use the **DELETE** command to remove one or more breakpoints. (See the "LIST" and "DELETE" listings for details.)

Only one breakpoint can be set on a statement. Setting a breakpoint deletes any previous breakpoint on the same statement.

EXAMPLES

Here are some simple uses of the BREAKPOINT command:

`> BREAKPOINT -Here`	*(Set a breakpoint at the current line.)*
`> BREAKPOINT 15`	*(Set a breakpoint at line 15.)*
`> BREAKPOINT aloe\37`	*(Set a breakpoint in routine aloe at line 37)*
`> BREAKPOINT vera\`	*(Set a breakpoint on the first procedural line of routine vera.)*
`> BREAKPOINT -Exit`	*(Set a breakpoint at every line in which this routine can return to its caller.)*
`> BREAKPOINT -VA 16#10050`	*(Set a breakpoint at hexadecimal virtual address 10050.)*
`> BREAKPOINT 17 -After 5`	*(Set a breakpoint at line 17, but don't actually trigger the break until the fifth time that the statement has been executed.)*

The following is an example of a breakpoint set with a –DO action–list clause.

```
> BREAKPOINT 19 -DO [PRINT 'x = ', x]
> GO
Break at: SAMPLEMOD\vera\19
x = 5.23
```

Suppose that you want to know the value of a particular variable in a loop with each iteration of the loop, but you don't want to restart after every break. Fortunately, DEBUG supplies two simple ways to let you do this. The first method involves the –Trace option. For example, suppose you want to know variable r's value just before line 22 is executed. To find this out, you could issue a command like the following:

```
> BREAKPOINT 22 -Trace -DO [PRINT 'r = ',r]      (Trace the value of r)
> GO
Trace at: TEST_C\main\22
r = 550                           (First time through loop r = 550)
Trace at: TEST_C\main\22
r = 600                           (Second time through loop r = 600)
Trace at: TEST_C\main\22
r = 650                           (Third time through loop r = 650)
```

The second method involves an action-list that contains the **GO** command. It produces results nearly identical to the first method. For example:

```
> BREAKPOINT 22 -DO [PRINT 'r = ', r; GO]
> GO
Break at: TEST_C\main\22
r = 550                           (First time through loop r = 550)
Break at: TEST_C\main\22
r = 600                           (Second time through loop r = 600)
Break at: TEST_C\main\22
r = 650                           (Third time through loop r = 650)
```

Now consider another common debugging chore. Suppose you want your program to halt when the value of variable p becomes negative or zero. To accomplish this, set a series of breakpoints wherever you think the value of p might change. The **BREAKPOINT** commands might look something like this:

```
> BREAKPOINT 17 -DO [IF p > 0 [GO]]
> BREAKPOINT 25 -DO [IF p > 0 [GO]]
> BREAKPOINT 44 -DO [IF p > 0 [GO]]
```

Then to set the program in motion, just issue a simple **GO** command. The program won't halt until the value of p becomes greater than zero.

Debugger Variables -- Special variables that you can create or access during your debugging session.

DESCRIPTION

In addition to program variables (i.e., the variables from your source code), you can also manipulate two other kinds of variables in a DEBUG session, namely:

- User-defined debugger variables
- Predefined debugger variables

We describe them separately.

You can use debugger variables wherever you would use program variables. You can also mix program variables and debugger variables in expressions.

User-Defined Debugger Variables

You can define your own variables for temporary use during a debugging session. Use the **SET** command to create and initialize these variables; for example:

```
> SET `heather = 4
> SET `weather = 30.10
> SET `rainy = 'c'
```

Notice that each user-defined debugger variable name must begin with an accent grave (`). The value you supply determines the variable's data type. For example, by setting `` `weather `` equal to 30.10, you implicitly declare the variable as a 64-bit real. You can create user-defined debugger variables of the following types:

- 32-bit integer (if you initialize the variable to any integer value).
- 64-bit real (if you initialize the variable to any real value).
- Logical (boolean) (if you initialize the variable to .TRUE. or .FALSE.).
- Character (if you initialize the variable to a character constant).

Assigning a new value to an existing debugger variable may change its type.

NOTE: You cannot define a debugger variable with the same name as an existing macro or definition.

We now demonstrate a practical application for user-defined debugger variables. Suppose you want to discover how many times a program executes a particular statement. To do so, you can create a debugger variable named `` `counter ``, initialize it to zero, then increment it each time a statement is executed. The whole sequence looks something like this:

```
> SET `counter = 0
> BREAKPOINT 27 -DO [SET `counter = `counter + 1] -Trace
> GO

...                    (The program increments `counter each time the
                        statement at line 27 is executed.)
> EXAMINE `counter
`counter = 43          (The program reaches the statement 43 times.)
```

Predefined Debugger Variables

The predefined debugger variables are special purpose variables that provide you with an easy way to control some of the parameters associated with DEBUG's operation. The term **predefined** can be somewhat misleading. Debugger variables are predefined in the sense that DEBUG understands a special meaning for each of them; however, in a sense they are not predefined because DEBUG does not allocate any storage space for a predefined variable until you activate them. You activate a predefined debugger variable by setting its value with the **SET** command.

You can set and examine debugger variables just as you would a program variable. (However, unlike program variables, you cannot examine a debugger variable until it has been set.) The predefined DEBUG variables are:

`FAULT_STATUS	DEBUG automatically sets its value to the status code of the most recent fault. See the "Faults" listing for details.
`MAX_ARRAY_DIM	Controls the maximum number of array dimensions that the **PRINT**, **ARGS**, and **EXAMINE** commands display. See the "Arrays" listing for details.
`MAX_BAD_CHARS	Limits the number of unprintable characters that DEBUG displays when outputting data in ASCII format.
`MAX_QUAL	Limits the number of qualifiers in a routine–name or variable–name.
`MAX_STRING_LEN	Controls the length of a string that the **PRINT** command displays. For details, see the "PRINT" listing later in this encyclopedia.
`MAX_VAR_LEN	Truncates all but the specified number of characters in a variable name.
`SRC_ADJUST	Sets the minimum number of lines of source code displayed above and below the current source line. See the "Source Code Display" listing for details.
`SRC_TRY_BAK	Controls the display of the backup version of a file of source code. See the "Source Code Display" listing for details.

To activate a predefined debugger variable, you merely use the **SET** command to establish the variable's initial value; for example:

```
> SET `MAX_VAR_LEN = 3
```

You cannot delete a predefined variable, but you can deactivate one. To deactivate any of these variables, set its value to a negative number; for example, to deactivate **`max_var_len** issue a command like:

```
> SET `MAX_VAR_LEN = -1
```

To see a list of active debugger variables (both user defined and predefined), use the **LIST** command.

Let's now examine some of the predefined debugger variables.

`max_bad_chars

When you use the **PRINT** command to display a string (particularly in FORTRAN or Pascal) it is possible that some of the characters in the array are "unprintable." You use **`max_bad_chars** to limit the number of unprintable characters that DEBUG displays. For example, notice how the following use of **`max_bad_chars** eliminates some needless characters:

```
> PRINT cstr
Hi ma<^P><^P><^P><^P><^P>

> SET `max_bad_chars = 2
> PRINT cstr
Hi ma<^P><^P>

> SET `max_bad_chars = 0
> PRINT cstr
Hi ma
```

DEBUG truncates output when the specified number is exceeded. Setting the value of **`max_bad_chars** to zero causes DEBUG to truncate strings when it encounters the first unprintable character. This is useful in the common case of variable-length data stored in a fixed-length buffer, leaving part of the buffer uninitialized.

When you use **EXAMINE** to display a character array, DEBUG displays each element of the array as a separate one-character string. Using **`max_bad_chars** will not reduce the number of elements that DEBUG displays.

DEBUG separates unprintable characters into two categories -- unprintable control characters and other unprintable characters. DEBUG displays unprintable control characters in the format '<^x>', and other unprintable characters in the format '<nn>', where nn is the character's hexadecimal value. The newline character (^J) is considered printable, but all other control characters are considered unprintable.

`max_qual

By default, the **ARGS, EXAMINE,** and interactive **SET** commands display variable names in *fully-qualified form.* That is, DEBUG precedes the variable names with the routine-name containing them. Use **`max_qual** to limit the maximum number of qualifiers (routine-names) displayed prior to the requested routine-name or variable-name. For example, consider the effect of **`max_qual** on the examination of the following variable:

```
> SET `max_qual = 2
> EXAMINE self
TEST_C\f\self = 3.400000      (2 qualifiers prior to variable-name.)

> SET `max_qual = 1
> EXAMINE self
f\self = 3.400000             (1 qualifier prior to variable-name.)

> SET `max_qual = 0
> EXAMINE self
self = 3.400000               (0 qualifiers prior to variable-name.)

> SET `max_qual = -1
> EXAMINE self
TEST_C\f\self = 3.400000      (Fully qualified variable-name.)
```

Setting **`max_qual** to a negative number forces DEBUG to fully qualify all names.

`max_var_len

Use the **`max_var_len** variable to reduce the length, not values, of the variable names displayed by the **ARGS, DESCRIBE,** or **EXAMINE** commands. For example, before issuing the ARGS command, you might set the debugger variable **`max_var_len** so that each displayed argument has only a certain number of characters. First, consider an ARGS command with the default **`max_var_len** specified:

```
> ARGS
ERROR\TEXT(1) = H
ERROR\TEXT(2) = i
ERROR\TEXT(3) =
ERROR\TEXT(4) = M
ERROR\TEXT(5) = o
ERROR\TEXT(6) = m
```

Now consider setting **`max_var_len** to 2 as follows:

```
> SET `max_var_len = 2

> ARGS
...1) = H
...2) = i
...3) =
...4) = M
...5) = o
...6) = m
```

Notice that DEBUG truncates from the *beginning* of the name. Also, if the variable-name has fewer than three characters, DEBUG will not truncate it.

DEFINE -- Defines an abbreviation for a frequently used command or string.

FORMAT

DEFine debugger-name text

REQUIRED ARGUMENTS

debugger-name	Any legal debugger-name. All debugger-names must begin with an accent grave (`).
text	Any text. Do not enclose the text in quotes or brackets unless you want them to become part of the definition. The text can include blanks.

OPTIONAL ARGUMENTS

None.

DESCRIPTION

Use **DEFINE** to create a name (i.e., a nickname) for some text. The text must either be a string (e.g., a routine-name) or be one DEBUG command.

For example, consider the long-hand method of setting up breakpoints in a routine:

```
> BREAKPOINT test\main$\math_io\write_io\28
> BREAKPOINT test\main$\math_io\write_io\56
```

However, by using **DEFINE** as follows, we can abbreviate `test\main$\math_io\write_io` to the simpler `` `r``:

```
> DEFINE `r test\main$\math_io\write_io\
> BREAKPOINT `r28
> BREAKPOINT `r56
```

As already noted, the text string could also be a single DEBUG command. For instance, if you frequently type the command

```
> EXAMINE barray[10, 5:*]
```

you could create an abbreviation called `` `b`` as follows:

```
> DEFINE `b EXAMINE barray[10, 5:*]
```

and then issue the abbreviation in place of the longer command; for instance:

```
> `b
```

DEBUG does not check strings for illegal commands until you actually issue the corresponding debugger-name.

Use the **LIST** command to display a list of current definitions, and use the **DELETE** command to remove definitions.

> **NOTE:** The **MACRO** command is similar purpose to **DEFINE**, but **MACRO** is more sophisticated. **MACRO** can abbreviate an entire action-list; **DEFINE** can only abbreviate a single command.

Using DEBUG Definitions as Text for Other Definitions

You can specify a DEBUG definition as text. By doing so, you can build a chain of definitions. For example, the following sequence of commands is legal:

```
> DEFINE `x  quarts
> DEFINE `y  EXAMINE `x
> `y
quarts = 4
> DEFINE `x  pints
> `y
quarts = 4
```

Notice that when we redefined `` `x``, the value of `` `y`` did not change, because the value of `` `x`` was expanded at the time we defined `` `y``. The definition of `` `y`` will not change unless you explicitly change `` `y``.

The Predefined Symbol `` `. ``

DEBUG translates the predefined symbol `` `. `` (accent grave followed by a period) as the current routine-name. For example, suppose the program is halted at a breakpoint in routine `$main\math`. If you make the following definition

```
> DEFINE `e  EXAMINE `.extraterrestrial
```

then entering the DEBUG symbol `` `tom`` at any point in the program, even in a routine other than MAIN, is exactly equivalent to entering the following command:

```
> EXAMINE $main\math\extraterrestrial
```

NOTE: You can only use this predefined symbol inside macros or inside the text of a definition. You cannot use it anywhere else.

DELETE -- Deletes macros, definitions, breakpoints, or source directories.

FORMAT

Delete	-Macro	*macro-name1 ... macro-nameN* *-All*
	-DEFine	*debugger-name1 ... debugger-nameN* *-All*
	-Breakpoint	*statement-id* *routine-name -ALL* *-HERE* *-VA address* *-ALL*
	-SDIR	*an_integer* *-All*

REQUIRED ARGUMENTS

Following DELETE, you must specify at least one of the following four arguments:

-Macro Deletes one or more DEBUG macros. Following -Macro you must specify one of the following two subarguments:

macro-name	Deletes only those macros that you specify. Macros are created with the **MACRO** command.
-All	Deletes all current DEBUG macros.

-DEFine Deletes the definitions for one or more debugger-names created by the **DEFINE** command. Following -Define you must specify one of the following two subarguments:

debugger-name	Deletes only those definitions that you specify.
-All	Deletes all definitions. The -ALL option does not affect the predefined symbol `..

-Breakpoint Deletes one or more breakpoints. Following -Breakpoint, you can optionally specify one of the following subarguments. If you do not specify a subargument, the -HERE subargument is assumed.

statement-id	Deletes the breakpoint at the given statement. See the "Statement-ID" listing for details on specifying a particular statement.
routine-name -ALL	Deletes all the breakpoints in that routine.
-HERE	Deletes the current breakpoint. In order to use this subargument, the program must currently be stopped at a breakpoint. -HERE always refers to the run environment, not any user environment defined through an **ENVIRONMENT** command.

	–VA address	Deletes the breakpoint on the given virtual address. (By default, DEBUG expects the address to be a decimal integer. Should you wish to supply a hexadecimal integer, precede the integer with 16#; for example, 16#1003C.
	–All	Deletes every breakpoint in the program.
–SDIR	Deletes one or all of the alternate source listing directories generated through the **SOURCE** command (described later in this encyclopedia) or the –SDIR command line option (described in Chapter 2). Following –SDIR, you must specify one of the following two subarguments:	
	n	Deletes exactly one of the source listing directories. n must be a positive integer between 1 and 32,767. Assuming that there are x source listing directories, you specify the *n*th one to delete. If you specify an n greater than x, DEBUG takes no action. Note that DEBUG orders the source listing directories in the same order that you define them. You can use the **LIST** command to check the order. And by the way, a DELETE –SDIR command deletes the directory's name from DEBUG's internal list, it does not delete the directory itself.
	–All	Deletes all of the source listing directories.

OPTIONAL ARGUMENTS

None.

DESCRIPTION

Use the **DELETE** command to remove just about any DEBUG construct except a debugger variable. (You cannot *delete* a debugger variable, but you can deactivate one by setting its value to a negative integer.)

You cannot delete –EXIT breakpoints without giving the specific statement numbers. You can use the LIST command (described in this chapter) to display the statement numbers.

EXAMPLES

```
> DELETE -Macro `penny                (Delete the macro named `penny.)
> DELETE -Macro -all                  (Delete every macro.)

> DELETE -DEFine `pfennig             (Delete the definition named `pfennig.)
> DELETE -DEFine -All                 (Delete all definitions.)

> DELETE -Breakpoint 22               (Delete the breakpoint at line 22.)
> DELETE -Breakpoint f\               (Delete breakpoint at first statement of
                                       routine f\.)
> DELETE -Breakpoint f\ -All          (Delete every breakpoint in routine f\.)
> DELETE -Breakpoint -Here            (Delete breakpoint where program is
                                       currently stopped.)
> DELETE -Breakpoint -VA 16#1004C (Delete breakpoint at the statement that begins
                                       at the hexadecimal virtual address 1004C.)
> DELETE -Breakpoint -ALL             (Delete every breakpoint in the entire program.)


> LIST -SDIR                          (Get a listing of all the source directories.)
Source Directories:                   (There are four source directories.)
    ~PROGS_REV6/
    ~PROGS_REV7/
    ~PROGS_REV8/
    ~PROGS_REV9/
> DELETE -SDIR 2                      (Eliminate the second source directory.)
> LIST -SDIR
Source Directories:
    ~PROGS_REV6/
    ~PROGS_REV8/
    ~PROGS_REV9/
> DELETE -SDIR -All                   (Eliminate the other three source directories.)
> list -sdir
Source Directories:
```

DESCRIBE -- Describes the data type of one or more variables.

