Data General

Software Documentation

AOS/VS Debugger and File Editor User's Manual

AOS/VS
Debugger and File Editor
User's Manual

093-000246-01

For the latest enhancements, cautions, documentation changes, and other information on this product, please see the Release Notice (085-series) supplied with the software.

Ordering No. 093-000246

Revision 01, January 1985

NOTICE

AOS/VS
Debugger and File Editor
User's Manual
093-000246

Revision History:

Original Release - November 1980
First Revision - January 1985 AOS/VS Rev. 1.00

CONTENT UNCHANGED

The content in this revision is unchanged from (093-000246-00). This revision changes only printing and binding details.

Preface

This manual describes the use and operation of the Advanced Operating System/Virtual Storage (AOS/VS) Debugger (DEBUG) and File Editor (FED) utility. Use this product to debug ECLIPSE® MV/8000 assembly language programs and to edit AOS/VS disk files.

We have designed this manual for programmers familiar with the ECLIPSE® MV/8000 assembly language instruction set and the AOS/VS operating system. We present information at a medium pace and provide numerous examples so that relatively inexperienced programmers will be able to use this manual.

Suggested Manuals

Many concepts we mention in this manual are documented in greater depth in other Data General publications. In certain instances, you may need to refer to one of the following documents for further information:

- *AOS/VS Link and Library File Editor User's Manual* (093-000245)
- *AOS/VS Macroassembler (MASM) Reference Manual* (093-000242)
- *AOS/VS Programmer's Manual* (093-000241)
- *AOS/VS and AOS Command Line Interpreter (CLI) User's Manual* (093-000122)
- *ECLIPSE® MV/8000 Principles of Operation* (014-000648)

Reader, Please Note:

We use these conventions for command formats in this manual:

```
arg [,arg]... $X
```

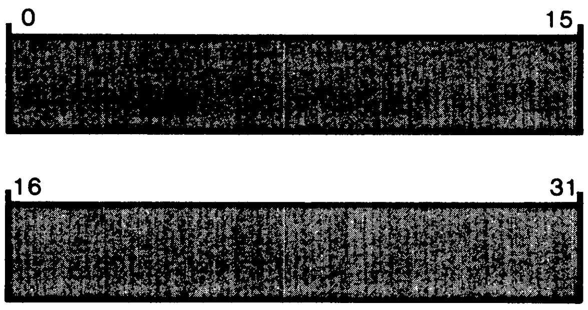
Where	Means
arg	You must enter some argument. Sometimes, we use $\left\{\begin{array}{l}\text{arg1}\\\text{arg2}\end{array}\right\}$ which means you must enter *one* of the arguments. Don't enter the braces; they only set off the choice.
[arg]	You have the option of entering this argument. Don't enter the brackets; they only set off what's optional.
...	You may repeat the preceding entry or entries. The explanation will tell you exactly what you may repeat.
$	You must enter the escape character (ASCII value 33_8). Enter this character by pressing the ESC key on your keyboard. The Debugger echoes the escape character as a dollar sign.
X	You must enter the single letter as shown. This character represents a particular command to the Debugger.

Other conventions we use throughout this manual are:

Symbol	Meaning
↵	The NEW LINE character (ASCII value 12_8)
↓	The carriage return (CR) character (ASCII value 15_8)
↑	The uparrow character (ASCII value 136_8)
$	The escape (ESC) character (ASCII value 33_8)
< $ >	The dollar sign character (ASCII value 44_8)
CTRL-X	A control character sequence. Hold down the control (CTRL) key while you strike another key (represented by X).
TAB	The horizontal tab character (ASCII value 11_8)

If your keyboard does not conform to the ANSI standard, the meanings of ⤶ and ↓ are reversed. That is, ↓ means NEW LINE and ⤶ means carriage return. Refer to Chapter 1 for more information.

We use the following format to present bit fields:



SD-02305

Bit 0 is the first bit; we call this the *most significant* bit. Bit 31 is the *least significant* bit in a 32-bit sequence.

Note that we divide bit fields into 16-bit quantities. Each 16-bit segment is called a *word.*

Lastly, we use the term *console* to refer to both video (CRT) terminals and hard-copy devices.

Contacting Data General

We welcome your comments and suggestions concerning this and other Data General publications. If you:

● Have comments on this manual -- Please use the prepaid Remarks Form that appears after the Index.

● Require additional manuals -- Please contact your local Data General sales representative.

● Experience software problems -- Please notify your local Data General systems engineer.

End of Preface

Contents

Illustrations

Tables

Chapter 1
Introduction to the AOS/VS Debugger

Overview

Frequently, the most formidable task in assembly language program development involves detecting, locating, and removing errors. We call this process *debugging* your program.

The debugging process comes after you successfully assemble and link your program (i.e., MASM and Link do not return any errors). If, at this time, you execute your program and it does not perform the desired action(s), you must correct the errors (or "bugs"). (See the *AOS/VS Macroassembler Reference Manual* for an overview of the assembly language program development process.)

Sometimes you can readily detect the errors in your program. In these cases, simply correct your source file, then reassemble and relink your program.

If you cannot easily detect the errors in your assembly language program, you can use the *AOS/VS Debugger (DEBUG) utility.* Using the Debugger, you can monitor and control your program's execution; that is, you can stop your program at any location, examine or modify the contents of memory and registers, then continue program execution at any location.

After you locate the errors in your program, leave the Debugger utility and correct your source file. Then, reassemble and relink the program. Lastly, execute the new program file to make sure you did indeed correct the problem.

Debugger Features

The following is a partial list of the AOS/VS Debugger (DEBUG) debugging tools and features. Detailed descriptions appear elsewhere in this manual.

Interactive Debugging (Chapter 2)	The AOS/VS Debugger is an interactive program; that is, DEBUG executes your commands one at a time as you enter them. This allows you to make decisions during your debugging session on the basis of DEBUG's response to earlier commands.
Memory Access (Chapter 3)	The AOS/VS Debugger provides a variety of commands that allow you to examine and modify the contents of memory. Command options permit you to view one to four words of memory at a time, search for a particular value, and step forward or backward through memory.
Register Access (Chapter 4)	You can examine and modify the contents of the following ECLIPSE® MV/8000 machine state registers: carry bit, fixed-point accumulators, stack registers, floating-point accumulators, program counter, and processor status register. In addition, the Debugger provides four special registers that access internal DEBUG variables (for example, the output radix).