SYNTAX

DEScribe variable1, ... *variableN*

REQUIRED ARGUMENT

variable	One or more program variables or debugger variables.

OPTIONAL ARGUMENTS

None.

DESCRIPTION

Use **DESCRIBE** to find the data type, size, and storage attribute (local, static, or register) of one or more variables. Note that **DESCRIBE** does not display the contents of the variables. Use the **EXAMINE** or **PRINT** commands (described in this chapter) to display the contents of variables.

Before issuing **DESCRIBE,** you may wish to set the debugger variables **`max_var_len**, **`max_array_dim**, or **`max_qual**. See the "Debugger Variables" listing for details.

Also before issuing **DESCRIBE**, you may wish to use the **ENVIRONMENT** command to change the user environment. See the "ENVIRONMENT" listing for details.

Note that **DESCRIBE** can only describe variables; it cannot describe types or typedefs. Furthermore, although DEBUG can describe a variable having a user-defined data type, it describes the variable in terms of the underlying standard data type.

> NOTE: In optimized code, the compiler may assign a variable to memory in some part of the program, and to a register in others. The **DESCRIBE** command always reflects the current location. Therefore, using **DESCRIBE** on the same variable at different points of the program may possibly return a different description.

EXAMPLES

First, we use **DESCRIBE** to explore some simple variables:

```
> DESCRIBE r
TEST_C\F\r = 32-bit real, local.
> DESCRIBE x
TEST_C\main\math\x = 32-bit integer, register.
```

Consider a Pascal record named `test` declared as follows:

```
VAR
   test : record
        value : integer16;
        percentage : single;
   end;
```

Here's how the **DESCRIBE** command analyzes this record:

```
> DESCRIBE test
SAMPLE\test = record (2 fields, 6 bytes), static.
SAMPLE\test.VALUE = 16-bit integer, static.
SAMPLE\test.PERCENTAGE = 32-bit real, static.
```

The following **DESCRIBE** command provides information on a Pascal set variable:

```
> describe ice_cream
TEST\F\ice_cream = set of enumerated type (6 members), local.
```

Note that **DESCRIBE** cannot list the legal identifiers of an enumerated type.

Finally, here's the description of a 3x3 array named board:

```
> DESCRIBE board
TIC_TAC\board = array (9 bytes) of array (3 bytes) of character, static.
TIC_TAC\board(1,1) = character, static.
 : : : : : : : : :
TIC_TAC\board(3,3) = character, static.
```

Notice that **DESCRIBE** gives you a general description of the array and does not explicitly describe all the elements. (Because it is an array, all the elements have the same characteristics.)

ENVIRONMENT -- Changes the user environment, or identifies the current program and user environments.

FORMAT

```
              statement-id
              -Run
ENVironment   -User
              -Caller
              -Sub
```

REQUIRED ARGUMENTS

None.	If you do not supply an argument, the command returns the name of the current environment.

OPTIONAL ARGUMENTS

statement-id	Sets the user environment to the given location. Usually, you just specify a routine-name, but sometimes you specify a particular statement-number within the routine. When dealing with recursive routines, you may also want to specify an activation-number. (See the "Routine-Names" and "Statement-ID" listings of this encyclopedia for details.)
-Run	Sets the current environment to the run environment.
-User	Sets the current environment to the most recently defined user environment.
-Caller	Sets the user environment to the statement that called the current routine. The current routine must be active. This option and the -SUB option provide a convenient way to "walk" up and down the call stack.
-Sub	Sets the user environment to the routine called by the current routine. The current routine must be active. This option is essentially the opposite of -Caller.

DESCRIPTION

Use the **ENVIRONMENT** command to set or describe an environment. You can optionally supply one argument to the command. An environment is really just a particular region of source code. DEBUG actually differentiates between two kinds of environments: the run environment and the user environment.

The **run environment** is the routine and statement where execution stopped. Execution can be stopped by a breakpoint, a **STEP** command, or a fault. Ordinarily, the program also resumes execution at the run environment, though you can use a GO -Location command to resume execution somewhere else.

The **user environment** is the routine and statement that you set with an **ENVIRONMENT** command. You set a user environment to facilitate debugging operations outside the run environment. Suppose, for example, you want to set a breakpoint in a routine stored in a file other than the current one. You have two choices. Either you could fully qualify the routine's name, or you could set the user environment to that routine and then set breakpoints without fully qualifying the routine name.

The **current environment** is the routine and statement that DEBUG is currently using in its calculations. The current environment is always either the run environment or the user environment.

Whenever the program stops, DEBUG sets the current environment to the run environment. Whenever you define a user environment, it automatically becomes the current environment. You can set the current environment equal to either the run environment or the user environment by using the **ENVIRONMENT** command. DEBUG displays the source code surrounding the current environment.

Many debugger operations implicitly reference the current environment. For example, if you ask to examine variable X, and don't qualify X with a particular routine name, the debugger shows you the X visible from the current environment. If X is not visible from the current environment, for instance, if it is defined in another routine, then DEBUG will not be able to find it.

In review, DEBUG understands the following three kinds of environments:

user environment	The user environment is the statement–ID that you set with the **ENVIRONMENT** command.
run environment	The run environment is the statement–ID at which the program is stopped.
current environment	The current environment is the environment that DEBUG is currently using; it may be either the run environment or the user environment.

There are a few points worth noting about the **ENVIRONMENT** command:

- The **ENVIRONMENT** command has no effect on the execution of the target program. Although a user environment overrides the run environment for source display and name references, the run environment remains defined and unchanged.
- If you are using source–display debugging (–SRC), then setting a user environment causes DEBUG to display the specified user environment. Specifying a particular line number within a routine affects only the positioning of the source code display.
- The specified routine does not have to be active. However, if a routine is inactive, you can only access the static storage in it.
- Normally you cannot set the user environment to a routine compiled without debug information. However, the –CALLER and –SUB options set up a partial environment in these cases. You cannot examine variables, etc. in these routines but at least you can move over them in the stack.

EXAMPLE 1

Given a program consisting of two routines and three static variables organized as follows:

```
PROGRAM "TEST"

GLOBAL VARIABLE "G"

ROUTINE "P"
   STATIC VARIABLE "LP"

ROUTINE "M"
   STATIC VARIABLE "LM"
```

The following sample DEBUG session demonstrates the ENVIRONMENT command:

```
> ENVIRONMENT                          (Display the current environment.)
Stopped at: TEST\m\16                  (It's routine m.)
> EXAMINE g, lm                        (We can access g and lm, but...)
TEST\g = 0
TEST\main\lm = 4
> EXAMINE lp                           (we can't access lp since it's in)
*** Error: Unknown variable name.      (another environment.)
ex lp
---^
>
> ENVIRONMENT p\                       (However, if we change the environment...)
Stopped at: TEST\main\16
User environment: TEST\p\7 (inactive routine)
> EXAMINE g, lp                        (we can access g and lp, but...)
TEST\g = 0
TEST\p\lp = 10
> EXAMINE lm                           (we can't access lm.)
*** Error: Unknown variable name.
ex lm
---^
```

EXAMPLE 2

Now let's consider a second program consisting of two files of source code organized as follows:

```
FILE "ONE"

ROUTINE "X"
    lines 5-30


ROUTINE "Y"
    lines 35-76
```

```
FILE "TWO"

ROUTINE "Z"
    lines 4-150
```

Suppose that the current user environment is ROUTINE "X", but you want to set a breakpoint at line 55 of ROUTINE "Z". If we issue the following statement, DEBUG sets a breakpoint, but at line 55 of FILE "ONE":

```
> BREAKPOINT 55
```

We can set a breakpoint in "Z" by specifying a routine–name in front of the statement–id:

```
> BREAKPOINT Z\55
```

or, we can change the user environment to routine Y2 as follows:

```
> ENVIRONMENT Z\
```

and then set one or more breakpoints without having to specify a routine–name; for example:

```
> BREAKPOINT 55
```

EXAMINE -- Displays the value or values of variables, arrays, or other data structures.

FORMAT

```
                                    -Ascii
                                    -Binary
                                    -Decimal
                                    -Hex
Examine  variable1, ... variableN   -Octal
                                    -Unsigned
                                    -Float
                                    -Real
                                    -DOUble
```

REQUIRED ARGUMENT

variable	One or more program variables (of any data type) or debugger variables.

OPTIONAL ARGUMENTS

Each of the following optional arguments causes **EXAMINE** to ignore the data types of all variables in the list and to display the data in a specific way.

-Ascii	Displays the data as one or more ASCII characters. (An ASCII value takes up 8 bits, so a 32-bit variable is translated as 4 consecutive ASCII values.)
-Binary	Displays the data in binary (base 2) format.
-Decimal	Displays the data in decimal (base 10) format.
-Hex	Displays the data in hexadecimal (base 16) format.
-Octal	Displays the data in octal (base 8) integral format.
-Unsigned	Displays the data in unsigned decimal format.
-Float	Displays the data in 32-bit (single-precision) floating-point format.
-Real	Identical to **-Float.**
-DOUble	Displays the data in 64-bit (double-precision) floating-point format.

Note that data is never converted (cast).

DESCRIPTION

Use the **EXAMINE** command to display the current value of one or more variables. The variables can be of any data type. If you specify a compound variable (such as an array, structure, or record) **EXAMINE** displays the value of each component of the variable.

You can optionally specify one formatting option. The option may precede or follow the list of variables but cannot be placed in the middle of the list.

The **PRINT** command provides you with an alternative way to display the contents of a simple variable. Refer to the "PRINT" listing for more information. You should also refer to the "Arrays", "Pointers", and "Variables" listings for more information on specifying variables in **EXAMINE** commands.

Examining Simple Variables

It is easy to display the value of a simple variable; for example, to examine the value of a floating-point variable named pi, you would merely enter the following command:

```
> EXAMINE pi
TEST_c\pi = 3.14
```

Note that specifying an integral option such as -DECIMAL does not *convert* 3.14 to a decimal integer, DEBUG only interprets pi's bits as a decimal integer, leading to bizarre results.

Examining Pointer Variables

Before attempting to examine or de-reference a pointer variable, you should read the "Pointer Variables" listing later in this encyclopedia.

Examining Records, Structures, and Unions

Specifying a record, structure, or union variable causes **EXAMINE** to display each member (field) of the variable. For example, the display of a variable of the standard type status_$t would show:

```
> EXAMINE statrec
TEST\statrec.FAIL = .FALSE.
TEST\statrec.SUBSYS = 1
TEST\statrec.MODC = 2
TEST\statrec.CODE = 14
TEST\statrec.ALL = 16908302
```

Should you wish to examine only one field (member) of such a variable, specify the name of the variable followed by a period and the name of the field; for example:

```
> EXAMINE statrec.modc
TEST\statrec.modc = 2
```

Examining Arrays

If one of the variables given is an array, EXAMINE displays the array's elements one by one. For example, EXAMINE displays an array of dimensions 3 x 2 as:

```
> EXAMINE cambridge
PROG1\CAMBRIDGE(1,1) = 100
PROG1\CAMBRIDGE(1,2) = 150
PROG1\CAMBRIDGE(2,1) = 75
PROG1\CAMBRIDGE(2,2) = 125
PROG1\CAMBRIDGE(3,2) = 50
PROG1\CAMBRIDGE(3,2) = 100
```

NOTE: The first element of C arrays is labeled 0 rather than 1, so DEBUG displays the contents of C arrays beginning from element 0. For all languages, DEBUG displays arrays with the same subscript range as the source code defined for the array.

You can specify a portion of an array in the many ways explained in the "Arrays" listing earlier in this encyclopedia.

Reducing the Length of Variable Names in Output

Before issuing **EXAMINE**, you can set the debugger variable **`max_var_len** so that EXAMINE displays only a certain number of characters in variable names. For example, consider the affect of **`max_var_len** on the following array:

```
> EXAMINE board
C_FRAME\board(1,1) =
C_FRAME\board(1,2) =
C_FRAME\board(1,3) =
C_FRAME\board(2,1) =
C_FRAME\board(2,2) =
C_FRAME\board(2,3) =
C_FRAME\board(3,1) =
C_FRAME\board(3,2) =

> SET `max_var_len = 4
>
> EXAMINE board
 ...1,1) =
 ...1,2) =
 ...1,3) =
 ...2,1) =
 ...2,2) =
 ...2,3) =
 ...3,1) =
 ...3,2) =
```

The debugger variable **`max_qual** also reduces the length of the variable–name that DEBUG displays. See the "Debugger Variables" listing earlier in this encyclopedia for details.

Note To FORTRAN Users

When you are in a FORTRAN subroutine with alternate entry points, do not attempt to use **EXAMINE** with arguments that are not valid for that entry. If you try to access an invalid argument, you will get bad data and may cause an access violation or odd address fault.

EXIT -- Ends a debugger session and terminates the target program.

FORMAT

EXIt

REQUIRED ARGUMENTS

None.

OPTIONAL ARGUMENTS

None.

DESCRIPTION

Use this command to quit the debugger session. The **EXIT** command has the same effect as the **QUIT** command. See the "QUIT" listing for more information.

Expressions -- Some DEBUG commands accept expressions as arguments. We describe expressions here.

DESCRIPTION

Several DEBUG commands accept expressions as arguments. DEBUG has its own expression syntax, which draws from Pascal, C and FORTRAN, but is not identical to any of them. The same expression syntax is accepted regardless of the source language of the program being debugged.

A DEBUG expression can consist of variables, constants, and operators. In this listing, we describe constants and operators. See the "Variables" listing for a description of variables.

Integer Constants

Decimal integer constants are written in the usual way, as a string of one or more decimal digits optionally preceded by a sign. Integers may be written in base 2–16 using the <base>#<value> notation. The base specification is always given in decimal. For example, all of the following are equivalent:

```
27
16#1B
8#33
2#11011
```

DEBUG treats integer constants as signed 32 bit integers. DEBUG accepts values in the range 2^{31} through $2^{32} - 1$. Such values have the proper unsigned integer internal representation, but DEBUG treats them as negative for computation and printing.

Floating–Point Constants

DEBUG uses Pascal notation for floating–point constants. In particular, at least one digit must precede and follow the decimal point. DEBUG treats all floating–point constants as double precision.

String (and Character) Constants

You must enclose string constants in either single or double quotes. You can designate quotes in the string by doubling the character, although using the other type of quote is usually more convenient. For example, the following are equivalent:

```
'"Don''t" he cried'
"""Don't"" he cried"
```

String and character constants can only consist of printable characters; C '\' escapes are not accepted. There is no separate notation for character constants. DEBUG converts a string of length 1 to a character in appropriate contexts.

Boolean (Logical) Constants

You must write boolean constants by using the FORTRAN–style names .TRUE. and .FALSE. (uppercase or lowercase).

Complex Constants

To represent complex constants, use the format

```
(x,y)
```

where x and y are both floating–point constants.

Pascal Set Constants

To write Pascal set constants, create a list of values enclosed in square brackets, for example:

```
[2, 4, 6, 8]
[x+1, y, 100-x]
```

Each component's value must evaluate to an integer in the range 0–255. (Note that enumerated type constants evaluate to integers for this purpose.) You cannot use Pascal's subrange notation. For example, compare the right and wrong ways to represent 2 through 8:

```
[2..8]              (WRONG)
[2,3,4,5,6,7,8]     (RIGHT)
```

Operators

DEBUG expressions can include any of the operators shown in Table 3–1. Operators listed on the same line are identical.

Table 3–1. DEBUG Operators

Function	DEBUG Operators
add	+
subtract	–
multiply	*
divide	/ DIV
modulo	% MOD
equal	== = .EQ.
not equal	<> != .NE.
less than or equal	<= .LE.
less than	< .LT.
greater than or equal	>= .GE.
greater than	> .GT.
AND	&& & .AND. AND
OR	! || | .OR. OR
negation, NOT	–
left shift	<<
right shift	>>
set membership	IN

All operators in DEBUG expressions have equal priority. Therefore, if you do not place any parentheses in an expression, DEBUG evaluates the expression in strict left–to–right order. Use parentheses to force a different order of evaluation.

DEBUG permits you to mix values of different data types in expressions. If you mix integer and real values in an expression, DEBUG converts the integer value to real before evaluating the expression. In expressions, DEBUG treats pointers and enumerated variables as ordinary integers. Therefore, you can use pointers and enumerated variables anywhere that you would use an integer. C programmers should note that DEBUG does not scale by the size of the objects when performing pointer arithmetic. However, see the "Pointers" listing for a way around this.

The operators / and DIV are equivalent. Both operate like FORTRAN division, that is, if both operands are integers, the result is an integer. Otherwise, the result is a real number.

Applied to integers, all forms of the AND and OR operators work bitwise and produce integer results. Applied to logical operands, the operators generate the correct boolean result. However, they do not "short circuit" like the C operators && and ||.

The unary minus operator (–) acts as a logical NOT operator when applied to a boolean or logical value.

Faults -- This listing explains how DEBUG handles faults.

When a fault occurs in the target process, DEBUG intercepts it, suspends program execution, and displays a message describing the fault. If you type **GO** or **STEP**, DEBUG *attempts* to continue execution from the point of the fault, but the attempt will not always be successful. The –Location option of the **GO** command may be useful in skipping over the faulting statement, or in restarting the program at the faulting statement if you were able to repair the cause of the fault. Note that DEBUG warns you if the operating system had marked the fault unrestartable.

You can send the intercepted fault to whatever fault handling routine the target program has established by entering the command:

```
> GO -Cleanup
```

If you want to debug a fault handler or cleanup handler, be sure to set a breakpoint on it before resuming execution with GO -Cleanup.

The macro name **`fault_action** has a special meaning to DEBUG. If you define a macro by this name, DEBUG executes the associated action–list whenever a target program fault occurs. The predefined debugger variable **`fault_status** is set to the status code of the most recent fault. For example, the following macro automatically resumes program execution if a floating–point overflow fault occurs:

```
> MACRO `fault_action [IF `fault_status = 16#00120026 [GO]]
```

Causing Faults

You can send asynchronous faults to an executing program with

- Shell command SIGP *or*
- CTRL–Q

DEBUG ignores faults sent to the target while it is suspended. However, you can simulate an asynchronous fault at the point where the target is suspended with the **SIGNAL** command. Using the **SIGNAL** command allows you to cause a fault at a precise point of execution. For example, to debug a fault handler, you would first set a breakpoint on it, run or step the target to the point where you want the fault to occur, enter a **SIGNAL** command to simulate the fault, and then type GO –Cleanup to restart the target and invoke the fault handler.

Faults in DEBUG

Faults can occur in DEBUG itself if you attempt to access data via a bad pointer or outside of array bounds, and in other similar situations. A fault in DEBUG aborts execution of the current command and returns you to the command prompt level (except for a stop fault, which terminates execution.) Note that the message describing a fault is prefixed with "(debug)" for faults in DEBUG, and with the target process name for faults in the target.

When the debugger and target are running in the same window (whether or not DEBUG has its own window pane), a CTRL–Q will cause a fault in both processes. DEBUG must then decide whether the CTRL–Q was meant for DEBUG or for the target program. Here are the criteria that DEBUG uses to decide:

- If the target program was running when the fault occurred, DEBUG assumes that the intention was to stop the target. Therefore, DEBUG ignores its own quit fault.

- If the target program was not running when the fault occurred, DEBUG assumes that the fault was intended for itself. As noted above, the fault is ignored in the target process.

GO -- Begins or resumes execution of the target program.

FORMAT

	Zero or one from this group.	Zero or one from this group.	Zero, one, or more from this group.
Go	*-Location statement-ID* *-Exit* *-Return (expr) -Alt integer* *-Cleanup*	*-After integer*	*-Until statement-ID -OR statement-ID*

REQUIRED ARGUMENTS

None. If you enter GO with no arguments, DEBUG resumes execution from the statement where the program is currently stopped.

OPTIONAL ARGUMENTS

-Location Causes your program to resume execution at the specified statement. The specified statement *must* be in the run environment routine. You cannot use -Location to jump to a different routine.

-Exit Sets one-shot breakpoints on all exits of the current routine thus allowing you to examine the state of the routine's variables just before control returns to the caller. Note that an "exit" is a *return* to the calling routine, an "exit" is *not* a call to another routine.

-Return Causes the current routine to exit immediately. The optional **(expr)** argument defines a return value if the routine is a function. You must enclose the **(expr)** argument in parentheses. The phrase **-Alt integer** is optional, and you can only use it when debugging FORTRAN programs. Use it to specify an alternate routine return. Note that **(expr)** and **-Alt** are mutually exclusive, since the former applies only to functions and the latter only to subroutines.

-Cleanup This option is valid only after the target program has stopped for a fault (other than a normal breakpoint or single-step.) It causes the fault to be delivered to the target program, so that any cleanup handlers are executed. See the "Faults" listing earlier in this encyclopedia for a complete discussion of fault handling.

-After Applies only if the target program is currently stopped at a breakpoint. If it is, the current break will not occur again for the number of passes through the statement that you specify with "integer." The action of this option is identical to that of the -After option of the **BREAKPOINT** command. This option replaces any existing after count on the breakpoint.

-Until, -OR Sets one or more one-shot breakpoints at the given statement-IDs. -Until and -OR produce identical results.

DESCRIPTION

Use the GO command to begin or resume execution of the target program. All options are mutually exclusive except for -After and -Until, which can be combined with any of the others.

Several of GO's arguments establish one-shot breakpoints. A **one-shot breakpoint**, like all other breakpoints, stops the program *before* the statement executes. A one-shot breakpoint remains in effect until the program stops for *any* reason. As soon as the program stops, DEBUG deletes the one-shot breakpoint. A one-shot breakpoint replaces any existing breakpoint at the same statement. Therefore, after you set a one-shot breakpoint, you lose any permanent breakpoint set at the same statement.

NOTE: Because of program optimization, it is possible that the GO -Location command may cause unexpected results. The compiler optimizes code by assuming that the program executes in a certain order. By using the GO -Location command, you depart from this expected order and possibly cause program bugs. The -DBA compiler option prevents the optimizations that can lead to unexpected results when you debug. (See Appendix C for complete details on optimization and debugging.)

EXAMPLES

Here are several sample **GO** statements:

`> GO`	*(Begin execution from current line.)*
`> GO -LOCATION 27`	*(Jump to the statement at line 27, and begin execution. Line 27 must be in the current routine.)*
`> GO -EXIT`	*(Begin execution at current line, and set temporary breakpoints at all routine exits.)*
`> GO -RETURN`	*(Return immediately to calling routine.)*
`> GO -RETURN (5)`	*(Return immediately from a function and set the function's value to 5.)*
`> GO -RETURN -ALT 2`	*(Return to FORTRAN's second alternate return point.)*
`> GO -AFTER 3`	*(Set a temporary breakpoint at the current line. The breakpoint will be activated on the third, sixth, ninth, etc. times that the program reaches this line.)*
`> GO -UNTIL 40`	*(Set a one-shot breakpoint at line 40 and then begin program execution from the current line.)*
`> GO -UNTIL 40 -OR move\6`	*(Set two one-shot breakpoints, then begin program execution from the current line.)*

HELP -- Displays information about DEBUG commands.

FORMAT

Help *command -Verbose*

REQUIRED ARGUMENTS

None.	If you specify **HELP** with no arguments, DEBUG displays a list of all the DEBUG commands.

OPTIONAL ARGUMENT

command	Any DEBUG command name or a valid command abbreviation. Previous revisions of DEBUG permitted you to put an optional dash prior to the command, but SR9.5 DEBUG no longer permits the dash.
-Verbose	Provides you with a longer, more-detailed help file.

DESCRIPTION

Use **HELP** to display an on-line description of a particular DEBUG command, or a general summary of all available DEBUG commands. If you do not use the -Verbose option, DEBUG displays a brief file containing one or more examples of the command. If you do use the -Verbose option, DEBUG displays a format-oriented description of the command.

EXAMPLES

`> HELP`	*(DEBUG displays a complete list of all available DEBUG commands.)*
`> HELP BREAKPOINT`	*(DEBUG displays several examples of the BREAKPOINT command.)*
`> HELP B`	*(Equivalent to HELP BREAKPOINT.)*
`> HELP BREAKPOINT -Verbose`	*(DEBUG displays a detailed description of the syntax of the BREAKPOINT command.)*

IF -- Evaluates a given expression, and executes an action-list conditionally.

FORMAT

IF expression action-list1 *-Else action-list2*

REQUIRED ARGUMENTS

expression	Any legal expression that produces a boolean (logical) or integral result. See the "Expressions" listing earlier in this encyclopedia for details.
action-list1	An action-list. See the "Action-Lists" listing earlier in this encyclopedia for a definition of action-list. If **expression** is boolean and evaluates to true, or if **expression** is arithmetic and evaluates to nonzero, then DEBUG executes action-list1.

OPTIONAL ARGUMENT

-Else action-list2	An action-list. If the expression is boolean and evaluates to false, or if the expression is arithmetic and evaluates to zero, then DEBUG executes action-list2.

DESCRIPTION

Use the **IF** command to conditionally execute one or more DEBUG commands stored in an action-list.

EXAMPLE

Here are four **IF** command examples:

```
> IF x > 0 [EXAMINE x]

> IF x > 0 [EXAMINE x] -ELSE [EXAMINE y]

> IF ((x > 0) AND (q < p)) [DELETE -B -All; BREAKPOINT f1\; GO]

> IF flag [PRINT 'flag is true']
```

For our final example we consider an **IF** statement used in the action-list of a **BREAKPOINT** command:

```
> BREAKPOINT 24 -DO [IF (status.all .eq. 0) [GO]
_                    -ELSE [EXAMINE status] ]
```

The preceding command would be particularly useful for testing the success or failure of a system call. Basically, it tells the debugger to GO through the breakpoint if the system call succeeded, but to halt and EXAMINE the error code if the system call failed.

JUMP -- Moves between commands in an action-list.

FORMAT

Jump label

REQUIRED ARGUMENT

label	A command label. The label must appear within the same command action-list as the **JUMP** command. A label is an identifier consisting of one or more alphanumeric characters followed by a colon (:). The first character in a label must be a letter; it cannot be a digit.

OPTIONAL ARGUMENTS

None.

DESCRIPTION

You can only use **JUMP** within an action-list. **JUMP** works like a goto statement. Use **JUMP** to move from one portion of an action-list to another.

In the action-list, you can specify the label before or after the corresponding **JUMP** command; that is, you can jump forward or backward.

DEBUG looks for the label using a simple string matching search. Therefore, be careful that you do not pick a label that conflicts with another part of the line. If you have a program variable called `label`, for example, a line like the following would loop incorrectly:

```
_ [SET label:= 5;
_      label: PRINT 'Want loop here';
_      JUMP label]
```

DEBUG would jump to the first "label:", rather than the second.

EXAMPLE

In this example we set a label called `loop:` within a macro's action-list. The following macro prints the contents of a linked list. Notice how the macro *borrows* the program variable `head` to walk the list. A debugger variable would not have the correct type.

```
> MACRO `print_list [
_ SET `save_head = head;
_ loop:
_       EXAMINE head^.car;
_       SET head = head^.cdr;
_       IF head <> 0 [jump loop];
_ SET head = `save_head ]
```

You do not have to put each statement in the action-list on a different line or indent the command's in the loop, but it does help readability.

LIST -- Produces a list of program routines, program variables, DEBUG definitions, DEBUG macros, breakpoints, or alternate search directories.

FORMAT

```
        -Macro     macro-name
        -DEFine  definition-name
        -Breakpoint   statement-ID
List    -Routines
        -VARiables  routine-name
        -DV
        -SDIR
```

REQUIRED ARGUMENTS

None.	If you specify **LIST** with no arguments, DEBUG returns a list of all macros, definitions, breakpoints, routines, program variables, DEBUG variables, and alternate search directories.

OPTIONAL ARGUMENTS

-Macro Lists the names and definitions of DEBUG macros. A macro is something you create with the **MACRO** command. Use this option in one of the following two ways:

List -Macro	Lists the names and definitions of all macros.
List -Macro macro-name	Lists the definition for this particular macro.

-DEFine Lists the names and definitions of DEBUG definitions. A DEBUG definition is something you create with the **DEFINE** command. Use this option in one of the following two ways:

List -Define	Lists the names and definitions of all DEBUG definitions.
List -Define definition-name	Lists the definition for this particular DEBUG definition.

-Breakpoint Lists information about breakpoints. Use this option in one of the following three ways:

List -Breakpoint	Describes every breakpoint in the program.
List -Breakpoint routine-name	Describes every breakpoint in that routine.
List -Breakpoint statement-id	Describes the breakpoint at that line number.

-Routines Lists all routines in the program (including the startup routines provided by the compiler). If you compiled the routine with either the -DBS or the -DBA compiler option, DEBUG prints the message "(symbol table available)" next to the routine-name. If this message does not appear, then you cannot access the variables in this routine. DEBUG lists nested routines by indenting them under their parent routine or module.

-VARiables Lists the names of program variables. Use this option in one of the following two ways:

List -Variables	Lists the names of all program variables in the current routine.
List -Variables routine-name	Lists the names of all program variables in the specified routine.

-DV Lists all active DEBUG variables. You activate DEBUG variables with the **SET** command.

-SDIR Lists the current alternate source file pathname(s) specified by the **SOURCE** command or the -SDIR option.

DESCRIPTION

Use the **LIST** command to identify the components of a DEBUG session. You can issue the **LIST** command without any arguments or with one or more arguments. If you issue the **LIST** command without arguments, DEBUG displays information on all components. Specifying **LIST** with one or more arguments limits the information to particular components.

EXAMPLES

The following example shows typical output of the LIST -BREAKPOINT command:

```
> LIST -BREAKPOINT              (Lists every breakpoint in the program.)
Breakpoints:
    SAMPLE\22
        -AFTER 4                (See the -After option of BREAKPOINT.)
        current counter = 0
    SAMPLE\subr3\12
        -AFTER 1  -DO [PRINT x]    (An action-list at this breakpoint.)
        current counter = 0
```

Using the -ROUTINES option provides a list of every routine in the program. The following example shows that the program consists of four user-written routines divided into two modules (MAIN and MANIPULATE_MASTER_FILE):

```
> LIST -ROUTINES                (Lists every routine in the program.)
Routines:
    <apollo_c_startup>   (symbol table available)
    MAIN_C    (symbol table available)
      main    (symbol table available)
      <apollo_c_startup>   (symbol table available)
    MANIPULATE_MASTER_FILE_C    (symbol table available)
      open_master_file   (symbol table available)
      initialize_master_file    (symbol table available)
      search_for_correct_program    (symbol table available)
```

Here are two examples of the LIST -VARIABLES command:

```
> LIST -VARIABLES               (Lists program variables in current routine.)
  Variable(s):
      XR
      K
      J
      I
> LIST -VARIABLES subr3\        (Lists program variables in routine subr3.)
  Variable(s):
      QUARK
      STRANGE
```

MACRO -- Defines a sequence of DEBUG commands that you can invoke using a debugger-name.

FORMAT

Macro debugger-name action-list

REQUIRED ARGUMENTS

debugger-name	Any legal DEBUG name. All such names must begin with an accent grave (`).
action-list	Any legal DEBUG action-list. See the "Action-Lists" listing earlier in this encyclopedia for details.

OPTIONAL ARGUMENTS

None.

DESCRIPTION

Use the **MACRO** command to create a macro (i.e., a script) of DEBUG commands. For instance, you might create a macro that examines the values of several variables. Such a macro might be defined as follows:

```
> MACRO `m1 [EXAMINE ponce; PRINT ponts; BREAKPOINT r3\]
```

You can invoke a macro anywhere you could use a normal DEBUG command. To invoke the macro, simply specify its name, for instance:

```
> `m1
```

Note that you cannot abbreviate a macro name. Also, don't forget that a macro name must begin with an accent grave (`). The name you choose cannot duplicate that of an existing debugger variable or debugger definition, though it can be the name of an existing macro. By picking the name of an existing macro, you delete the old definition.

DEBUG does not check macro definitions for illegal commands until you actually invoke the macro.

You can use macros as part of other macro definitions, but you cannot create a recursive macro (i.e., a macro that calls itself).

By default, macro definitions disappear when the DEBUG session ends. (Such a pity when you've written a long macro.) Therefore, you might consider storing the macro definitions in one of the startup files described in Section 3.1.

The Special `CR Macro

When you press the RETURN key on a blank DEBUG input line, DEBUG automatically invokes a special macro named `CR, if it is defined. (By default, `CR is not defined.) The most common use of `CR is to cause DEBUG to STEP when you input a blank line. To accomplish that, simply define the following macro:

```
> MACRO `CR [STEP]
```

DEBUG Definitions in Macro Definitions

Suppose you specify a DEBUG definition in the action–list of a macro definition. Does DEBUG substitute the definition's value when you define the macro or when you invoke the macro? For most definitions, DEBUG substitutes the value when you invoke the macro. The predefined symbol

is an exception to this rule. This symbol designates the current routine name that appears in macro definitions. For example, suppose you make the following macro definition while the current routine is $MAIN\:

```
> MACRO `ezra [EX `.pound]
```

Entering the DEBUG symbol `` `ezra `` as a DEBUG command, even while the program is executing a routine other than $MAIN, is exactly equivalent to entering:

```
>EX $main\pound
```

The `FAULT_ACTION Macro

If you define a macro named `` `FAULT_ACTION ``, DEBUG automatically invokes it when the target program faults. See the "Faults" listing for an example.