Breakpoints (Chapter 5)	DEBUG allows you to place up to 20_{10} breakpoints in your program. A *breakpoint* is a location in your program where you wish to stop execution. When your program encounters a breakpoint, you can examine and modify the contents of memory and registers.
Program Symbol Recognition (Chapter 6)	DEBUG automatically recognizes the global symbols in your program (i.e., those symbols declared in .ENT pseudo-op statements). Using DEBUG commands, you can enable and disable symbol table files at will.
Temporary Symbol Definition (Chapter 6)	DEBUG allows you to define up to 20_{10} symbols during your debugging session.
Expressions (Chapter 7)	When passing numeric values to the Debugger, you may enter mathematical expressions. DEBUG provides a variety of operators (for example, arithmetic, logical, relational, and indirection operators).
Display Modes (Chapter 8)	You can direct DEBUG to present values in a variety of different display modes including numeric, half-word, symbolic, instruction, floating point, and ASCII character mode.
16-Bit Program Support (Chapter 9)	You can use DEBUG with 16-bit programs (i.e., programs developed for the 16-bit Advanced Operating System (AOS) but relinked for use on the 32-bit AOS/VS operating system). The Debugger provides full overlay support for 16-bit programs.
General Use Commands (Chapter 2)	The AOS/VS Debugger provides special commands that make DEBUG especially convenient to use. Using these commands you may generate a son CLI process and save a copy of your debugging session in a log file. In addition, the Debugger includes a help command that presents information about all DEBUG commands.

The File Editor (FED)

The AOS/VS Debugger utility is only for use with executable program files; that is, files of type PRV (program filenames usually end with the .PR extension). However, built into your AOS/VS Debugger software package is a disk file editor utility, called the *AOS/VS File Editor (FED)*. Using FED, you can examine and modify any kind of file; you are *not* limited to executable program files.

The FED commands form a subset of the DEBUG commands. When you edit a file, that file is not being executed. Thus, all DEBUG commands relating to program execution, breakpoints, and machine state registers are illegal under FED.

Refer to Chapter 10 for a complete description of the AOS/VS File Editor (FED) utility.

Differences Between Consoles

The AOS/VS Debugger supports a variety of consoles, both video (CRT) terminals and hard-copy devices. For the most part, DEBUG commands are the same for all consoles.

However, certain DEBUG commands vary depending on your keyboard layout. More specifically, some DEBUG commands vary according to the following keyboard characteristics:

- whether your keyboard has *function keys*
- whether you keyboard conforms to the *ANSI standard*

Function keys are blank keys located at the top of your terminal keyboard. Data General's DASHER™ D1 (6052), D2 (6053), D3, D4, D5, and D200 video display terminals all have function keys on their keyboards; the DASHER D100 terminal and hard-copy devices TP1 and TP2 do not. The DEBUG display mode commands vary depending on whether you keyboard has function keys or not. "Setting Display Modes" in Chapter 8 provides a complete discussion on the command differences.

The *American National Standards Institute (ANSI)* has developed a standard for the arrangement of keys on terminal keyboards. According to this standard, the NEW LINE key is larger than the other keys since you use NEW LINE so often. However, on some non-ANSI keyboards, carriage return (CR), *not* NEW LINE, is the larger key.

NEW LINE and carriage return have special meanings to the Debugger when you are examining memory. A NEW LINE character simply closes an open location while a carriage return closes the location and also opens the subsequent address. "Examining Memory Locations" and Table 3-2 in Chapter 3 provide more information on this use of NEW LINE and carriage return.

If you are using a non-ANSI keyboard, the meaning of NEW LINE and carriage return is reversed. That is, carriage return closes an open location while NEW LINE closes the location and opens the subsequent one. In sum, the big key (either NEW LINE or carriage return) closes an open location. The smaller of the two keys closes an open location and opens the next one.

In this manual, we use ↲ to represent the NEW LINE character and ↓ to represent carriage return. Reverse the meaning of these two arrows if you are using a non-ANSI keyboard.

Note that DEBUG uses your terminal's device characteristics to determine whether your keyboard has function keys and conforms to the ANSI standard. Refer to the CHARACTERISTICS command description in the *AOS and AOS/VS Command Line Interpreter (CLI) User's Manual* for more information.

End of Chapter

Chapter 2
DEBUG Operating Procedures

This chapter contains two major sections. The first explains the various ways you may enter and then leave the AOS/VS Debugger utility. It also presents general information on Debugger commands and explains how to begin program execution while in the Debugger.

The second section entitled "General Use Commands", explains how to save a copy of your debugging session, how to generate a son CLI process, and how to use the help command.

Entering the Debugger

To operate the AOS/VS Debugger utility, you must satisfy the following general requirements:

- your user profile must allow you to generate a son process
- you must leave four I/O channels open for DEBUG's exclusive use

Assuming you satisfy these requirements, you can enter a debugging session in one of two ways:

- issue a DEBUG command from the Command Line Interpreter (CLI)
- issue a ?DEBUG system call from your assembly language program

The following two sections explain these methods for invoking the AOS/VS Debugger utility.

Entering the Debugger from the CLI

In most cases, you will invoke the Debugger from the AOS/VS Command Link Interpreter (CLI). Simply enter the following command:

DEBUG program-file *[program-arg]...*

where:

DEBUG	is the CLI command that invokes the AOS/VS Debugger
program-file	is the pathname of the program file you wish to debug; you need not include the .PR extension when entering the filename
program-arg	is one or more arguments that program-file requires for execution; not all programs require arguments so *program-arg* is optional

There are no DEBUG command switches. If your program uses switches, you may include them immediately after program-file on the DEBUG command line.

After DEBUG loads your program, it displays the values of the four accumulators and the carry bit. Then the Debugger presents an underscore character, the DEBUG prompt. For example,

```
) DEBUG   PROG1↵
AOS/VS Debugger - Rev. 001.000.000.000
00000000000 00000000000 00000000000 00000000000 00000000000
_
```

The number following *Rev.* indicates which revision of the AOS/VS Debugger software you are using. The five numbers on the second line are the octal values for AC0, AC1, AC2, AC3, and the carry bit.

The third line shows an underscore character. Whenever DEBUG displays this prompt, it is ready to accept a command.

When you first receive the DEBUG prompt, your program has not yet begun execution. Thus, before you execute your program, you may set breakpoints and modify memory or registers. To begin execution of your program, use the $R command (see "Starting Program Execution" later in this chapter).

As an alternative to issuing the DEBUG command, you may use the /DEBUG switch on the CLI PROCESS command. For example,

```
) PROCESS/DEBUG/DEFAULT/BLOCK/IOC      PROG1↵
```

This command loads program PROG1 and begins your debugging session. Refer to the *AOS and AOS/VS Command Line Interpreter (CLI) User's Manual* for more information on the PROCESS command and other PROCESS switches.

When you enter a debugging session, DEBUG automatically searches for your program's symbol table file (i.e., a file with the same name as your program but with an .ST extension). If available, DEBUG uses the .ST file for symbol recognition. See "Program Symbol Recognition" in Chapter 6 for more information.