Pointer Variables -- Here, we explain how to use pointer variables during DEBUG sessions.

You can use DEBUG to reference and de-reference pointer variables. For example, assume that pstr is a pointer variable that points to a character variable. Consider how we reference and de-reference `pstr`:

```
> EXAMINE pstr      (Find the address that pstr points to.)
pstr = 000101C8
> EXAMINE pstr^     (Find the contents at the address that pstr points to.)
pstr^ = h
```

Notice that we must use the Pascal de-referencing symbol ^ because the C de-referencing symbol * is not available for debugging. C programmers can use ^ wherever you would have used *. Also, C programmers should read Section 4.2.5 in Chapter 4 for full details on C pointers in DEBUG sessions.

DEBUG permits you to freely interchange pointers and integers; for example, in a DEBUG expression, you can substitute a pointer anywhere that an integer is permitted. See the "Expressions" listing for more details.

De-referencing a Pointer to a Function or Procedure

You can de-reference a pointer to a function or procedure. When you de-reference such a pointer, DEBUG returns a character string representing the name of the routine. You can **EXAMINE** or **PRINT** the string, but you cannot change its value.

De-referencing a Virtual Address

You can de-reference any legal virtual address. To do so, use the syntax

address^data-type

where address is the virtual address and data-type is any legal DEBUG data type shown in Table 3-2. DEBUG de-references the address and interpret the results as having the given data-type. For example, the following command displays two bytes of data at the absolute address 3A8180 (hex) as an integer:

```
> EXAMINE 16#3A8180^integer16
```

To find the virtual address of a variable, use the VA command.

Table 3-2. Legal DEBUG Data Types

integer	float	boolean	complex	univ_ptr	char	
integer16	real	logical			byte	8-bit integers
integer32	double				int8	8-bit integers

Pascal UNIV_PTRs

DEBUG allows you to de-reference a variable of the UNIV pointer type (UNIV_PTR) even though the Pascal language does not. To de-reference such a variable, use the same syntax you would use for de-referencing a virtual address. For example, suppose that a variable named `pointless` has the

UNIV_PTR data type. The following SET command causes DEBUG to de-reference pointless and to reset its value to 1000498:

```
> SET pointless^INTEGER32 = 1000498
```

FORTRAN Pointers

You can examine and set FORTRAN pointers in a similar way. However, the address to which a FORTRAN pointer refers must have variable names. You can use these variable names directly, without explicitly de-referencing the pointer. For example, the following is a possible FORTRAN pointer definition:

```
INTEGER*4 mirror POINTER /mirror/ alice, carroll, dodgson
```

Once you define the pointer, mirror, in the program, the variables alice, carroll, and dodgson have locations and values. You can then treat the variables as ordinary variables. You can also treat mirror in the same manner as any variable, except that changing its value changes the locations of alice, carroll, and dodgson. For example, all of the following DEBUG commands are legal:

```
SET carroll
EXAMINE mirror
EXAMINE alice
PRINT carroll + alice
DEFINE `look [P mirror]
DESCRIBE alice, carroll, mirror, dodgson
```

NOTE: You cannot explicitly de-reference a FORTRAN pointer.

PRINT -- Displays the values of one or more expressions.

FORMAT

Print	**Zero or one from this column:** *-Ascii* *-Binary* *-Decimal* *-Hex* *-Octal* *-Unsigned* *-FLoat* *-Real* *-DOUble*	*expression1, ... expressionN*

REQUIRED ARGUMENTS

None.	If you issue the PRINT command with no arguments, DEBUG prints a blank line.

OPTIONAL ARGUMENTS

You can optionally supply one formatting option from the following list. Each formatting option causes **PRINT** to ignore the data types of all variables in the list and to display the data in a specific way.

-Ascii	Displays the data as if it were one or more ASCII values. (An ASCII value takes up 8 bits, so a 32-bit variable will be translated into 4 ASCII values.)
-Binary	Displays the data in binary (base 2) integral value.
-Decimal	Displays the data in decimal (base 10) integral value.
-Hex	Displays the data in hexadecimal (base 16) integral value.
-Octal	Displays the data in octal (base 8) integral value.
-Unsigned	Displays the data in unsigned decimal integral value.
-Float	Displays the data in 32-bit (single-precision) floating-point value.
-Real	Identical to **-Float.**
-DOUble	Displays the data in 64-bit (double-precision) floating-point value.

expression	One or more legal DEBUG expressions. (See the "Expressions" listing earlier in this encyclopedia for details.) If you specified a formatting option, you must specify at least one expression.

DESCRIPTION

Use the **PRINT** command to display the values of one or more expressions. The **PRINT** command is similar to the **EXAMINE** command except for the following differences:

- **EXAMINE** can only display the values of variables; **PRINT** can display the values of any expression (including a variable by itself).

- The **PRINT** command can only show the value of individually named array or record elements. It cannot show the contents of each element of a record or array when given the aggregate name. The one exception is that **PRINT** can display a string; in fact, it prints it on one line. In contrast, **EXAMINE** displays each element of an array of characters on a separate line. **PRINT** can only display an entire string; you cannot use a subrange specification. For C strings, **PRINT** prints characters up until it reaches the string termination character \0.

- The **EXAMINE** command displays the name and value of the variable; the **PRINT** command displays only the value of the variable.

- The **PRINT** and **EXAMINE** commands accept the same data type options (–ASCII, –BINARY, etc.); however, **EXAMINE** accepts them before or after the variable, and **PRINT** only accepts them *before* the expression.

- The **PRINT** command displays all expression values on the same line with no spaces between them. In contrast, the **EXAMINE** command displays each variable on a separate line.

EXAMPLES

Here are some examples that distinguish **PRINT** from **EXAMINE**:

```
> EXAMINE letter                      (EXAMINE reports variable's name...)
TEST_C\letter = Z

> PRINT letter                        ( but PRINT does not.)
Z

> PRINT –Decimal letter               (Display value as a decimal number.)
90

> PRINT ´The value is ´, letter       (A string is a valid expression.)
The value of letter is Z

> EXAMINE stu                         (EXAMINE can display the value of
TEST_C\stu.x = b                       an aggregate type like a record
TEST_C\stu.y = 325432                  or a structure...)

> PRINT stu                           (but a PRINT cannot.)
*** Error: Cannot have aggregate reference here.
PRINT stu
------^

> EXAMINE carray                      (EXAMINE displays every value, even
TEST_C\carray[0] = M                   the meaningless ones, of a
TEST_C\carray[1] = a                   character array.)
TEST_C\carray[2] = r
TEST_C\carray[3] = y
TEST_C\carray[4] = <00>
TEST_C\carray[5] = <00>

> PRINT carray                  (PRINT only displays the significant part of  a C string.
Mary                             When you debug FORTRAN or Pascal programs,
                                 DEBUG prints the entire array unless you specify
                                 `max_bad_chars .)
```

QUIT -- Ends a debugger session and terminates the target program.

FORMAT

Quit

REQUIRED ARGUMENTS

None.

OPTIONAL ARGUMENTS

None.

DESCRIPTION

Use this command to get out of the debugger. The **QUIT** command has the same effect as the **EXIT** command.

This command forces the target program to quit, then it forces DEBUG to quit. The target exits as if it had called `pfm_$exit`. Therefore, if you set cleanup handlers in the program, DEBUG invokes them as the program exits. If the target program has inhibited faults when this command is issued, DEBUG may not be able to force the target program to exit. In such a case, DEBUG exits and leaves the target program running.

READ -- Reads and executes a file of DEBUG commands.

FORMAT

Read pathname

REQUIRED ARGUMENTS

pathname	The pathname of a file containing DEBUG commands.

OPTIONAL ARGUMENTS

None.

DESCRIPTION

A **READ** command is similar to a Shell script. When you issue the **READ** command, DEBUG reads the specified file and executes each line in the file as a DEBUG command. DEBUG echoes each line of the input file as it reads the line.

When you issue DEBUG commands from a command file, DEBUG defers the processing of **GO** and **STEP** commands until all other commands in the file have been processed. For example, consider the following sample command file:

```
SET `max_var_len = 66
BREAKPOINT 262
GO
EXAMINE try
```

When DEBUG processes this file, it defers the execution of the third command, **GO**, until last. The **EXAMINE** command, therefore, will execute before the **GO** command. To avoid confusion, you should only put a **GO** or **STEP** command at the end of the file.

Command files can themselves contain **READ** commands.

EXAMPLE

Suppose you store the following information in a file named debugging_my_prog:

```
PRINT r
BREAKPOINT rout3\
GO
```

To invoke these commands, simply enter the following command:

```
> READ debugging_my_prog
PRINT r                       (DEBUG echoes the command here...)
10388142
BREAKPOINT rout3\             ( and here...)
GO                            ( and here.)
Break at: TEST_C\rout3\10
```

Routine-Name -- The name of a subroutine, procedure, or function.

FORMAT

A routine-name is not a command, but many DEBUG commands accept a routine-name as an argument. When you specify a routine-name, you must give it the following format:

routine-identifier\routine-identifier\\...activation-number\

REQUIRED

routine-identifier	One or more program, module, subroutine, procedure, or function names separated by backslashes.

OPTIONAL

activation-number\\	An integer that indicates a specific activation if the routine is on the call stack more than once. If the activation-number is positive, it refers to an absolute activation-number from the first activation of the routine. If the activation-number is negative, you subtract it from the most recent activation, and the result is the absolute activation-number. Note that you must follow the activation-number with a backslash (\\), and the activation-number cannot be a variable. Note also that there cannot be spaces between the routine-name and the activation-number. The default is the most recent activation.

DESCRIPTION

Use a routine-name to specify a particular subroutine, procedure, or function. Several DEBUG commands accept a routine-name as an argument.

In general, specifying a routine-name is easy. In most cases you merely have to specify the name of the routine (i.e., subroutine, procedure, or function) followed by a backslash. In a few cases (particularly those involving Pascal programs), you may also have to specify the names of other routines or modules. Let's examine routine-names language by language.

NOTE: Routine-names must always end with a backslash \\.

Routine-Names in FORTRAN

For FORTRAN programs, a routine-name usually consists of a single identifier: the name of the routine. (An unnamed main program bears the name `$MAIN`.) A two-component name such as `foo\bar\` denotes a statement function (bar) defined within a routine (foo).

Routine-Names in C

For C programs, you usually specify a routine-name of the simple format:

```
function-name\
```

However, in order to fully qualify a routine-name, you can precede the routine with the name with the module; in this case a routine-name takes the following form:

```
module-name\function-name\
```

The longer form is of particular help when a program has two functions with the same name stored in different modules.

See the *DOMAIN C Language Reference* for details on naming modules.

Routine-Names in Pascal

In Pascal, the first component of a full routine-name is the name of a program or a module. Successive components denote nested procedures or functions. For example

```
EUROPE\FRANCE\PARIS
```

denotes a routine (procedure or function) named PARIS which is internal to another routine named FRANCE, which is in turn nested in a module or program named EUROPE.

In most cases, it is not necessary to use fully qualified routine-names. If a routine (i.e., subroutine, procedure, or function) is visible from the current environment, you need only specify the name of that routine; for example, PARIS\. To specify a routine that is not visible from the current environment, you need to specify the name of the program or module containing it, followed by the name of the routine itself; for example, EUROPE\FRANCE\PARIS\. A partially qualified name, such as FRANCE\PARIS\, can also be used if the first component is visible in the current environment. (See the "ENVIRONMENT" listing for a complete discussion of environments.)

For example consider the following Pascal program structure:

```
program P
  procedure A
    procedure X
  procedure B

module M
  procedure X
    function Y
```

If current routine is	then	is equivalent to
A	X\	P\A\X\
A	B\	P\B\
A	Y\	error: Y is not visible from A
A	X\Y\	error: X refers to P\A\X\
A	M\X\Y	M\X\Y\ (full name required here)
B	X\	M\X\
B	X\Y\	M\X\Y\
B	A\X\	P\A\X\

We have not yet described precisely what is meant by a routine being "visible". For the most part, DEBUG's visibility rules are the same as those of the source language. In other words, a routine is visible if and only if it could be called from the current point of execution. However, there are a few exceptions:

- The visibility of external routines in Pascal and C programs depends on whether and where the routines are declared (typically via an insert file.) The debugger assumes that all top-level routines in modules are globally visible.

- DEBUG ignores Pascal 'INTERNAL' and C 'static' declarations in modules. Internal routines are globally visible, just as if they were exported.

- The debugger is insensitive to the order of declaration of routines within a program or module. It can make forward references not permitted in the language.

Ambiguous References

In some cases, the rules DEBUG uses to identify the routine-name can lead to ambiguous routine references. For example, suppose two different routines -- an 'INTERNAL' routine in one module and an exported routine in another -- have the same name. DEBUG cannot distinguish between the two routines unless the names are fully qualified. If an unqualified name is given, DEBUG uses whichever routine it happens to find first.

If a module has the same name as a routine, it is possible for a routine-name to be interpreted as either the fully-qualified name of one routine or a partially qualified name of another. For example, if a module named FOO contains a function named FOO, then does FOO\ refer to the module or the procedure? DEBUG normally assumes that it refers to the module. In general, if a name can be interpreted as fully-qualified, then it will be. An exception to this rule occurs when the routine-name is part of a statement-ID (see the "Statement-ID" listing). Since modules never have statements, DEBUG assumes you are referring to a routine rather than a module.

Pascal and C do not require module names to be unique. It follows that it is possible to have two routines with the same fully-qualified name. There is no way for DEBUG to distinguish between such routines. The only cure is to rename one of them.

If you compiled a file with the the -DB compiler option, the nesting structure of the routines is not known to DEBUG. You can only use simple unqualified routine-names in this case. If two routines have the same name, there is no way to distinguish between them.

In most cases DEBUG will do what you expect. When in doubt, use fully-qualified names. Also note that in some commands the current routine is the default and you need not specify it at all.

Activation-Number

In most cases a routine-name is sufficient to specify the environment for accessing variables, etc. However, for recursive routines, more than one activation of the routine, each with its own set of arguments and local variables, may be on the call stack. You use an **activation-number** to specify a particular activation of a recursive routine.

> NOTE: You cannot specify an activation-number when setting a breakpoint. However, you can simulate an activation-number by using the -AFTER option of the **BREAKPOINT** command.

By using an activation-number we can study the effects of recursion on a function's arguments. For example, assume a recursive function named FACTORIAL which takes one argument (n). Suppose we set a breakpoint at the beginning of FACTORIAL and let the function call itself three times. Here are the results:

```
> TB
Stopped at RECURSIVE_EXAMPLE\FACTORIAL\11
Called from RECURSIVE_EXAMPLE\FACTORIAL\-1\11
Called from RECURSIVE_EXAMPLE\FACTORIAL\-2\11
Called from RECURSIVE_EXAMPLE\18

> ARGS factorial\                       (Current value of N.)
RECURSIVE_EXAMPLE\FACTORIAL\N = 5
> ARGS factorial\0\                     (Current value of N.)
RECURSIVE_EXAMPLE\FACTORIAL\0\N = 5
> ARGS factorial\1\                     (First value of N.)
RECURSIVE_EXAMPLE\FACTORIAL\1\N = 7
> ARGS factorial\2\                     (Second value of N.)
RECURSIVE_EXAMPLE\FACTORIAL\2\N = 6
> ARGS factorial\3\                     (Third value of N.)
RECURSIVE_EXAMPLE\FACTORIAL\3\N = 5
```

Negative activation–numbers can be quite useful. They trace values directly backwards from the current value.

```
> ARGS factorial\0\                    (Current value of N.)
RECURSIVE_EXAMPLE\FACTORIAL\0\N = 5
> ARGS factorial\-1\                   (Value of N before that.)
RECURSIVE_EXAMPLE\FACTORIAL\-1\N = 6
> ARGS factorial\-2\                   (Value of N before that.)
RECURSIVE_EXAMPLE\FACTORIAL\-2\N = 7
> ARGS factorial\-3\                   (That's all.)
*** Error: Specified routine activation does not exist.
ARGS factorial\-3\
-----^
```

SDIR -- This command is obsolete. Use the SOURCE command instead.

SET -- Sets the values of one or more variables.

FORMAT

The **SET** command supports two different formats:

Set *-F1* *-F2* *-F3* *-F4* variable = := expression	**Set** variable1,... *variableN*
(Assignment Format)	**(Interactive Format)**

REQUIRED ARGUMENTS (Assignment Format)

variable	Any program variable or debugger variable. The variable cannot be a record, structure, or union, although it can be a single element of a record. See the "Arrays" listing for details on manipulating array variables.
:= or =	The two assignment operators are equivalent, and you can use either operator no matter what language the program was written in.
expression	Any legal DEBUG expression. See the "Expressions" listing earlier in this encyclopedia for details on DEBUG expressions.