Entering the Debugger from a Program

Rather than entering a debugging session from the CLI, you may invoke the Debugger directly from the program you wish to debug. Simply issue the ?DEBUG system call from your assembly language program. Like other system calls, ?DEBUG has both error and normal return statements. The *AOS/VS Programmer's Manual* presents the format for the ?DEBUG system call.

When your program encounters the ?DEBUG call at runtime, control passes to the AOS/VS Debugger utility. DEBUG presents the four accumulator values and the carry bit followed by the Debugger prompt (underscore):

```
AOS/VS Debugger - Rev. 001.000.000.000
00000000000 00000000000 00000000000 00000000000 00000000000
_
```

Whenever DEBUG displays its prompt, you may issue a Debugger command.

If you enter the Debugger from your program (i.e., via ?DEBUG), you must adhere to the following rules:

- you may *not* modify the contents of memory
- you may *not* set breakpoints

All other DEBUG commands are legal. You may examine the contents of memory and examine or modify the various register values.

Note that your program may issue a ?DEBUG system call when you are already in a debugging session. That is, you may issue a CLI DEBUG command and then begin or continue program execution. During execution, your program may issue a ?DEBUG system call and the Debugger will again get control. The above ?DEBUG restrictions do *not* apply when you enter the AOS/VS Debugger in this manner.

As a final note, when you enter the Debugger with a ?DEBUG call, the program counter (PC) holds the address of ?DEBUG's normal return statement, *not* your program's starting address.

Debugger Commands

After the Debugger displays its prompt (the underscore character), you can enter a *Debugger command* -- a command that directs the Debugger to perform some operation.

Appendix C lists and briefly describes all the Debugger commands; detailed descriptions appear throughout this manual.

Command Format

Debugger commands conform to the following general format:

[arg]... $X

where:

arg	is one or more arguments to the Debugger command. Not all commands permit arguments; some allow several. If you supply more than one argument, separate them by commas.
$	represents the escape character (ASCII code 33_8). Enter this character by pressing the ESC key on your keyboard. DEBUG echoes the ESC key with a dollar sign.
X	is a single uppercase alphabetic character from A to Z. This character represents a particular Debugger command.

When DEBUG encounters an escape character (i.e., $ in the above format), it expects a single command character to follow (i.e., X). As soon as you enter the command character, DEBUG immediately executes the command. You do *not* have to terminate Debugger commands with NEW LINE or carriage return.

DEBUG echoes the escape (ESC) character as a dollar sign ($) at your console. Consequently, in this manual, we use the symbol $ to represent the escape character, *not* the dollar character. When we want to refer to the dollar character (ASCII 44_8), we surround the dollar sign with angle brackets, <$>, to represent the dollar character. Refer to the Preface for a complete list of the notation conventions we use in this manual.

Though most Debugger commands conform to this general command format, the commands that open and close memory locations do not (i.e., commands /, \ , ↵ , ↓ , and ↑). Refer to Chapter 3 for a complete description of these commands.

Correcting Typing Errors

If you make a typing error while entering a value or command, you can correct it by using

- the RUBOUT or DEL (delete) key
- the CTRL-U command

Pressing the *RUBOUT* or *DEL* key removes the last character you typed.

Use the *CTRL-U* command to delete your entire entry. Typing CTRL-U deletes everything you entered since the last Debugger prompt. To enter a CTRL-U command, hold down the CTRL key and press U at the same time.

You can *not* delete the escape (ESC) character. If you make a mistake and have already hit the ESC key, enter a NEW LINE or carriage return. The debugger will return a question mark followed by a prompt. You can then enter the correct command.

Error Responses

Occasionally, you will enter a command or expression that the Debugger does not understand. DEBUG responds to errors in one of two ways:

- If the error involves an undefined symbol, DEBUG responds with *U?*
- For all other errors, DEBUG simply returns a question mark: *?*

After DEBUG returns *U?* or *?*, it displays the Debugger prompt (an underscore). Usually, you can readily detect your error. In these cases, simply enter the correct version of the command.

The following examples show *illegal* commands and DEBUG's error responses:

```
_429,439$S     9 is an illegal octal digit
?_
_FILE1$X       DEBUG cannot locate file FILE1
?UNABLE TO OPEN OR ACCESS SPECIFIED FILENAME
_START/ U?     Symbol START is undefined
_7$A ?_        There is no accumulator 7
```

If you do not know why the Debugger returned an error, you can issue the $? command. This command directs DEBUG to print a diagnostic error message for the last error it encountered. The command format is

$?

where:

$	represents the escape (ESC) character
?	directs DEBUG to display a diagnostic error message for the last error it detected

The following examples show illegal DEBUG commands followed by $? commands:

```
_429,439$S
?_ $?
ILLEGAL OCTAL DIGIT IN NUMBER
_START/ U? _$?
UNDEFINED SYMBOL
_7$A
?_ $?
UNRECOGNIZED COMMAND FORMAT
```

In each example, we entered the $? command after DEBUG's error response. DEBUG then displayed the appropriate error message.

Appendix A lists the various error messages DEBUG can return when you enter an $? command.

Starting Program Execution

When you enter the Debugger from the CLI, your program has not begun execution. To start program execution, issue the $R command as follows:

[address] $R

where:

address	is an optional number, symbol, or expression that indicates an address in your program
$	represents the escape (ESC) character
R	directs DEBUG to start executing your program

If you do not pass an address to the $R command, program execution begins at the current value of the program counter (PC). When you use the CLI DEBUG command to enter the Debugger, your PC value initially specifies the starting address for your program. If you enter the Debugger via ?DEBUG, the PC value indicates the address of ?DEBUG's normal return statement. See Chapter 3 for information on the program counter.

To begin program execution at an address other than the current PC value, pass an address to the $R command in the *address* argument. DEBUG will start your program at the supplied address.

After you issue a $R command, your program will run until it encounters a breakpoint or until it ends (i.e., executes a ?RETURN system call or terminates itself via ?TERM). If your program hits a breakpoint, execution stops and you receive a DEBUG prompt. At this point, you may examine or modify memory and registers. We describe breakpoints in Chapter 5.

To continue program execution after a breakpoint, issue a $R or $P command. You may *not* use the $P command when you first begin program execution; only use $P after breakpoints. See "Program Restarts" in Chapter 5 for more information.

Leaving the Debugger

Your debugging session may end in one of three ways:

- you may issue a $Z command
- you may enter a CTRL-D sequence
- the program you are debugging may end

Normally, you will terminate your debugging session by entering the following Debugger command:

```
$Z
```

where:

$	represents the escape (ESC) character
Z	directs DEBUG to terminate the debugging session

A $Z command terminates the Debugger and returns control to the father CLI process. $Z always returns control to the CLI, regardless of how you entered the Debugger (via the CLI DEBUG command or ?DEBUG system call).

Though the $Z command is the preferred means for leaving a debugging session, a CTRL-D sequence will also terminate the Debugger. CTRL-D is an end-of-file indicator and, when issued from your keyboard (i.e., from generic file @CONSOLE), ends your debugging session. To enter a CTRL-D sequence, hold down the CTRL key and press D.

If your program ends while you are in the Debugger, your debugging session also ends. That is, if you enter a $R or $P command (start or continue program execution) and your program encounters a ?RETURN system call statement before a breakpoint, DEBUG will terminate.

General Use Commands

The AOS/VS Debugger provides several *general use commands.* These commands perform useful functions not related to any specific debugging operations. We present them here, at the beginning of the manual, because they will be useful during all stages in your program debugging session.

Table 2-1 describes the general use commands.

Table 2-1. General Use Commands

Command	Action
$C	Generate a son CLI process (i.e., "push" to the CLI)
$H	Help command: list the various topics that DEBUG can supply information about
keyword$H	Help command: supply specific information about the topic identified by keyword
$Y	Close the current log file
logfile$Y	Close the current log file, if any, and open a new one
;comment↲	Enter the character string comment in the current log file

The remainder of this chapter describes each general use command in detail.

The Help Command

The Debugger utility supplies a help command that displays information about using the various DEBUG commands. There are two formats for issuing the help command:

- A general help command format that lists the various help topics
- A specific help command format that lists detailed information about particular topics

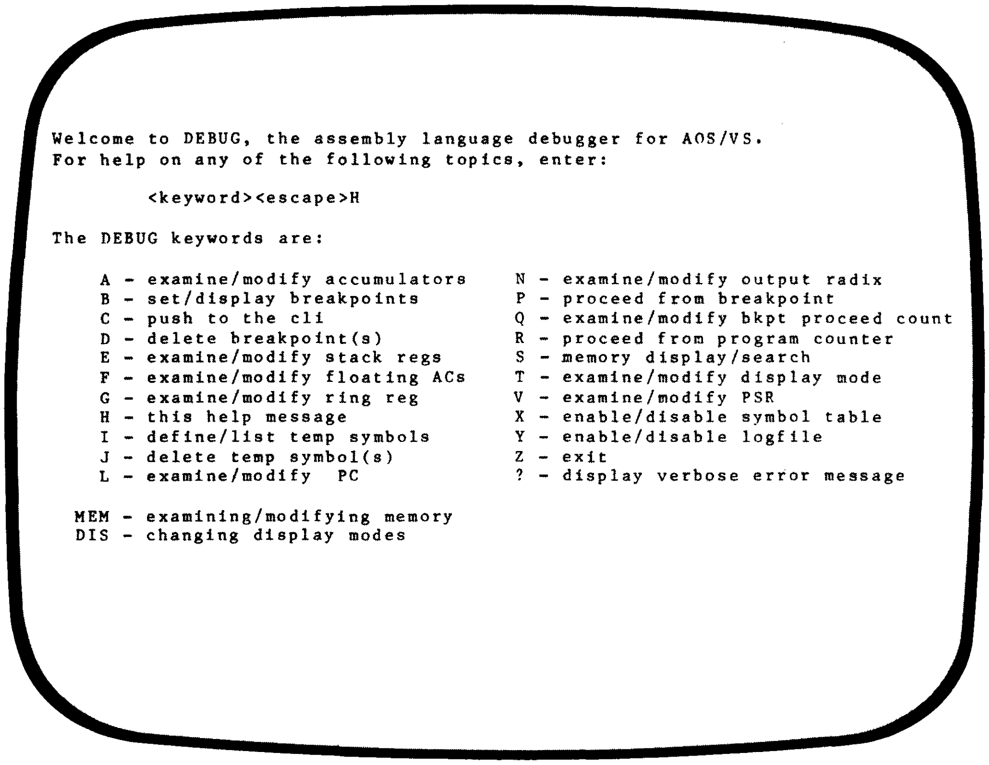
The format for issuing the general help command is

$H

where:

$	represents the escape (ESC) character
H	directs DEBUG to list the various help topics

When you issue a $H command, DEBUG lists all the topics it can supply more information about. Figure 2-1 shows the $H display.



```
Welcome to DEBUG, the assembly language debugger for AOS/VS.
For help on any of the following topics, enter:

        <keyword><escape>H

The DEBUG keywords are:

    A - examine/modify accumulators     N - examine/modify output radix
    B - set/display breakpoints         P - proceed from breakpoint
    C - push to the cli                 Q - examine/modify bkpt proceed count
    D - delete breakpoint(s)            R - proceed from program counter
    E - examine/modify stack regs       S - memory display/search
    F - examine/modify floating ACs     T - examine/modify display mode
    G - examine/modify ring reg         V - examine/modify PSR
    H - this help message               X - enable/disable symbol table
    I - define/list temp symbols        Y - enable/disable logfile
    J - delete temp symbol(s)           Z - exit
    L - examine/modify  PC              ? - display verbose error message

  MEM - examining/modifying memory
  DIS - changing display modes
```

SD-02306

Figure 2-1. $H Help Display

To obtain information about a specific help topic, enter the following:

keyword$H

where:

keyword	is a keyword that indicates which help topic you want information about (see Figure 2-1)
$	represents the escape (ESC) character
H	directs DEBUG to display information on the topic identified by **keyword**

As an example, to obtain information about fixed-point accumulator commands, enter

```
_A$H
```

DEBUG responds with a description of the various accumulator display and modify commands (i.e., $A, 0$A, 1$A, 2$A, and 3$A).

Saving the Debugging Session

Frequently, you will want to save a copy of your debugging session for future reference. For example, after you debug your program, you will probably want to modify your source file. By saving a copy of the debugging session, you can record all the changes that allow your program to run correctly.

The following sections explain how to record terminal activity (both your input and DEBUG's output) in a disk file, called the *log file.*

Opening a Log File

To open a log file, issue the command

```
logfile$Y
```

where:

logfile	is the pathname of the file that will contain a copy of the Debugging session
$	represents the escape (ESC) character
Y	directs DEBUG to record the debugging session in file **logfile**

When you issue this command, DEBUG opens file **logfile** and records all terminal activity (both your input and DEBUG's output) in that file.

If **logfile** already exists, DEBUG appends the debugging session to the existing file; otherwise, DEBUG creates **logfile**. If a log file is already open when you issue a **logfile$Y** command, DEBUG closes that file and opens the new one.