REQUIRED ARGUMENT (Interactive Format)

variable	One or more variables. You can specify any number of variables of any type, as long as they fit on one line.

OPTIONAL ARGUMENTS (Assignment Format)

-F1 -F2 -F4 -FN	Each of these options forces the expression's value into the variable. The -F1, -F2, and -F4 options force exactly one, two, or four bytes of data, respectively. The -FN option forces into the variable as many bytes of data as there are in the expression value. In any case, DEBUG does not perform any conversion on the data, and DEBUG does not check whether or not the variable can accommodate that amount of data. Therefore, be careful: you may inadvertently change the values of other variables if the value overflows the assigned variable.

OPTIONAL ARGUMENTS (Interactive Format)

None.	(The -F1, -F2, -F4, or -FN options are illegal in the interactive format.)

DESCRIPTION

Use the **SET** command to change the value of one or more variables during the DEBUG session. The changes you make will have no effect on any future debugging sessions.

The **SET** command takes two formats:

The assignment format	Allows you to change the value of exactly one variable.
The interactive format	Allows you to change the value(s) of one or more variables in response to prompts from DEBUG.

We describe these formats separately.

The Assignment Format

Use the assignment format to directly assign the value of an expression to one variable. For example, to change the value of integer variable xeno to 5, enter the following command:

```
> SET xeno = 5
```

You can also assign the value of one variable to another. For example, assuming that integer variable yeti has a value of 6, the following command sets xeno to 6:

```
> SET xeno = yeti
```

Although you can set one simple variable to another simple variable, you cannot assign one aggregate variable (array, record, structure, or union) to another, except for character strings. For example, given two variables myrec and yourec of the same record type, the following assignment is illegal:

```
> SET myrec = yourec
```

You can, however, assign one field of an aggregate variable to another; for example:

```
> SET myrec.status = yourec.status
```

You can use the **SET** command to set every element in an array to the same value. To do so, specify the name of an array as the variable and the initializing value as the expression. For example:

```
> EXAMINE ar
ar[1] = 111
ar[2] = 222
ar[3] = 333
> SET ar = 0          (Set every element in the array to zero.)
> EXAMINE ar
ar[1] = 0
ar[2] = 0
ar[3] = 0
```

As mentioned earlier, you can assign a character string to a character array variable. If the string is shorter than the target array, DEBUG pads the string with blanks. If the string is longer than the target array, DEBUG signals an error and makes no assignment. For example, consider an array of 20 characters and the following legal and illegal assignments:

```
> PRINT s
The Cat in the Hat

> SET s = 'Comes Back'
> PRINT s
Comes Back               (DEBUG pads the remainder of the array with blanks.)

> SET s = 'is a really wonderful book for children'
***Error: String too long for the dimension of the character array.
```

NOTE: C users should note that DEBUG pads arrays with blanks, and therefore does not use a \0 to mark the end of the string. Since most C output functions depend on a delimiting \0, you will probably have to use the **SET** command to insert one yourself.

You cannot use a range subscript to assign a string to part of a character array, nor can you assign a string to a C pointer that implicitly denotes a string.

The Interactive Format

To invoke the interactive format, enter the word **SET** followed by the variables whose values you might want to change. DEBUG responds by displaying the present value of each variable one by one, and waiting for you to type in its new value. For instance, consider the following command sequence:

```
> SET x, y, z
TEST_C\x = 3 = 5
TEST_C\y = 100 =
TEST_C\z = 4.200000 = 3.14
> EXAMINE x, y, z
TEST_C\x = 5
TEST_C\y = 100
TEST_C\z = 3.140000
```

In the previous **SET** command, we changed the value of x from 3 to 5 and the value of z from 4.2 to 3.14. Since we did not enter a new value for y, DEBUG kept its old value of 100.

If you specify a structured variable (i.e., an array, record, structure, set, or union), DEBUG prints the current value of each element or field of that structured variable. Thus, you can use the interactive format to change all or part of a structured variable. For example, to change all three elements of a one-dimensional array of integers, we could enter the following:

```
> SET einstein
TEST_C\einstein[0] = 400 = 5
TEST_C\einstein[1] = 300 = 19
TEST_C\einstein[2] = 200 = 99
```

DEBUG supports many features for specifying a portion of an array. For details, see the "Arrays" listing earlier in this encyclopedia. Note that you can specify array subscripts that refer to portions of memory outside the array. If you do that, DEBUG issues a warning that you have used out-of-bounds subscripts, but you can still set the contents of the locations referred to as though they were actually part of the array. (DEBUG does not, however, issue a warning in the case of single-element one-dimensional arrays, because DEBUG assumes that you create such arrays only for the purpose of referencing memory outside the array boundaries.)

NOTE: Be careful with Pascal variant fields and C unions because the *next field* may actually occupy the same memory space as the previous field.

Special Symbols in the Interactive Format

When prompted for data, you can either enter a new value for the variable or you can do one of the following:

<blank>	If you enter a blank line, DEBUG leaves the old value of the variable unchanged.
,	If you enter a comma, DEBUG skips to the next *variable* in the command list. This is useful for prematurely ending operations on an array or record variable.
;	If you close the line with a semicolon, DEBUG completely terminates the **SET** command, even if you have given more variables on the command line list. This is useful for ending the **SET** operation without seeing the rest of the variables.

EXAMPLE OF INTERACTIVE FORMAT

```
> SET i, j, CAMBRIDGE, table, k
i = 1234 = 5678<RETURN>                (Contents changed.)
j = -1000 = <RETURN>                   (Contents unchanged.)
CAMBRIDGE(1,1) = 1 = <RETURN>          (Contents unchanged.)
CAMBRIDGE(1,2) = 2 = 0, <RETURN>       (Contents changed, and we end
                                        operations on the array CAMBRIDGE.)
table.nextp = 0012F4A4s = <RETURN>     (Contents unchanged.)
table.value = 1 = 0<RETURN>            (Contents changed.)
table.hash = 457234 = 0;<RETURN>       (Contents changed, and we end the
                                        entire SET command. Notice that DEBUG
                                        does not display the last field of "Table" or
                                        the variable "k".)
```

Variable Assignment To a Different Data Type

If you assign the result of an expression to a program variable, the data type of the result generally must be valid for the target variable. DEBUG does cross assignments in the cases shown in Table 3-3. Note that this table applies to both the interactive and the assignment format.

Table 3-3. Variable Cross Assignments

A Variable of This Type	Can Be Set Equal To a Value of These Types
Character	Any Character value. Any integer between −128 and +255.
8-bit integer (This is the C char type)	Any character value. Any integer between −128 and +255.
16-bit integer	Any 8-bit integer value. Any 16-bit integer value. Any 32-bit integer value between −32768 and 65535 inclusive.
32-bit integer	Any integer value. Any pointer.
Real	Any integer value. Any real value.
Pointer	Any pointer (need not be pointing to same base type). Any integer value.
Logical or boolean	Any logical or boolean value.
Enumerated	An enumerated value of the same type. Any integer value (DEBUG does not check the range).

Note To FORTRAN Users

If you enter a routine that has alternate entry points, DEBUG does not know which entry point you used. Therefore, DEBUG cannot determine which set of arguments is valid. The **TB** and **ARGS** commands assume the primary entry point. If you try to access a different argument, your results will be invalid and addressing violations may occur.

SHELL -- Invokes a DOMAIN Shell or executes a Shell command.

FORMAT

SHell *a-shell-command; ... a-shell-command*

REQUIRED ARGUMENTS

None.	If you issue **SHELL** without an argument, the Shell prompt ($) appears in the DEBUG window pane.

OPTIONAL ARGUMENTS

a_shell_command	A Shell command described in the *DOMAIN System Command Reference*. If you specify more than one Shell command, you must separate the different commands with semicolons (;) and enclose the entire sequence of Shell commands in single or double-quotes.

DESCRIPTION

Use **SHELL** to access Shell commands. You can issue the **SHELL** command with or without an argument.

If you issue **SHELL** without an argument, the Shell prompt ($) appears in the DEBUG commands window pane. You respond to the Shell prompt by entering Shell commands, running programs, or doing anything you would normally do in a Shell. To return to your DEBUG session, simply type an End-of-File character (usually CTRL/Z).

If you issue the **SHELL** command with an argument, the argument must be one or more Shell commands. After the operating system executes these Shell commands, control returns to DEBUG and you can continue your debugging session.

SHELL invokes the Shell in the debugger process, *not* in the target process. Thus process-sensitive Shell commands, such as LAS or LOPSTR, apply by default to the debugger process. Note that most of the process-sensitive commands permit you to specify the name of the process, so you can specify the process being debugged if you choose. For example, assuming that you want to check on process_27, you could enter the following command:

```
> SHELL las -p process_27
```

EXAMPLES

```
> SHELL
$ wd
//fountains/aqua
$ *** EOF ***

> SHELL wd
//fountains/aqua

> SHELL "wd; ld a?*"
//fountains/aqua

appendix   appendixb  assembly

3 entries listed.

>
```

SIGNAL -- Simulates a fault.

FORMAT

SIGnal *status*

REQUIRED ARGUMENTS

None.	If you do not specify an argument, DEBUG supplies the default status code. The default status code is a quit fault.

OPTIONAL ARGUMENTS

status	The numeric status code for the fault to be simulated. By default, DEBUG assumes that you are entering a decimal (base 10) integer. You can enter a number in a different base if you desire. (See the "Expressions" listing for details.)

DESCRIPTION

Use this command to simulate a particular fault at the current point of execution. DEBUG responds with its normal fault message and invokes the `FAULT_ACTION macro (if it is defined). If you continue execution with the GO -Cleanup command, DEBUG invokes program fault handling for the specified fault as described in the "Faults" listing.

DEBUG signals an error if you issue this command while the target program is stopped for another fault.

EXAMPLE

```
> SIGNAL 16#00120026
```

(Simulates a floating-point overflow fault. Notice how we specified a hexadecimal status code by prefixing the number with 16#.)

Source Code Display -- Here, we detail the source code that DEBUG displays. (See also the "SOURCE" listing which appears next in this encyclopedia.)

In Chapter 2 we described the DEBUG invocation options affecting the source code display. Here, we try to explain more fully what the source code display is by detailing the following topics:

- Source file display characteristics
- How DEBUG finds the right source file to display
- How DEBUG finds DSEE files
- Source file display with nonstandard fonts
- Programs compiled with the -DB option
- The **`src_adjust** and **`src_try_bak** DEBUG variables

Source File Display Characteristics

The source display window pane contains source code and its associated line numbers. An arrow in the source display indicates your current position in the program.

In general, the source file display is similar to a standard read window (as if you had read the source file with the Display Manager's READ key). Most Display Manager commands are available to you if you position the cursor in the source file window pane. You can scroll, search, and move around in the pad. The line numbers and arrow are in a separate window pane from the source text. DEBUG normally keeps these synchronized, but if you scroll the text, DEBUG does not automatically scroll the line numbers and arrows. You can use the Display Manager "=" command to determine the correct line number of any line in the display.

You cannot edit the contents of the source display window pane. However, you can bring up the source code file as a separate window and edit it during your DEBUG session. The changes will not appear in the source display until DEBUG refreshes the source display (e.g., if you specify a **SOURCE** command). When you make a change to your source code, the change is not reflected in the object code. Therefore, changing the source code while debugging can be somewhat misleading.

You can use the **ENVIRONMENT** command or the **SOURCE** command to display any source code you want to see (as long as the file is available). See the "ENVIRONMENT" and "SOURCE" listings in this chapter for details.

How DEBUG Finds the Right Source File to Display

In most cases, DEBUG has no problem finding the appropriate source file to display. However, between the time you compile and debug the program, it is possible that you have modified the source file, moved it, or renamed it. If you've done any of these things, DEBUG may require some assistance in finding the proper file to display. DEBUG searches for the file in the following order:

1. If you specify a file in a **SOURCE** command, DEBUG displays that file.
2. If the pathname and time-stamp recorded in the object module match those of an existing file, then DEBUG displays that file.
3. If you have set the debugger variable **`src_try_bak** to a non-zero value, DEBUG appends the .BAK suffix to the compilation pathname. Then, if this file exists and its time-stamp matches the compilation file, DEBUG displays it.

4. If you have specified alternate search directories with the –SDIR option or **SOURCE** command, then DEBUG searches these directories for a file with the same name and the proper time–stamp. DEBUG searches the directories in the same order you defined them. If **`src_try_bak** is non–zero, DEBUG also checks the .BAK versions of the file.

5. DEBUG next searches for the file in the working directory.

6. If DEBUG cannot find a file with the proper time–stamp but can find one or more with the right name, then DEBUG displays the first such file it finds. DEBUG issues a warning message when it displays such a file.

7. If all previous steps have failed, DEBUG displays the error message "SOURCE UNAVAILABLE." In this case, track down the file yourself and issue a **SOURCE** command to tell DEBUG of the file's location.

How DEBUG Finds DSEE Files

The preceding search procedure is extended for files managed by the DOMAIN Software Engineering Environment (DSEE) as follows:

8. If a file encountered in the search is a DSEE element and the default version has the wrong time–stamp, then DEBUG looks for an alternate version with the proper time–stamp.

When you compile a reserved file under DSEE, the source file name recorded in the object module is the DSEE library element name, not the actual name of the reserved copy. DEBUG therefore initially looks for the file in the DSEE library rather than the working directory. To enable DEBUG to find the proper file you must do one of the following:

- Run DEBUG in a process created by DSEE's CREATE ENVIRONMENT command, so that all source file references are automatically redirected to the proper versions. (We recommend this method.)

- Run DEBUG with the same working directory as that used by DSEE. DEBUG finds the file at step 5 of the procedure above.

- Use the SOURCE command (or the –SDIR option) to direct DEBUG to search DSEE's working directory.

Source File Display With Nonstandard Fonts

If you use a standard font, DEBUG has no trouble drawing an arrow to indicate the current line number. However, if you invoke DEBUG in a Shell that is running a nonstandard font (i.e., your own font), you must define the appropriate characters. To print an arrow, DEBUG prints hexadecimal characters 1E and 1F. Therefore, you must use the EDFONT program (described in the *DOMAIN System Command Reference*) to create definitions for these two ASCII values.

You can create any characters you want for 1E and 1F. If you want to add the arrow pointer, define character 1E as a horizontal bar representing an arrow shaft ("–", for example). Then, define character 1F as a right–pointing arrowhead.

Programs Compiled With the –DB Option

If you compile a program with the –DB option, DEBUG reports

```
SOURCE FILE UNKNOWN
```

However, you can still use the source display feature by supplying the name of the file in a **SOURCE** command. See the "SOURCE" listing for details.

Note the difference between the preceding message and the following, which indicates that DEBUG knows the name of the source file, but is unable to find it.

```
SOURCE FILE UNAVAILABLE
```

The `src_adjust DEBUG Variable

DEBUG's default strategy for displaying the source location is:

- If the line is already visible in the display, DEBUG moves the arrow to the line without scrolling the text.
- If the line is not visible in the display, DEBUG scrolls the text to put the desired line at the top of the window pane.

The **`src_adjust** DEBUG variable enables you to display a specified minimum number of lines above and below the target line. For example, if you enter:

```
Break at: $MAIN\8
> SET `src_adjust = 2
> BREAKPOINT 15
> GO
Break at: $MAIN\15
```

the source window display begins with line 13 rather than with the target line 15. This variable is helpful when you want to view entire sections of code in context.

If you set **`src_adjust** to a negative number, the variable is deactivated, and DEBUG displays source code lines in the default manner.

The `src_try_bak DEBUG Variable

If you invoke DEBUG for a program that you modified but didn't recompile, DEBUG issues the warning message:

```
***Warning:   Source file and object file have different "modified" dates
20-Mar-1985 14:15:00  //WALDEN/POND/HENRY.FTN
13-Mar-1985 14:49:31 in the object file.
```

The preceding message indicates that DEBUG discovered that the current version of the source file contains a newer modified date than the one recorded in the object. Therefore, the object code you are about to debug does not contain the changes you made to the program. However, if the **`src_try_bak** debugger variable is activated, DEBUG prints the warning message and then attempts to display the .bak (backup) file. To activate the variable, just set it to any number greater than zero; for example:

```
> SET `src_try_bak = 1
```

If you want **`src_try_bak** to apply to the first source code file displayed, you have to use the -set command line option when you invoke DEBUG; for example:

```
$ debug -set `src_try_bak=1 -src henry.bin
```

SOURCE -- Specifies the source code file that DEBUG displays.

FORMAT

		Zero or one from this column.
SOurce	*compiled-file*	*pathname*
		-NONE

You can abbreviate the **SOURCE** command to **SO** or **SRC**.