The following example shows how to use the **logfile$Y** command:

```
_DEBFILE$Y ⏎         Open file DEBFILE as the log file. Display the contents of locations 500₈ through
_500,510$S           510₈
16000000500/ 000000
16000000501/ 000001
16000000502/ 000002
16000000503/ 000003
16000000504/ 000004
16000000505/ 000005
16000000506/ 000006
16000000507/ 000007
16000000510/ 000010
_$Z                  End the Debugging session
```

After this debugging session, log file DEBFILE will contain the following:

```
_500,510$S
16000000500/ 000000
16000000501/ 000001
16000000502/ 000002
16000000503/ 000003
16000000504/ 000004
16000000505/ 000005
16000000506/ 000006
16000000507/ 000007
16000000510/ 000010
_$Z
```

Closing a Log File

You may want to record only part of your debugging session in a log file. DEBUG allows you to open and close your log file as often as you wish. The Debugger only records terminal activity when a log file is open.

To close a log file, issue the command

```
$Y
```

where:

$	represents the escape (ESC) character
Y	directs DEBUG to close the current log file

When DEBUG encounters a $Y command, it closes the log file. That is, DEBUG will stop recording the debugging session.

If you issue a $Y command and a log file is not open, DEBUG returns an error.

The following example shows how to open and close a log file:

```
_LIST1$Y ↵          Open file LIST1 as the log file. Display the contents of locations 470₈ through 475₈
_470,475$S
16000000470/ 000000
16000000471/ 000000
16000000472/ 000000
16000000473/ 000000
16000000474/ 000000
16000000475/ 000000
_$Y ↵               Close the current log file (i.e., LIST1). Examine AC0 and AC1
_0$A 00000000000 ↵
_1$A 00000000000 ↵
_LIST1$Y ↵          Re-open file LIST1 as the log file. Display the contents of locations 1000₈ through
_1000,1005$S        1005₈
16000001000/ 177777
16000001001/ 177777
16000001002/ 177777
16000001003/ 177777
16000001004/ 177777
16000001005/ 177777
_$Z                 End the debugging session
```

Because DEBUG only records the debugging session when a log file is open, after the above sequence of Debugger commands, file LIST1 contains the following:

```
_470,475$S
16000000470/ 000000
16000000471/ 000000
16000000472/ 000000
16000000473/ 000000
16000000474/ 000000
16000000475/ 000000
_$Y
_1000,1005$S
16000001000/ 177777
16000001001/ 177777
16000001002/ 177777
16000001003/ 177777
16000001004/ 177777
16000001005/ 177777
_$Z
```

Note that the log file does not contain the accumulator examine commands (0$A and 1$A). Again, DEBUG only records terminal activity when a log file is open.

Entering Comments in a Log File

Frequently, you will want to document the debugging changes you make to your program in your log file. This can save a great deal of time and effort when reviewing the log file at a later date.

To enter *comments* in a log file, type the following command:

```
;comment↵
```

where:

;	is a semicolon character that directs DEBUG to enter **comment** in the current log file
comment	is the ASCII character string that you want to place in the log file
↵	represents the NEW LINE character and terminates the **comment** string; you may use carriage return (↓) instead of NEW LINE (↵) if you wish

When DEBUG encounters a comment command, it simply copies the comment into the currently enabled log file, if any. The Debugger does not interpret comments and they do not influence the debugging session in any way.

The following example shows how to use the ;comment↵ command to document a debugging session:

```
        .
        .
        .
_LOG4$Y _↵
_;EXAMINE ACCUMULATORS↵
_$A   (AC0,AC1,AC2,AC3,CARRY)
00000000025 00000000050 00000000077 00000000100 00000000001
_;AC2 SHOULD EQUAL 75↵
_2$A 00000000077 _75↵
_;CONTINUE EXECUTION↵
_$P
1B 1600000003600
00000000025 00000000050 00000000075 00000000100 00000000001
_;SUCCESS!!↵
_$Z
```

After this Debugging session, log file LOG4 will contain the following:

```
_;EXAMINE ACCUMULATORS
_$A   (AC0,AC1,AC2,AC3,CARRY)
00000000025 00000000050 00000000077 00000000100 00000000001
_;AC2 SHOULD EQUAL 75
_2$A 00000000077 _75
_;CONTINUE EXECUTION
_$P
1B 1600000003600
00000000025 00000000050 00000000075 00000000100 00000000001
_;SUCCESS!!
_$Z
```

Pushing to the CLI

In certain situations, you may wish to execute CLI commands without terminating the current debugging session. DEBUG allows you to generate a son CLI process by entering the command

$C

where:

$	represents the escape (ESC) character
C	directs DEBUG to generate a CLI process

To issue the $C command, your user profile must give you the privilege to create two or more son processes.

When you enter $C, DEBUG generates a son CLI process (or pushes to the CLI). After you receive the CLI prompt,), you can enter CLI commands.

To return to your debugging session, enter the CLI command BYE. The BYE command terminates the son CLI process and returns control to DEBUG.

The following example shows how to use the $C command:

```
_$C
AOS/VS CLI  REV 01.00.00.00 19-MAY-80  11:33:52
) DIRECTORY↲
:UDD:JEFF
) WHO↲
PID: 9 JEFF                 009          :CLI.PR
) BYE↲
AOS/VS CLI TERMINATING  19-MAY-80  11:35:56
_
```

In this example, the $C command directed DEBUG to generate a CLI process. Once in the CLI, we entered two CLI commands: DIRECTORY and WHO. When we typed BYE, we terminated the CLI process and returned to the debugging session (as signaled by the DEBUG prompt _).

End of Chapter

Chapter 3 Accessing Memory

The Debugger allows you to view the contents of memory before your program begins execution and after encountering a breakpoint. DEBUG provides two methods for accessing the contents of your program's memory space. Each serves a different purpose and has its own command format. Generally, you will use the two methods to

- examine or modify the contents of a few specific memory locations
- display or search a wide range of memory locations

Table 3-1 gives more information.

Table 3-1. Memory Access Methods

	Examining and Modifying Memory	**Displaying and Searching Memory**
Command Syntax	address/ (One-word examine) address\ (Two-word examine)	add1,add2, *[cond],[incr],[size]* $S
Major Uses	*Examine* the contents of a few specific memory locations *Modify* the contents of a memory location	*Display* the contents of a wide range of memory locations *Search* for all occurrences of a value in a range of memory locations
Special Feature(s)	You may step forward or backward through memory (one or two words at a time)	You may specify an *increment* value so that DEBUG will only display every nth location You may specify a *size* value so that DEBUG will display one to four words of memory starting at each location

Before discussing these two memory access methods, we should briefly review some characteristics of AOS/VS memory and also describe the location counter and the program counter.

Memory

For addressing purposes, the basic unit of AOS/VS memory is the 16-bit *word.* The AOS/VS operating system allows your program to occupy up to 256MW or 512MB of memory (MW and MB mean megaword and megabyte and equal 1,048,576 $_{10}$ words and bytes, respectively). The area of memory accessible by your program is your *logical address space.*

Numeric memory addresses are in octal (base 8) and refer to word locations in your program. Location 0 contains the first word in your logical address space, location 1 holds the second word, etc. Thus, when we refer to location 525_8, we mean the 526_8 th 16-bit word in the current ring within your logical address space.