REQUIRED ARGUMENTS

None.	If you specify **SOURCE** with no arguments, DEBUG displays the name of the current source file.

OPTIONAL ARGUMENTS

compiled-file	The name of a target program source file at the time it was compiled. You may enter either a full pathname or a partial pathname that is unique within your program name. (A leafname is usually all that DEBUG requires.)
pathname	The pathname of a file or a directory.
-NONE	A keyword that tells the debugger that no source file is available for the given or implied compiled-file.

DESCRIPTION

In the "Source Code Display" listing we described how DEBUG displays the source code that you are debugging. Most of the time, DEBUG chooses the correct source code to display. However, in some cases, DEBUG chooses incorrectly. For these cases, you can use the **SOURCE** command to specify the correct source code for DEBUG to display. In fact, you can use the **SOURCE** command to display any file you desire (even source code from another program).

SOURCE can take zero, one, or two arguments. With no arguments, the command shows the name of the current source file. If the file used for source display is different from the originally compiled file, both file names are shown. (Note that specifying **SOURCE** with no arguments produces results even if you specified -NSRC when you invoked DEBUG.)

With one or two arguments, the interpretation of the command depends on whether **compiled-file** is present and on whether **pathname** resolves to a file or to a directory. There are five cases.

Case 1: Compiled-file is Present, Pathname is a File

If you specify both a pathname and a compiled-file, and if pathname resolves to a filename, then pathname specifies the file to be used for displaying source for the given compiled-file. For example, the following command tells DEBUG to use file `old_util.pas` (in the working directory) for source display of the file whose leaf name was `util.pas` when it was compiled:

```
> SOURCE  util.pas  old_util.pas
```

Case 2: Compiled-file is not Present, Pathname is a File

If you specify a pathname that resolves to a file, and you do not specify a compiled-file, then DEBUG displays the file stored at the pathname. For example, the following command causes DEBUG to display the contents of file `old_util.pas` stored in the working directory.

```
> SOURCE  old_util.pas
```

This command works even if you compiled `old_util.pas` with -DB instead of -DBA or -DBS. However, it does not work if you compiled with -NDB.

DEBUG records the association between the current routine and this source code; therefore, if you return to this routine, DEBUG redisplays this source code.

If the current environment shifts from a routine with no symbol table (i.e., compiled with -DB) to another routine with no symbol table, DEBUG assumes that the two routines are in the same source file. If this is not the case, enter another **SOURCE** command to specify the correct file.

Case 3: Compiled-file is Present, Pathname is a Directory

If you specify both a pathname and a compiled-file, and if the pathname resolves to a directory, then the command is equivalent to:

```
> SOURCE compiled-file pathname/compiled-file
```

For example, the following two commands are equivalent:

```
> SOURCE util.pas archives
> SOURCE util.pas archives/util.pas
```

Case 4: Compiled-file is not Present, Pathname is a Directory

If you only specify one argument -- a directory -- then the pathname is added to the list of alternate directories to be searched for all source files. For example, suppose that directory `archives` is directly underneath the working directory. By issuing the following command:

```
> SOURCE archives
```

DEBUG searches `.archives` if it cannot find the source code in the working directory.

Case 5: The -NONE Option

The -NONE option used in place of a pathname specifies that no source file is available for the given or implied compiled-file. This is useful when the debugger displays the wrong source file and the right one is not available. DEBUG leaves the source display window pane blank.

Statement-ID -- This listing tells you what to provide when a command takes a statement-ID as an argument.

FORMAT

A statement–ID is not a command, but is an optional argument to many commands. It takes the following format:

routine-name\line-ID+offset

REQUIRED

You must specify a routine–name, a line–ID, or both.

OPTIONAL

routine–name	The name of the routine containing the statement. (See the "Routine–Names" listing for a description.)
line–ID	The line number of the statement in the program source file or the name of a label used in your source code. (We mean a "label" in the traditional programming sense; that is, something that a goto statement jumps to.) If the label is numeric, you must precede it by a pound sign # to distinguish it from a line number.
offset	An integer describing the position of the statement on the line.

DESCRIPTION

Several DEBUG commands require a statement–ID. A statement–ID identifies a particular statement in the source code. A statement–ID can take any of the following forms:

1. A line–ID by itself; for example, `46`
2. A line–ID followed by an offset; for example, `46+2`
3. A routine–name by itself; for example, `jersey\newark\kozinski\`
4. A routine–name followed by a line–ID; for example, `jersey\newark\kozinski\46`
5. A routine–name followed by a line–ID followed by an offset; for example, `jersey\newark\kozinski\46+2`

You can omit a routine–name (that is, you can use Forms 1 or 2) if the statement–ID is either:

- a label in the current environment, that is, the current routine or any statically enclosing one.
- a line–number in the current source file, that is, the source file containing the current routine.

If you omit a line–ID, you must specify a routine–name by itself (Form 3). In this case, DEBUG interprets the statement–ID as the first executable statement in the routine.

Offset

You can use an offset (Form 2 or 5) whenever you want; however, the only practical use of an offset is to distinguish between two or more statements occupying the same line. In such a case, the first statement has an offset of zero (the default); the second has an offset of one, and so on. For instance, suppose line 50 contains two statements as shown below:

```
(00050)  a := b;  d := a + 50;
```

To distinguish between them, you can refer to the first statement as statement-ID 50 and the second statement as statement-ID 50+1. Notice that you cannot leave any blank spaces between the line-ID, the plus sign, and the offset. The plus sign is mandatory (minus signs won't work).

An unusual but legal use of offsets is to refer to statements on subsequent lines. For example, the following might be two statements in a compiler listing:

```
(00100) IF (A = B) THEN c := d;
(00101) i := i + 1;
```

The statement-ID 100 refers to:

```
IF (A = B) THEN
```

The statement-ID 100+1 refers to:

```
c := d;
```

The statement-IDs 100+2 and 101 both refer to:

```
i := i + 1;
```

EXAMPLES

The **BREAKPOINT** command takes an optional statement-ID as an argument. Here are several different ways to use statement-IDs:

`> BREAKPOINT 50`	*(Line 50 of the current file.)*
`> BREAKPOINT #50`	*(The statement preceded by the program label named 50 in the current file.)*
`> BREAKPOINT math\12`	*(Line 12 in the file containing routine math.)*
`> BREAKPOINT math\`	*(The first executable statement in routine math. Don't forget that a routine-name must end with a backslash.)*
`> BREAKPOINT loop`	*(The line that begins with the program label named loop.)*

STCODE -- Displays the text of a system error code.

FORMAT

STCode expression

REQUIRED ARGUMENT

expression	Any integer expression that evaluates to a legal status code.

OPTIONAL ARGUMENTS

None.

DESCRIPTION

You supply a numeric status code to STCODE; in return, it provides the error message associated with the number.

An important difference exists between the DOMAIN Shell command STCODE and the DEBUG command **STCODE**. The Shell command expects a hexadecimal (base 16) value, but the DEBUG command expects a decimal (base 10) value. If you want to provide a hexadecimal value to the DEBUG **STCODE** command, you must precede it with 16# to indicate that the value is base 16. (See the "Expressions" listing for details on different bases.)

EXAMPLE

The following example shows a single status code in decimal first and then in hexadecimal:

```
> PRINT status.all
84279297
> STCODE 84279297
(stcode) end of file (from library / read)

> PRINT -HEX status.all
05060001
> STCODE 16#05060001
(stcode) end of file (from library / read)

> STCODE status.all
(stcode) end of file (from library / read)
```

STEP -- Executes the next statement in the target program.

FORMAT

```
STep  -Over
      -Until  statement-ID
```

REQUIRED ARGUMENTS

None.	If no arguments are given, the command advances the program to the next statement and then halts the program.

OPTIONAL ARGUMENTS

-Over	Suppresses single-stepping through a routine about to be called.
-Until	Single-steps the program from its current location up until statement-ID.

DESCRIPTION

Use the **STEP** command to advance the program counter through one or more statements. The **STEP** command can optionally be followed by one of two arguments. The two arguments are mutually exclusive. We examine these options separately.

The -Over Option

Use STEP -Over to jump straight through a routine. DEBUG implements a STEP -Over command in two parts. First, it sets a temporary breakpoint at the statement immediately following the current statement. (In other words, it sets the breakpoint on the statement the routine will return to.) Second, DEBUG internally issues the equivalent of a **GO** command, so that the program runs at full speed through the routine and then stops when the routine returns.

To study this option, consider a program having the following line numbers:

```
15      routine y
16              first statement in routine y
17              .
18              .
19              .
20      end of routine y
21
22      routine x
23              .
24              .
25              .
26              a statement that calls routine y
27              .
28              .
29              .
30      end of routine x
```

Now compare the following two transcript pads to understand how **STEP** works with and without -Over:

STEP WITHOUT -OVER

```
> STEP
Stepped to: x\26
> STEP
Stepped to: y\16
> STEP
Stepped to: y\17
> STEP
Stepped to: y\18
> STEP
Stepped to: y\19
> STEP
Stepped to: y\20
> STEP
Stepped to: x\27
```

STEP WITH -OVER

```
> STEP
Stepped to: x\26
> STEP -OVER
Stepped to: x\27
```

Without -Over the program breaks at every statement in the called routine, but with -Over the program breaks at no statements in the called routine. Nevertheless, whether or not you use -Over, DEBUG does execute every statement in the called routine. In other words, the results will be the same, you'll just get there faster with -Over.

The -Over option only has meaning when the program is stopped at a routine that contains a call to a procedure, function, or subroutine. If the statement does not contain one of those calls, a **STEP** -Over command works just the same as a **STEP** command.

The -Over option does not suppress any breakpoints in the called routine. For example, if we had set a breakpoint at line 19 of routine y, the STEP -Over command would have stopped at line 19. Interestingly, if you would then issue a **GO** command from line 19, DEBUG will halt at line 27 and report "`Stepped to: x\27.`"

NOTES: If you use **STEP** without -Over and if a called routine does not have available debugging information, then DEBUG will treat the command as a **STEP** -Over.

DEBUG always steps over calls to DOMAIN system library routines. See Appendix C for information about installed libraries.

The -Until Option

Use the -Until option to single-step the program from the current statement to the statement you specify. STEP -Until is similar to the GO -Until command; the end results are the same. However, Go -Until is several thousand times faster (no exaggeration) than STEP -Until. In fact, STEP -Until is so slow that you may be fooled into thinking that your program is hung. The obvious question, therefore, is "why use STEP -Until?" The not-so-obvious answer is that a GO -Until command cannot be used on read-only code (such as installed libraries), but a STEP -Until command can.

If STEP Doesn't Seem To Work Properly

You may not be able to use **STEP** to go past a FORTRAN READ or WRITE statement that contains an error and/or exit clause. A FORTRAN IOSTATUS parameter can also cause DEBUG to lose control. See Section 4.1.4 for details on both problems.

STEP sometimes steps across more than one statement. Depending on compiler optimization, STEP *may* consider a "tight" loop like the following to be a single statement:

```
          DO 100, i = 1, 9
100    init(i)=' '
```

Note that DEBUG will single-step the machine instructions that make up this loop. Single-stepping is several thousand times slower than full-speed execution. For a long loop, this slow-down may create the impression that DEBUG has hung.

Similarly, you may compile other combinations of statements so that individual statements are not accessible. Use of the -DBA option minimizes such problems. (See Appendix C for a complete discussion of compiler optimization and debugging.)

TB -- (Traceback) Displays the current call stack.

FORMAT

TB *-Frames n* *-Args*

REQUIRED ARGUMENTS

None.

OPTIONAL ARGUMENTS

-Frames	Displays the *n* most recent calls only.
-Args	Displays routine arguments and shows their contents.

DESCRIPTION

Use the **TB** command to follow the chain of calls leading to the current routine. For instance, if you want to know what routine called the routine you are currently in, use the **TB** command. Note that the **TB** command has no long form; that is, you cannot invoke it by spelling out TRACEBACK.

You can issue the **TB** command with no options, one option, or two options. With no options, the **TB** command displays all calls from the most recent to the least recent. If you specify the -Frames option, **TB** displays the n most recent calls only. Specifying the -Args option is something like specifying a combination **TB** command and **ARGS** command. In other words, DEBUG traces the names of the routines and the values of all arguments to those routines.

EXAMPLES

```
> TB                                    (List the sequence of routine calling.)
Stopped at TEST_C\g\4
Called from TEST_C\double_it\17
Called from TEST_C\main\27
Called from UNIX_$MAIN\190
Called from <apollo_c_startup>


> TB -Frames 2                          (List
Stopped at TEST_C\g\4                    the current routine and
Called from TEST_C\double_it\17          the routine that called it.)


> TB -Frames 3                          (List
Stopped at TEST_C\g\4                    the current routine and
Called from TEST_C\double_it\17          the routine that called it and
Called from TEST_C\main\27               the routine that called it.)


> TB -Frames 2 -Args                    (List
Stopped at TEST_C\g\4                    the current routine and
TEST_C\g\motion = 5.340000000000000      the names and values of its arguments and

Called from TEST_C\double_it\17          list the routine that called it and
TEST_C\double_it\multiplier = 2          the names and values of its arguments.)
```

VA -- (Virtual Address) Displays the virtual address of a variable, routine, or program counter.

FORMAT

```
     Zero or one of the
     following options.
VA | variable1, ..., variableN |
   | -Routine  routine-name    |
```

REQUIRED ARGUMENTS

None.	If you specify **VA** without arguments, DEBUG displays the current symbolic location and the virtual program counter (PC).

OPTIONAL ARGUMENTS

variable — To find the address of program variable or debugger variable, specify its name. You can specify one or more variable names.

-Routine — To find the starting address of a particular routine, you can specify one of the following three formats:

-Routine	Displays the address of the current routine.
-Routine routine-name	Displays the address of the named routine.
routine-name	Displays the address of the named routine.

(See the "Routine-Name" listing for details on routine-names. Don't forget that a routine-name must end with a backslash \.)

DESCRIPTION

Use the VA command to determine the address of one or more variables or of one routine. You can use VA with or without arguments.

If you specify a routine-name, the starting address shown is that of the first instruction in the routine's procedure section.

It is not usually necessary to be concerned with memory addresses when you use DEBUG. However, the need to inspect raw storage may occasionally arise. The -VA command, combined with the ability to de-reference an absolute address as a pointer (see the "Pointer Variables" listing) provides a means of doing this.

The VA command cannot be abbreviated to V, nor expanded to VIRTUAL.

> NOTE: You cannot mix routine-names and variables in the same VA command. Also, you cannot specify more than one routine-name in the same VA command.

EXAMPLES

The following examples show some typical output of the VA command.

```
> VA                             (Return the program counter.)
Stopped at: TEST_C\main\19
PC = 10030
> VA r                           (Return the address of variable r.)
TEST_C\main\r = 32-bit integer, local. VA = 9872EC

> VA r, str                      (Return the address of two variables.)
TEST_C\main\r = 32-bit integer, local. VA = 9872EC
TEST_C\str = array (6 bytes) of 8-bit integer, static. VA = 1017C
TEST_C\str[0] = 8-bit integer, static. VA = 1017C
 : : : : : :
TEST_C\str[5] = 8-bit integer, static. VA = 10181

> VA -Routine                    (Return the address of the current routine.)
TEST_C\main\   VA = 10024

> VA -Routine f\                 (Return the address of routine f.)
TEST_C\f\   VA = 10000

> VA f\                          (Return the address of routine f.)
TEST_C\f\   VA = 10000
```

Variables -- Here, we explain how to specify program variables in commands that accept them as arguments.

FORMAT

"Variables" is not a command, but many DEBUG commands accept one or more variables as arguments. A variable takes the following format in DEBUG:

routine-name\variable-name

REQUIRED

variable-name Specify a variable of any data type. When debugging a FORTRAN or Pascal program, DEBUG is case-insensitive to variable-names. But when debugging a C program, DEBUG is case-sensitive to variable-names. You should refer to the "Arrays" listing and the "Pointer Variables" listing for details on specifying variables of those data types.

OPTIONAL

routine-name Enter the name of the routine containing the variable. If you omit the **routine-name,** DEBUG searches the current routine first, followed by routines which statically enclose it, working outward. Don't forget that a routine-name can include an activation-number in order to distinguish between multiple activations of a recursive routine. See the "Routine-Name" listing for details.

DESCRIPTION

When a DEBUG command takes a program variable as an argument, you must specify a variable in the format shown at the top of the page. That is, there must at least be a variable-name, and there can optionally be a routine-name and an activation-number. The variable-name can take any of the following forms:

- A simple identifier representing a simple or an aggregate variable (though aggregates are restricted in some commands).
- A variable name denoting an array (or C pointer) followed by a set of subscript expressions. A subscript can be any expression that evaluates to an integer. (If the source code was written in Pascal, the subscript can be any expression that evaluates to a value of the array index type.) DEBUG supports several means for specifying all or a portion of an array. See the "Arrays" listing for details. Here are some sample array variables:

```
my_array[i,j]
keys(current+1)
name[*]
```

- A variable name denoting a pointer (or Pascal file variable) followed by the de-referencing operator "^". DEBUG does not support the standard C de-referencing operator "*". For example:

```
head_ptr^
```

- A variable name denoting a record, structure, or union, followed by the field selection operator "." and a field name. For example, here are two sample variables:

```
status.all
rec.code
```

DEBUG does support both the C and the Pascal de-referencing operators for pointers to structures and unions, namely "->" and "^.". Here are some valid uses of pointer variables:

```
head_ptr^
ptr_to_rec->priority
ptr_to_rec^.priority
```

Note that you can only apply subscripting, de-referencing, and selection to variable names, not to general expressions. That is, DEBUG does not permit expressions such as (p+1)^.

FORTRAN Common Names

You cannot use FORTRAN common names to reference variables. You must make a reference to a specific instance of the variable in a specific routine.

EXAMPLES

Here are a few samples of variables:

`v1`	*(A variable. v1 must be visible from the current environment. See the "Routine-name" listing for details on visibility.)*
`tree_walk\v2`	*(A routine-name and variable. v2 must be visible from from routine tree_walk.)*
`tree_walk\4\v3`	*(A routine-name, activation-number, and variable. v3 must be visible from routine tree_walk. The activation-number 4 specifies the fourth activation of variable v3.)*
`tree_walk\-1\v4`	*(A negative activation-number denotes an activation relative to the most recent one. Therefore, we are denoting the variable v4 in the next to most recent activation of the routine.)*

# -- Adds a comment line to DEBUG.

FORMAT

\# *comment*

REQUIRED ARGUMENTS

None.

OPTIONAL ARGUMENTS

comment	Any comment that you want to enter into your debugging session.

DESCRIPTION

This command ignores any information that follows it. You cannot add comments on the same line as a valid command.

Comment lines are most frequently placed in command files. Sometimes though, you might want to enter a comment from the keyboard in order to add it to a transcript.

EXAMPLES

The following examples show the valid and invalid use of the # command.

```
> # THIS IS A VALID COMMENT.
> B COREY\CAR\    # THIS IS AN INVALID COMMENT
```

Chapter 4

Language Specific Issues

With a few exceptions, DEBUG is not sensitive to the source language of the program being debugged. While this has the advantage of presenting a uniform user interface, it also means that the interface is not optimally tailored to any single language. For example, DEBUG's expression syntax borrows elements from Pascal, C, and FORTRAN, but is not identical to any of them.

This chapter discusses some language–specific issues for users of each of the three languages supported by DEBUG. We describe the following topics:

- Areas where DEBUG's behavior is sensitive to the source language.
- Places where DEBUG's behavior differs from similar source language constructs.
- DEBUG features specifically targeted to a single source language.

Most of the material in this chapter is also covered in the encyclopedia section of the manual. The following discussion highlights topics in a language–dependent setting and provides some expanded examples.

4.1 Debugging FORTRAN Programs

Here, we address the unique ways that DEBUG handles the following FORTRAN features:

- Expressions
- Statement numbers
- Arrays declared with variable dimensions
- Alternate entry points
- I/O statements
- Statement functions

4.1.1 FORTRAN Expressions

DEBUG's rules for expression evaluation differ from FORTRAN's in the following areas:

- All DEBUG operators have equal precedence; however, as in FORTRAN, you can use parentheses to guarantee a specific order of evaluation.
- DEBUG does not support the following FORTRAN operators:

```
**        (exponentiation)
.NOT.
.XOR.
.EQV.
.NEQV.
```

- DEBUG interprets two identifiers separated by a period as a record, structure, or union. Therefore, if a DEBUG expression contains a dotted logical or relation operator (e.g., .LT.), then you must use blanks to separate the operator from any surrounding operands. If you don't, then DEBUG will probably confuse the operator for a record, structure, or union. For example, consider the right and wrong ways of using the dotted .GT. operator:

```
X.GT.0     (Wrong, DEBUG thinks this is a record named X instead
            of a .GT. operation.)
X .GT. 0   (Right, DEBUG views this as a .GT. operation.)
```

4.1.2 FORTRAN Statement Numbers

You must prefix a *statement* number with "#" in order to distinguish it from a *line* number; for example:

```
> BREAKPOINT #100   (Set a breakpoint at the statement preceded by the label "100")
> BREAKPOINT 100    (Set a breakpoint at program line number 100)
```

4.1.3 FORTRAN Alternate Entry Points

DEBUG has no information on alternate entry points. Therefore, to set a breakpoint on an alternate entry point, you must set it at the appropriate line of the main subroutine. If you enter a routine that has alternate entry points, DEBUG does not know which entry point you used. Therefore, DEBUG cannot determine which set of arguments is valid. The **TB** and **ARGS** commands assume the primary entry point. If you try to access a different argument, your results will be invalid and addressing violations may occur.

4.1.4 FORTRAN I/O Statements

READ, WRITE, and INQUIRE statements with an ERR, END, or IOSTAT specifier can cause DEBUG to lose control if you attempt to **STEP** through the statement. In the case of IOSTAT this can occur even though no apparent transfer of control results when the statement is executed.

To avoid loss of control, set breakpoints at the statements specified by ERR and/or END, and at the next sequential statement if there is an IOSTAT parameter. For example, suppose you are stepping through a program and the next statement is:

```
READ (5, 100, END = 900, IOSTAT = RSTAT) X, Y, Z
```

In this case, before doing the **STEP** you should set breakpoints at statement 900 and at the statement following the READ statement.

4.1.5 FORTRAN Statement Functions

You cannot use any of the following commands when the current run environment is a statement function:

```
> BREAK -EXIT
> GO -EXIT
> GO -LOCATION
> GO -RETURN
```

4.2 Debugging C Programs

Here, we examine the unique ways that DEBUG handles the following C features:

- Case–sensitivity
- Expressions and operators
- Preprocessor symbols
- Characters and strings
- Integers
- Pointers
- Inner–block variables

4.2.1 C Case–Sensitivity

The C compiler is case–sensitive. That is, the compiler sees the following names as representing three different variables:

```
ROBERTA
roberta
Roberta
```

Because the C compiler is case–sensitive, when debugging a C program, DEBUG is also case–sensitive to names. However, if you compile a C program with the –DB option, DEBUG will be case–insensitive to names.

4.2.2 C Expressions and Operators

DEBUG's rules for expression evaluation differ from C's in the following areas:

- All DEBUG operators have equal precedence; however, as in C, you can use parentheses to guarantee a specific order of evaluation.
- DEBUG's arithmetic operators treat all long integer operands as signed long integers.
- DEBUG uses a different format than C for establishing octal and hexadecimal constants. (See the "Expressions" listing of Chapter 3.)
- DEBUG does not scale by the size of objects when doing pointer arithmetic. Hence if P is a pointer to an element of an array of long integers, the address of the next element is P+4, not P+1.
- In DEBUG, you can only apply pointer de–referencing and subscripting to variables, not to arbitrary expressions. Expressions such as (p+4)^ are not valid.
- DEBUG *does not* support the C operators shown in Table 4–1.

Table 4-1. C Operators Not Supported By DEBUG

Operators	Meaning
++	increment
--	decrement
~	bitwise not
!	logical not
unary &	address (see VA command)
*	pointer dereference (use Pascal ^ operator instead)
^	bitwise xor
&&	logical and (DEBUG uses it as a bitwise and)
||	logical or (DEBUG uses it as a bitwise or)
=, +=, -=, ...	assignment operators (use DEBUG's SET command instead)
? :	conditional operator
sizeof	sizeof operator
type casting	type casting operators

4.2.3 C Preprocessor Symbols

If the source file contains "#line" directives, they are reflected in the debugging information for the affected functions. DEBUG displays the source file specified by the #line directives and uses the specified logical line numbers rather than the physical line numbers of the file actually compiled. In order for this to work correctly, directives must not specify more than one source file and the logical line numbers should be in increasing order.

Except for #line, DEBUG does not recognize symbols defined by preprocessor control lines. For example, if you specify

```
#define end_flag 0
```

the identifier `end_flag` is *not* passed to the debugger.

4.2.4 C Characters and Character Strings

To specify either a character constant or a string, surround the character(s) in single or double-quotes; for example:

```
'A'
"big"
'thrill to be'
"here, Johnny."
```

NOTE: The escape sequence '\ddd' is not available.

When you **EXAMINE** or **PRINT** a character variable, DEBUG displays its value as a character by default. If you want to view the variable's integer equivalent, use the -Decimal option.

Assigning a String Constant to an Array Variable

DEBUG does not automatically append an end-of-string byte (\0) when you specify a string constant. Therefore, when your program depends on a string terminating with a \0, you must append the \0 yourself (with a **SET** command). For example, consider an array of char named `animal`. And suppose you want to change its value from "tiger" to "cat". Observe the following process:

```
> EXAMINE animal -decimal
TEST_C\animal[0] = 84               (t)
TEST_C\animal[1] = 105              (i)
TEST_C\animal[2] = 103              (g)
TEST_C\animal[3] = 101              (e)
TEST_C\animal[4] = 114              (r)
TEST_C\animal[5] = 0                (\0)

> SET animal = "cat"                (Change its name to "cat")
> EXAMINE animal -decimal
TEST_C\animal[0] = 99               (c)
TEST_C\animal[1] = 97               (a)
TEST_C\animal[2] = 116              (t)
TEST_C\animal[3] = 32               (blank)
TEST_C\animal[4] = 32               (blank)
TEST_C\animal[5] = 32               (blank)

> SET animal[3] = 0                 (Terminate string with a \0.)
```

It is generally poor debugging style to store a string with more characters than were allocated for the array variable because you run the risk of overwriting some other variable. Nevertheless, by using the -fn option of the **SET** command, DEBUG does permit you to store a large string into the space allocated for a small array; for example:

```
> SET -fn animal = "Elephant"
```

Your success in doing this will depend on how the compiler has allocated variable space.

The Special int8 and byte Types

If you use "char" as a type specifier for de-referencing an absolute address or debugger variable, "char" refers to the Pascal char type, which is not an integer. To refer to 8-bit integers, use the special DEBUG type specifiers "int8" or "byte" instead. For example:

```
> PRINT `p^char+1
*** Error: Incompatible operands in the expression.
> PRINT `p^int8+1
89
> PRINT `p^char
X
> PRINT `p^int8              (Doesn't matter in this case... )
X                            (since DEBUG displays C chars in ASCII by default.)
```

4.2.5 C Pointers

You must de-reference C pointers with trailing ^ characters, rather than leading * characters. You can de-reference pointers to structures and unions with either the -> or the ^. operator. Thus:

In C Notation	**In DEBUG Notation**
`*intp`	`intp^`
`structp->field`	`structp^.field` or `structp->field`

You can perform integer arithmetic on pointer variables; however, DEBUG does not scale the integer value by the size of the data type. You must do the scaling yourself. For example, consider the following C declarations:

```
char *cp;          /* a 1 byte data type */
int  *ip;          /* a 4 byte data type */
```

You can perform pointer arithmetic as shown below:

```
> EXAMINE cp^ -A
cp^ = N

> SET cp = cp + 1              (Adding 1 to get next char.)

> EXAMINE cp^ -A
cp^ = o


> EXAMINE ip^
ip^ = 1

> SET ip = ip + 4              (Adding 4 to get next integer.)

> EXAMINE ip^
ip^ = 2
```

Pointers and Arrays

You can examine and subscript C pointer variables the same way that you examine and subscript C arrays. For example, consider the following declaration of str and strp:

```
char str[10] = {"Bon jour"}, *strp = str;
```

The following DEBUG session manipulates the array variable and the pointer variable:

```
> PRINT strp            (Returns the address stored in strp.)
0001011C
> VA str                (The VA command is also helpful for finding addresses.)
TEST_C\str = array (9 bytes) of 8-bit integer, static. VA = 1011C
> PRINT strp^           (De-reference strp.  It returns the first char in str.)
B

> PRINT str^            (You cannot refer to an array as if it were a pointer.)
*** Error: Variable is not a pointer.

> EXAMINE str[4:6]      (You can EXAMINE a string directly...)
TEST_C\str[4] = j
TEST_C\str[5] = o
TEST_C\str[6] = u
> EXAMINE strp[4:6]     (   or you can EXAMINE it through its pointer.)
TEST_C\strp[4] = j
TEST_C\strp[5] = o
TEST_C\strp[6] = u
> PRINT strp[*]         (PRINT the entire string that strp points to.)
Bon jour

> SET strp = 16#0001011D  (You can SET a pointer to point to a new address.)
```

In the last example, we used an asterisk (*) to specify the *entire* range. You can only use the asterisk if the pointer points to a character array that ends with the \0 (null) terminator.

4.2.6 C Inner Block Variables

DEBUG supports a special naming convention (for C programs only) that allows you to access a variable declared in an inner block when that variable has the same name as an outer block variable. The naming convention takes the following format:

variable_name.*number*

where *number* is the nth declaration (in textual order) of that variable name. For example, you can access a variable named x (assuming that it has been defined in three different blocks) as:

```
x.1  or x            (The .1 modifier is optional.)
x.2
x.3
```

Now consider the following program which consists of several different layers of blocks:

```
struct foo {int q; float f;};
main()
{   int i = 100;                          /* i.1 */
    static struct foo abc = {4, -3.14};   /* abc.1 */
    static char s[] = "The first string"; /* s.1 */
    {   int i = -27;                      /* i.2 */
        { int i = 1234;                   /* i.3 */
          static char s[] = "The second string"; /* s.2 */
          static struct foo abc = {5,-2.1} /* abc.2 */
this_point: printf("GO to this point.\n");
        }
    }
}
```

Consider the following DEBUG session on the previous source file. In it, we GO to this_point and perform the following operations:

```
> EXAMINE i  i.1  i.2  i.3
TEST_C\main\i = 100          ("i" still refers to the outermost declaration.)
TEST_C\main\i.1 = 100
TEST_C\main\i.2 = -27
TEST_C\main\i.3 = 1234

> EXAMINE  abc.1  abc.2
TEST_C\main\abc.1.q = 4
TEST_C\main\abc.1.f = -3.140000
TEST_C\main\abc.2.q = 5
TEST_C\main\abc.2.f = -2.100000

> PRINT s.1
The first string

> PRINT s.2
The second string
```

Note that the compiler allocates all variables of a routine, including inner block variables, at the start of the routine. These variables remain valid to DEBUG until the entire program exits. DEBUG references, unlike references within the program itself, do not depend on the current position within the routine. Thus in the example, if the program stops in an inner block, i still refers to i.1, the outermost declaration of i.

4.3 Debugging Pascal Programs

Debugging a Pascal program is rather straightforward. However, DEBUG's handling of expressions, sets, and WITH statements will seem different to Pascal programmers.

4.3.1 Pascal Expressions

DEBUG's rules for expression evaluation differ from Pascal's in the following areas:

- All DEBUG operators have equal precedence; however, as in Pascal, you can use parentheses to guarantee a specific order of evaluation.
- Boolean constants must be entered in FORTRAN syntax, i.e., .TRUE. and .FALSE.. The terms TRUE and FALSE (no dots) will not produce the desired effect.
- The division operators / and DIV are identical; both behave like DIV if both operators are integers.
- The logical NOT operator is not implemented; however, unary minus "–" applied to a Boolean value has the same effect.
- Type–checking is less rigorous. Pointers and integers can be freely mixed, as can pointers to different base types.
- Subrange constraints, in particular those that imply unsigned integers, are ignored.

4.3.2 Pascal Sets

This section explains how to use Pascal sets (don't confuse Pascal sets with DEBUG's **SET** command). See also the "Expressions" listing in Chapter 3 for an explanation of set constants.

In this section, we will use the following definitions to explore sets:

```
TYPE
    eliot = (cruelest, month, spring, growth, melting,
             peaches, lilacs, rain, stirring, thomas, stearns);
VAR
    april : SET OF eliot;
```

Describing and Displaying Set Variables

Use the **DESCRIBE** command to see the base type and number of members of a set variable. DEBUG is not capable of displaying the names of the members of an enumerated set type. For example:

```
> DESCRIBE april
APRIL = set of enumerated type (11 members), static.
```

You can use **EXAMINE** or **PRINT** to display the value of a set variable. For instance:

```
> EXAMINE april
april = [MONTH,GROWTH,PEACHES,LILACS,RAIN]
```

Resetting the Value of a Set Variable

Use a **SET** command to change or initialize the value of a set variable. You must enclose the new value in square brackets. Each element of the new value must be a member of the set.

```
> SET april = [peaches, lilacs, rain]
```

To change a value to the null set, use square brackets with nothing inside them; for example:

```
> SET april = []
```

Note that the interactive format of the **SET** command does not permit an entry to run across more than one line. (See the "SET" listing in Chapter 3.) Therefore, an attempt like the following causes an error:

```
> SET april
SETS\APRIL = [] = [cruelest, spring, growth, melting, peaches, lilacs, rain
*** Error: Illegal character in the item.
[CRUELEST, SPRING, GROWTH, MELTING, PEACHES, LILACS, RAIN
---------------------------------------------------------^
```

4.3.3 Pascal WITH Statements

DEBUG does not recognize abbreviated variable references inside WITH statements.