Rings

AOS/VS organizes memory into a hierarchical series of *rings,* numbered 0 through 7. Ring 0, the innermost ring, is superior in the hierarchy. Every program running under AOS/VS resides in one of these eight rings. Refer to the *AOS/VS Programmer's Manual* for a complete description of the AOS/VS ring structure.

The Debugger allows you to examine and modify any location in rings 4 through 7. In addition, you may access the unshared portion of ring 3, which contains your program's system tables. You can never access locations in rings 0 through 2, the system rings.

Your program usually resides in ring 7. Thus, you can always view the contents of your program's memory space.

AOS/VS Addresses

All *AOS/VS addresses* are 32-bits long. Bits 1 through 3 compose the ring field; it contains the address' ring number (i.e., a value from 0 through 7). Thus, the address 16000000355_8 specifies location 355_8 in ring 7. Note that 16000000355_8 contains the value 7 in bits 1 through 3.

When DEBUG presents an address value, it always displays the complete 32-bit number, ring field included.

Ring Register ($G)

Frequently, when you specify an address, you do not indicate a value for the ring field. For example, when you enter the value 100 as an address, you really mean location 100_8 in some ring, usually 7. Thus, 16000000100_8 is the address that you want to reference (ring 7, location 100).

When you enter an address without a ring value, DEBUG automatically supplies one for you. By default, DEBUG inserts the ring value of the initial program counter (i.e., the address where your program begins execution). DEBUG stores this default ring value in the *ring register.*

To view the current default ring value, enter the $G command as follows:

```
$G
```

where:

$	represents the escape (ESC) character
G	directs the Debugger to display the contents of the ring register

As an example, suppose your program resides in ring 7. When you enter the Debugger, the ring register will hold the value 7. You can check this by entering

```
_$G
```

DEBUG responds as follows:

```
_$G 00000000007 _
```

This shows 7 as the current default ring value. Thus, any time you enter an address that does not explicitly include a value in the ring field (bits 1 through 3), DEBUG will assume you are referring to ring 7.

After DEBUG displays the ring value, the ring register is *open;* that is, you may enter a new default ring value. After entering a value, close the ring register with a NEW LINE or carriage return.

For example, to change the default ring to 6, open the ring register with a $G command, enter the value 6, and close the register.

```
_$G 00000000007 _6↵
_
```

The default ring is now 6. You may verify this by entering $G:

```
_$G 00000000006 _
```

If you now enter an address that does not include a ring value, DEBUG assumes you are referring to ring 6. For example, DEBUG interprets address 100_8 as 14000000100_8 (ring 6, location 100).

As mentioned throughout this discussion, DEBUG only uses the ring register value if your address does not have a value in the ring field (bits 1 through 3). Thus, if you enter

```
_16000000100/
```

DEBUG would not refer to the ring register because your address explicitly indicates ring 7.

Note that DEBUG will return an error if you try to address an illegal location. Unless you have explicitly issued a ?RINGLD system call, all your program resides in ring 7. Thus, references to locations in other rings will result in errors. For this reason, you will not usually modify the value in the ring register.

Ring Field Operator (;)

In addition to the ring register ($G), DEBUG provides the ring field operator ; (semicolon). You may use ; to insert a value in the ring field of an address. For example, DEBUG expands the expression 100;5 into 12000000100_8 (location 100 in ring 5).

If you enter the expression 100;5 as an address, DEBUG does not refer to the ring register because 100;5 explicitly indicates a ring value. Refer to Chapter 7 for a complete description of the ring field operator (;).

Location Counter and Program Counter

The *location counter* is an internal Debugger variable; it holds the address of the location that DEBUG most recently displayed. For example, if you view the contents of location 537_8, then the location counter equals address 537_8 until you display another location.

The single-character symbol . (period) represents the value of the location counter. You may use this symbol in any expression or anywhere you would use a 32-bit numeric value.

For example, you could use the location counter symbol to deposit an address in either a register or memory. Suppose you view location 110_8 and wish to place a pointer to that location in accumulator 0. You could open that accumulator with the 0$A command (see Chapter 4) and enter a period. This would deposit the address of the last location DEBUG displayed (110_8) into accumulator 0.

You will frequently use the location counter symbol in conjunction with the indirect operators @ and # . @ extracts a one-word value at the supplied address; # indicates the two-word value starting at the supplied address (see Chapter 7 for a complete description). The expression @ . equals the one-word value at the address in the location counter; # . equals the two-word value starting at the same address. You will find these two expressions particularly useful when issuing search ($S) commands since you usually want to compare the contents of the current location (i.e., @ . or # .) with some other value (see "Searching Memory" later in this chapter).

Do not confuse the location counter with the program counter. The *program counter (PC)* is a register that holds the address of the next instruction in your program that will be executed. Thus, if you stop program execution before the instruction in location 200_8, the program counter's value equals address 200_8. During this pause in program execution, you may examine location 405_8. At this point in the Debugging session, the program counter holds the address 200_8 (the address of the next instruction to be executed) and the location counter holds address 405_8 (the last address DEBUG displayed).

Again, it is the location counter you may access with the symbol . (period). The $L command allows you to examine the program counter (see Chapter 4) and the symbol < $ > L represents the program counter's value (see Chapter 6).

Examining and Modifying Memory

The commands we discuss in this section allow you to examine and optionally modify the contents of specific memory locations. You will perform two or possibly three operations when using these commands:

1. *Open a one- or two-word location* by supplying an address value to a memory examine command (i.e., / or \ ). When you open a location, DEBUG displays the contents of that location.

2. *Modify the contents of the open location.* When a location is open, you may deposit a new value. This is an optional step; you need not alter the contents of memory.

3. *Close the open location* by entering a NEW LINE, carriage return, or uparrow character, or by entering a new DEBUG command. Depending on how you close the location, DEBUG may open the following location, open the previous location, or execute a new command.

The following section, "Examining Memory Locations", explains how to open and close memory locations (steps 1 and 3 above). It also presents the syntax for using the / and \ commands. "Modifying Memory Locations" describes how to enter new values into memory (step 2).

If you enter the Debugger via a ?DEBUG system call, you may *not* modify memory (step 2). You may, however, examine the contents of any location. (See "Entering the Debugger from a Program" in Chapter 2.)