Appendix A

Helpful Debugger Hints

This appendix answers some commonly asked questions about DEBUG.

Q Why does my program behave differently under DEBUG than when running alone?

A It is nearly impossible for a debugger to have no effect whatsoever on its target. There are several likely possibilities:

- Programs with uninitialized variables may appear to behave consistently when run alone (though this is largely a matter of luck), but behave differently under DEBUG because the program and/or variables are loaded at different addresses. To minimize changes to the environment of a program running under DEBUG, use both the –PROC and –NC options.
- Values of uninitialized local variables allocated on the stack can also be affected by processing of breakpoint and single–step faults which occur when a program is being debugged.
- If a program generates an access violation when running stand–alone but does not generate the violation when being debugged, the program may be writing into a read–only section. DEBUG gives write access to such sections so that DEBUG can set breakpoints. You should be able to use the TB shell command (not under DEBUG) to locate the problem. Look especially for constants passed as arguments or accessed via pointers.

Q How can I stop program execution when the value of a certain variable changes?

A DEBUG does not provide data watchpoints as a built–in feature. However, breakpoint action lists can be useful here. Set breakpoints at points in the program where you wish to check the value of the variable. Specify an action list of the following form on each:

```
-DO [IF ptr <> nil [GO]]
```

This restarts execution if the desired stopping condition is not met.

Q How can I navigate through a linked data structure?

A The usual problem here is that you need a temporary pointer variable to walk through the structure, but you cannot declare a debugger variable with the correct pointer type. (Debugger variables only support pointers in the form of integers.) A useful trick is to temporarily "borrow" a program variable of the correct type. You can use a debugger variable to save and restore the program variable's value before resuming program execution. An example of this appears in the "JUMP" listing of Chapter 3.

Q How can I debug a program that borrows the entire display (i.e., a graphics program running in borrow mode)?

A By debugging remotely from a second node. Suppose, for example, that there are two nodes in your office, named node1 and node2. Use the following procedure to debug the program:

1. From node1, use the Shell command **CRP** (described in the *DOMAIN System Command Reference*) to create a remote process. For example:

```
$ crp -on //node2 -me
```

2. Using node1's keyboard, with the cursor in the remote process Shell, invoke DEBUG. Note that the debugger and target must both run on node2. Use the -nwp and -nsrc options when you invoke DEBUG; for example:

```
$ debug graphics_program -nwp -nsrc
```

   DEBUG creates a debugging window, but does not divide the window into separate windowpanes.

3. Debug your graphics program. The graphics will be displayed on node2. Obviously, this process is much more convenient if you can see node2 from node1.

In place of node1, you can use a terminal connected to an SIO port. Assume that the terminal is connected to node2. In this case, follow these steps:

1. Using the terminal's keyboard, invoke DEBUG. Use the -proc, -nwp, and -nsrc options when you invoke DEBUG; for example:

```
$ debug graphics_program -proc Process_14 -nwp -nsrc
```

2. On node2, invoke the program from the designated process (Process_14); for example:

```
$ graphics_program
```

3. Debug your graphics program. The graphics will be displayed on node2.

Q Can I use Display Manager key definitions in the debugger?

A Yes. For example, the following definition allows you to examine a variable by pointing at it in the source display and pressing M3 (mouse key 3):

```
KD M3 \[-A-Za-z0-9_$]\; ar;dr; (Find and mark start of name.)
        /[-A-Za-z0-9_$]/;xc;     (Find end of name; copy name.)
        ti;es'e ';xp;tr;en       (Make EXAMINE command.)
KE
```

Note, however, that there is no way to create key definitions that are local to DEBUG only.

Appendix B

Debugging Installed Libraries

You can use DEBUG to debug installed libraries. If you don't know what an installed library is, read the *DOMAIN Binder and Librarian Reference*. Some rather severe restrictions apply to debugging installed libraries, and this appendix details them. One general piece of advice: since there are so many restrictions in debugging installed libraries, you are probably better off avoiding the process. To avoid it, simply bind the routines into your object file and debug the object file. When the object file works properly, install the appropriate routines and rebind the object file without the installed routines.

B.1 Controlling Program Flow

Because the code in installed libraries is read–only, you cannot set an explicit or implied breakpoint in an installed library. Therefore, you cannot use any of the following commands:

- Any BREAKPOINT command
- GO –Until
- GO –Exit
- STEP –Over

Thus, the only way to control program flow through an installed library is with one of the following two commands:

- STEP
- STEP –Until

The STEP –Until command is useful for stepping through long sequences of code, although execution is several thousand times slower than setting a breakpoint and running at full speed.

If you invoke DEBUG with the –SMAP option, DEBUG announces each new library when encountered (and prints a load map if the information is available.)

B.1.1 The DEBUG Option –GLOB

If you invoke DEBUG with the –GLOB option, you can debug routines stored in `/lib/userlib.global`. If you do not use the –GLOB option, DEBUG automatically steps over calls to such routines.

DEBUG always steps over calls to system–defined global libraries. However, if the –GLOB option is set, it is preferable to use explicit STEP –OVER commands to step over calls to system routines. The reason is that if DEBUG attempts to step into a system library, DEBUG processes the entire library and enters it into the DEBUG symbol tables before recognizing it as system code. In some cases DEBUG may not be able to distinguish between system and user code and will step into the library, leaving you stuck in system code with no debugging information. (If this happens, use a **GO** command to get out.)

B.2 Variables and Breakpoints

You can use **PRINT, EXAMINE,** and **ARGS** to display the values of variables just as you would for variables not in installed libraries. As usual, if you compiled the source code with the –DB or –NDB compiler options, you will not be able to examine the variables' values.

You cannot set a breakpoint on a routine in a library that DEBUG has not seen yet. Therefore, the first entry to a library must be done by stepping into it, rather than by setting a breakpoint. In other words, step into the routine (with the **STEP** command) rather than jumping into it with the **GO** and **BREAKPOINT** commands.

Appendix C

Debugging Optimized Code

DEBUG permits you to debug optimized code. However, some of the optimizations performed by the compilers may cause unexpected or seemingly incorrect results during debugging. This section describes the symptoms of optimization–related problems and offers some advice on debugging optimized code.

In general, you face the tradeoff illustrated by Figure C–1. In other words, the more the compiler optimizes, the closer your code will be to the final production version of the program (assuming your production code is optimized), but the harder it will be to debug.

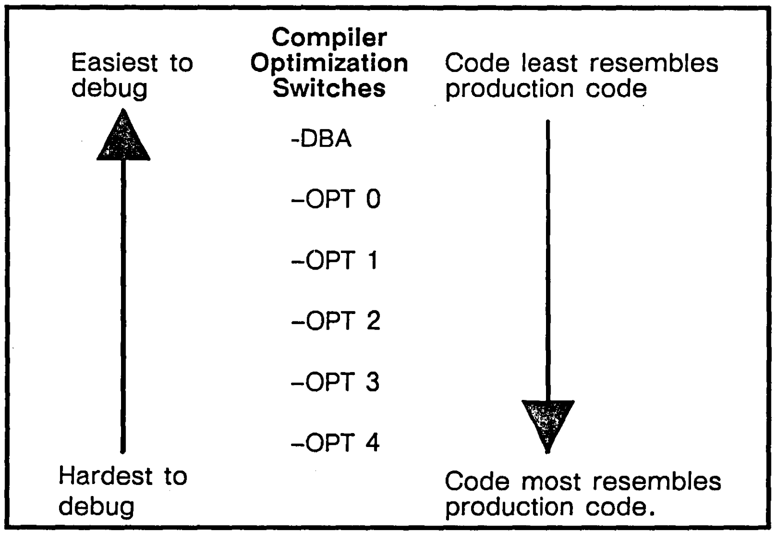

Easiest to debug
Compiler Optimization Switches
Code least resembles production code
-DBA
-OPT 0
-OPT 1
-OPT 2
-OPT 3
-OPT 4
Hardest to debug
Code most resembles production code.

Figure C–1. To Optimize or Not To Optimize?

Debugging is easiest if you switch off all optimization with the –DBA compilation option. This option ensures a direct correspondence between your source code and the object code produced. On the other hand, working with optimized code ensures that the code you debug is identical to production code compiled with –DB or –NDB options. This can increase your confidence in the correctness of the final result.

For the remainder of this appendix, we examine the debugging effects at each optimization level.

C.1 –DBA

The –DBA option tells the compiler to prevent any optimizations that could interfere with debugging. The only optimizations allowed are those that take place within a single source statement. Since DEBUG operates at a source statement level of resolution, such optimizations do not affect debugging.

C.2 -OPT 0

A Symptom -- You are stepping through a program and the source display unexpectedly jumps to another statement.

If you specify the -OPT 0 option, a compiler tries to perform **cross-jumping** (tail-merging) optimizations. That is, if the compiler detects two identical code sequences, it may replace one of them by a jump to the other. Therefore, a sequence of object code may correspond to two or more different source statements. DEBUG cannot represent this multiple correspondance, and arbitrarily relates the object code to one of the source statements. Therefore, if the program stops at such a location, the reported source location may be incorrect.

Cross-jumping optimization is done at the machine instruction level, rather than source statement level. For example, two calls to the same procedure with some, but not all, of the arguments identical can result in a cross-jump. If the program location following a cross-jump does not correspond to the start of a source statement, DEBUG reports the location as "between" two statements.

A Symptom -- The **ARGS** or **TB** command returns inaccurate results.

If a procedure has no executable statements (i.e., it is a "stub" routine), the compiler optimizes it into a single instruction. You can set a breakpoint or step into such a routine. However, if you attempt to examine its arguments or do a traceback, DEBUG returns incorrect results.

During program development it is sometimes useful to define stub routines for not-yet-written source code and then use the debugger to simulate the code's effects. You should compile such stubs with the -DBA option, so that DEBUG can access the routine's arguments.

C.3 -OPT 1

The -OPT 1 option can optimize in the following three ways:

- Dead code elimination
- Assignment merging
- Common subexpression elimination

We examine all three ways individually.

C.3.1 Dead-Code Elimination

A Symptom -- You use the GO -Location command to jump to a line of source code, but DEBUG tells you that the line does not exist. So, you double check your source code and, in fact, the line *does* exist.

Dead-code elimination means that the compiler discovered a line of source code that had no affect on the program and so decided not to generate any object code for it. For example, if you initialize a variable when you declare it, but you never use that initial value, then the compiler may decide to eliminate that initialization. Dead-code elimination has no effect on normal execution, but it does prevent you from using the GO -Location command to jump to the eliminated line of code.

C.3.2 Assignment Merging

A Symptom -- DEBUG claims that one or more record (or structure) initialization statements do not exist, but you know they do.

Multiple assignments to physically adjacent variables may be replaced by a block move that accomplishes several assignments in a single instruction. A common situation where this applies is initializing a record or struct. DEBUG may claim that some of the assignment statements do not exist (if you try to set a breakpoint on one for example), or appear to skip statements when single-stepping.

C.3.3. Common Subexpression Elimination

A Symptom -- You use the **SET** command to change the value of a variable that's part of a subexpression, but after you restart the program (with **GO** or **STEP**), subsequent mathematical operations behave as if the old value of the variable were still being used.

A subexpression used in two places may be computed once, and its value saved for the second use. If you manually change the value of one of the variables involved, the expression will not be recomputed; hence the change may appear to have no effect.

For example, consider the following two lines of Pascal code:

```
q := (x + y) * 4;
z := (x + y) - 7;
```

The subexpression `(x + y)` is common to both lines, so the compiler may decide to calculate it just once and save the result in a register for the second use. After all, it appears to the compiler that x and y do not change values between the first and second instructions. However, suppose you are using the debugger and you are stopped somewhere between the first statement and the second statement. If you use the **SET** command to change the value of x or y, then this change can have no effect on z.

C.4 -OPT 2

A Symptom -- You use the **SET** command to change the value of a variable, but after you restart the program (with **GO** or **STEP**), subsequent mathematical operations behave as if the old value of the variable was still being used. The variable was initialized to a constant value.

If the compiler can determine the value of a variable at compile time, it may replace the reference to the variable with a reference to a constant. If you use the **SET** command to change the variable's value, the change may have no effect.

For example, consider the following Pascal code:

```
CONST
   my_constant = 5;
VAR
   x, q : integer;
BEGIN
   x := my_constant;
   q := x * 4;
```

The compiler may decide to optimize x by viewing it as a constant. Therefore, the compiler will code the multiplication statement as q = 20. If you use the **SET** command to modify the value of x, the value of q will still be 20.

C.5 -OPT 3

The -OPT 3 option can optimize code in the following four ways:

- Putting local variables in registers.
- Eliminating assignments to dead static variables.
- Reordering instructions.
- Removing loop invariant statements.

We examine the four ways separately.

C.5.1 Putting Local Variables in Registers

A Symptom -- When you examine a loop index variable, its value does not seem correct.

Every variable in a program has an assigned memory location. However, local variables may be cached in machine registers for all or portions of their lifetime. In general, DEBUG knows approximately when a local variable is in memory and when it is in a register. By "approximately", we mean that the compiler knows that the variable will be in a register from somewhere in source statement A to somewhere in source statement B, and this is not quite accurate enough to ensure that DEBUG will always access the correct location.

If a variable stored in a register is "dead" (i.e., its value is not used again), then its value will not be written back to memory. If the variable was updated in the register, a stale value will thus be left in memory. For example, suppose that a loop index variable is assigned to a register during execution of the loop. Further assume that the final value of the index variable is not used after exiting the loop. In this case, the program has no reason to write the register value back to memory. Now suppose that you use DEBUG to examine the variable after the loop ends. The correct value no longer exists because the value in main memory is stale and the register value has probably been overwritten.

C.5.2 Eliminating Assignments to Dead Static Variables

A Symptom -- When you examine a static variable, its value does not seem correct.

The compiler eliminates assignments to dead variables. Therefore, the variable can be left with a stale value, similar to the case described in Section C.5.1. Assignment elimination produces a compiler warning since it usually indicates a program error.

C.5.3 Reordering Instructions

A Symptom -- The statement at line number x appears to get executed at line number x+n or line number x–n.

The compiler sometimes reorders machine instructions without regard to source statement boundaries. Therefore, the object code generated by a source statement may not necessarily directly follow the object code generated by the previous statement. Statement execution may not be complete even after the source display indicates that control has passed it. Conversely, some of the work of a statement may be done before control appears to reach it.

C.5.4 Removing Loop Invariant Statements

A Symptom -- You put a statement inside a loop, but the program appears to execute this statement before or after the loop.

A **loop invariant** statement is a statement that does not affect any other statements in the loop. Therefore, the compiler optimizes by moving it out of the loop. (After all, why calculate something 1000 times if it only has to be calculated once.)

C.6 -OPT 4

A Symptom -- When you examine a global variable used in a loop, its value does not seem correct.

Optimization level 4 extends register caching (see Section C.5.1) to global variables used in loops. However, for such global variables, DEBUG is unaware of any possible caching, so DEBUG always displays the value stored in memory. (For local variables, DEBUG can display the value stored in memory or the value stored in a register.) What this all means is that DEBUG may display the wrong value of a variable.

C.7 Summary and Advice

To close out this appendix, we offer the following advice:

- Optimization–related problems are more likely to arise if you attempt to change the values of variables than if you only examine them.

- Optimization levels 0–2 present few debugging difficulties if you refrain from changing variable values (and in practice, problems are rare even if you do.)

- Debugging at optimization levels 3 and 4 must be done with care. If DEBUG displays an unexpected value for a variable, you should consider the possibility of an optimization–related debugger problem before assuming that you have found a bug in your program.

- Optimization tends to affect procedure calls less than other kinds of code. Before making an external procedure call, the compiler must ensure that global variables and arguments passed by reference will be returned to memory. Therefore, procedure calls are often good choices for breakpoint locations.

- Programmers familiar with machine–level code will find expanded listings useful when debugging optimized code.. To get an expanded listing, simply use the –EXP option when you compile. Expanded listings show explicitly how code has been reordered, and where variables are cached in registers.

Index

The letter *f* means "and the following page"; the letters *ff* mean "and the following pages". Symbols are listed at the beginning of the index. Entries in color indicates procedural information.