Examining Memory Locations

DEBUG provides two general formats for examining memory locations:

address/	One-word (16-bit) examine
address\	Two-word (32-bit) examine

where:

address	is a number, symbol, or expression whose value specifies a word location in your logical address space; you will frequently place a label in this argument
/	is a slash and directs DEBUG to display the one-word (16-bit) value at location **address**
\	is a backslash and directs DEBUG to display the two-word (32-bit) value starting at memory location **address**; that is, DEBUG will present the contents of word locations **address** and **address+1**

When you enter one of these commands, the Debugger will display the appropriate value on the same line followed by a prompt. For example, suppose your logical address space contains the following values (all numbers are octal and all locations are ring 7):

Memory (Word) Locations	Contents
500	000001
501	000002
502	000003
503	000004
504	000005
505	000006

If you enter the command

```
_500/
```

the Debugger will return the one-word value at location 500. Thus, the command and the Debugger response would be

```
_500/ 000001 _
```

Similarly, if you enter the two-word examine command

```
_500\
```

the Debugger returns the contents of word locations 500 and 501 as a single 32-bit integer as follows:

```
_500\ 00000200002 _
```

After DEBUG displays the contents of a memory location, that location is *open;* that is, you may deposit a new value into that location, if you wish. We explain how to place new values in memory next, "Modifying Memory Locations".

After you examine (open) a location, you must *close* that location by entering one of the following:

- NEW LINE (⤸)
- carriage return (↓)
- uparrow (↑)
- a DEBUG command

In the following discussion, we describe each of these four methods for closing locations; Table 3-2 provides a brief summary.

Table 3-2. Closing Memory Locations

Closing Method	Representation in this Manual	Meaning
NEW LINE (ASCII value 012_8)	⤸ (curved arrow)	Close location
carriage return (ASCII value 015_8)	↓ (down arrow)	Close location and open subsequent location
uparrow (ASCII value 136_8)	↑	Close location and open previous location
A DEBUG command		Close location and execute new command

NOTE: If your keyboard layout does not conform to the ANSI standard, the meanings for NEW LINE and carriage return are reversed. Be sure to read "Differences Between Consoles" in Chapter 1.

The *NEW LINE* character simply closes the open location and returns the DEBUG prompt. For example,

```
_502/ 000003 _↲
_
```

Examine location 502.

Closing a one-word location with *carriage return* automatically instructs DEBUG to open the next location in memory. In this case, DEBUG will skip down a line and present the next address and its contents.

When displaying address values, DEBUG always writes the complete 32-bit address, including the ring value. For example, DEBUG will print 16000000502, not simply 502, when displaying address 502 in ring 7.

Also, DEBUG presents addresses in symbolic mode, by default. However, in this chapter we show DEBUG presenting numeric addresses since each program references a different symbol table. Refer to Chapter 6 for more information on symbols and symbol tables.

The following example shows how to use the carriage return to step forward through memory:

```
_500/ 000001 _↓
16000000501/ 000002 _↓
16000000502/ 000003 _↲
_
```

Examine locations 500 through 502. DEBUG does not display location 503 since we closed 502 with a NEW LINE, not a carriage return

When addressing double-word locations (via \ ), DEBUG will display the next 32-bit quantity if you close with a carriage return.

```
_500\ 00000200002 _↓
16000000502\ 00000600004 _ ↓
16000000504\ 00001200006 _↲
_
```

Examine sequential two-word locations starting with address 500

If you close a one-word location with the *uparrow* character, DEBUG automatically opens the previous location. For example,

```
_502/ 000003 _ ↑
16000000501/ 000002 _ ↑
16000000500/ 000001 _↲
_
```

Examine locations 502, 501, and 500

If you close a two-word location with uparrow, DEBUG will display the previous 32-bit quantity.

```
_504\ 00001200006 _ ↑
16000000502\ 00000600004 _ ↑
16000000500\ 00000200002 _↲
_
```

Examine the 32-bit values starting at addresses 504, 502, and 500

The last way you may close a location is by entering a DEBUG command. The Debugger will automatically close the location and execute the new command. For example, suppose you open location 502:

```
_502/ 000003 _
```

Open location 502

You may close that location by entering a DEBUG command as follows:

```
_502/ 000003 _0$A 00000000177 _
```

Close location 502 with the 0$A command. Then, display the contents of accumulator 0

The next example closes a location with another memory display command.

```
_500/ 000001 _504/ 000005 _
```

First, open location 500. Since we enter the command 504/, DEBUG closes location 500 and opens 504

Closing a Location with / or \

In the previous section, we said you could close a memory location by entering a new DEBUG command. Thus, you can close an open location by issuing another / or \ command. If you close with the single characters / or \ , DEBUG uses the contents of the open location as an address and opens the new location.

For example, suppose you examine the word at location 550_8.

```
_550/ 000720 _
```

The word at location 550_8 is now open. If you close with / or \ , DEBUG interprets 720_8 as an address and opens one or two words at that location. For example,

`_550/ 000720 _ / 001025 _`	Open location 550. Since we close with /, DEBUG interprets 720 as an address and opens that location (i.e., the value at location 720 equals 1025)

There is no limit to the number of indirections you may perform. As an example, suppose your logical address space appears as follows:

Location	Contents
5000	5001
5001	5002
5002	5003
5003	5004
5004	5005

Initially, you may open location 5000.

`_5000/ 005001 _`	Open location 5000

If you close with /, DEBUG interprets the contents of location 5000 as an address and opens that location.

`_5000/ 005001 _ / 005002 _`	Close location 5000 with /; open location 5001

You can continue to perform address indirections by closing each location with the / command.

```
_5000/ 005001 _ / 005002 _ / 005003 _ / 005004
```

In this example, the contents of each location is, in turn, used as an address.

Examining Locations in Various Display Modes

In all the previous examples, the Debugger displays the contents of memory locations as 16-bit or 32-bit integers; that is, by default, the Debugger presents values in the numeric display mode. You may direct DEBUG to present the contents of memory in a variety of other display modes (e.g., symbolic, half-word, ASCII, instruction, floating point, etc.). Chapter 8 describes the various display modes available.

In most modes, DEBUG abides by the size constraints associated with the / and \ commands. Thus, if you are half-word display mode, the / command still directs DEBUG to present 16-bits of memory; however, now the Debugger presents that word as two 8-bit integers. For example, instead of

`_500/ 000001 _`	Numeric display mode

DEBUG displays the contents of address 500 as follows:

`_500/ 000 001 _`	Half-word display mode

Similarly, the \ command still examines two words of memory but DEBUG presents those words in the current display mode.

There are two display modes for which the / and \ commands do *not* present one and two words of memory, respectively:

● floating point display mode
● instruction display mode

If DEBUG is in *floating point display mode,* the / command presents *two* words of memory as a single precision (32-bit) floating point number. The \ command displays *four* words of memory as a double precision (64-bit) floating point number. The following examples show the use of / and \ in floating point display mode:

```
_500/ 0.0000000E+0 _↓
16000000502/ 1.5213623E-69 _↓
16000000504/ -1.3036937E-32 _↵
_
```

In floating point display mode, the / command opens two words of memory

```
_500\ 0.0000000000000000E+0 _↓
16000000504\ -1.3036898356298341E-32 _↓
16000000510\ 2.8337356298367154E-78 _↵
_
```

The \ command opens four words of memory when DEBUG is in the floating point display mode

If DEBUG is in the *instruction display mode,* the / and \ commands direct the Debugger to present the complete instruction starting at the supplied address. Since instructions are of variable length (from one to four words), DEBUG does not use the one- and two-word display properties of the / and \ commands but rather opens as many locations as necessary to present the complete instruction. Stepping forward through memory (via the carriage return) displays consecutive instructions, not sequential one- or two-word values. The following examples will help clarify this discussion (instruction display mode):

```
_500/ XWLDA 0,1 _↓
16000000502/ LCALL 0,1,4 _↓
16000000506/ WADD 0,1 _↓
16000000507/ WBR 530 _↵
_
```

DEBUG presents consecutive instructions, not consecutive words

```
_500\ XWLDA 0,1 _↓
16000000502\ LCALL 0,1,4 _↓
16000000506\ WADD 0,1 _↓
16000000507\ WBR 530 _
```

For displaying purposes, the / command operates the same as the \ command when in the instruction mode

Modifying Memory Locations

As we have seen, when you enter an address followed by a / or \ character, DEBUG displays the one- or two-word value starting at that address. After DEBUG displays the contents of a memory location, that location is *open;* that is, you may deposit a new value, if you wish. Simply enter the new value on the same line as the Debugger's prompt and close that location.

Every time you modify memory, your console beeps. That is, when you modify the contents of a memory location, DEBUG sends a bell character (ASCII value 7) to your console.

As an example, suppose you want to place the value 23 in memory location 500_8. First, open that location. After DEBUG displays the contents of location 500, enter the value 23 and close the location.

```
_500/ 000001 _23↵
_
```

Subsequent examinations of location 500 will reveal the value 23.

```
_500/ 000023 _↵
_
```

If a two-word location is open, a new value will replace both words of memory. For example,

```
_500\ 00000200002 _23↵
_
```

The two-word value starting at address 500 now equals 23, as shown in the following examination:

```
_500\ 00000000023 _↵
_
```

In the previous examples, we closed all open locations with the NEW LINE character. As discussed earlier, NEW LINE directs DEBUG to enter the value in memory, close the location, and return a prompt.

If you wish to modify the contents of more than one location, you may close locations with either a carriage return or the uparrow character. Again, a carriage return closes the current location and opens the next one; the uparrow closes the current location and opens the previous one.

Using a carriage return, you may modify consecutive locations without having to specify each address. For example,

```
_501/ 000002 _12↓
16000000502/ 000003 _13↓
16000000503/ 000004 _14↵
_
```

Open location 501 and deposit the value 12. Since we close with a carriage return, DEBUG opens the next location (502). Again, we deposit a new value and close with a carriage return. Since we close the last location (503) with a NEW LINE, DEBUG returns a prompt

Similarly, using the uparrow, you may step backward through memory modifying the contents of each location.

```
_503/ 000014 _4↑
16000000502/ 000013 _3↑
16000000501/ 000012 _2↵
_
```

Open location 503 and deposit the value 4. Since we close with an uparrow, DEBUG opens the previous location (502). Again, we deposit a value and close with an uparrow. Since we close the last location (501) with a NEW LINE, DEBUG returns a prompt

As mentioned earlier, you need not modify each open location. Thus, using a carriage return or the uparrow, you can step through memory modifying select locations. For example,

```
_501/ 000002 _↓
16000000502/ 000003 _↓
16000000503/ 000004 _100↓
16000000504/ 000005 _↵
_
```

Examine the contents of locations 501 through 504. Modify the contents of 503 (i.e., deposit 100) but do not alter other locations

Our examples have mainly used the one-word examine command (/). You may use carriage return and uparrow to modify consecutive two-word locations as well. For example,

```
_500\ 00000000000001 _10↓
16000000502\ 00000000000002 _20↓
16000000504\ 00000000000003 _30↵
_
```

Deposit the two-word values 10, 20, and 30 starting in locations 500, 502, and 504, respectively. DEBUG does not open location 506 because we close 504 with a NEW LINE, not a carriage return

Legal Entry Values

In the above examples, we entered integer values when modifying memory. You may also enter symbols, expressions, floating point numbers, and instructions. The Debugger will compute the appropriate value and deposit it in the open location.

If you enter a number, symbol, or expression that is too large for the open location, DEBUG returns an error. In the following example, we open a one-word location and enter a two-word value:

```
_500/ 000000 _?
```

Since only one word of memory is open, DEBUG returns an error and location 500_8 remains unchanged.

If you enter a floating point number or instruction that is too long for the open location, DEBUG does *not* truncate the value. Instead, DEBUG inserts the complete value, modifying as many words of memory as are necessary. For example, suppose you open one word of memory and insert a two-word instruction:

```
_500/ 000000_XWLDA 0,1 )
_
```

DEBUG will insert the complete two-word value of the XWLDA instruction starting at address 500_8 even though only one word of memory is open. Thus, the above command sequence modifies locations 500 and 501.

For more information on symbols, refer to Chapter 6. Chapter 7 provides more information on expressions, instructions, and floating point numbers.

Permanent Memory Modifications

Generally, the values you deposit in memory are temporary. They are present during the debugging session, but are not transferred to the program file on disk. The next time you execute your program, you will find the original contents of memory unchanged by the debugging operations.

There is one exception to this rule: *If you modify the contents of a shared memory location, that change will permanently alter the program file.*

A shared page of memory is accessible by more than AOS/VS process at a time -- only one copy of the shared page exists in physical memory. AOS/VS always copies modified shared pages into the program file on disk. Thus, all modifications to a shared page will permanently alter the program file.

In addition, since more than one AOS/VS process can access a shared page in memory, changes you make in shared memory will affect all users. For example, if you set a breakpoint in a shared page, all processes executing that page will stop at the breakpoint (regardless of whether they are in the Debugger).

In general, you should make sure no one else is using a shared program while you are debugging it.

The *AOS/VS Link and LFE User's Manual* and the *AOS/VS Macroassembler (MASM) Reference Manual* provide information on the shared and unshared portions of your logical address space and also explain how to place code in these two areas of memory.

Displaying and Searching Memory

The commands we discussed in the previous section (i.e., / and \ ) are useful for examining a small number of memory locations. In this section, we will describe the $S command which allows you to

- display the contents of a large number of locations with a single command
- search for all occurrences of a value within a given memory range

The complete command syntax for the $S command is

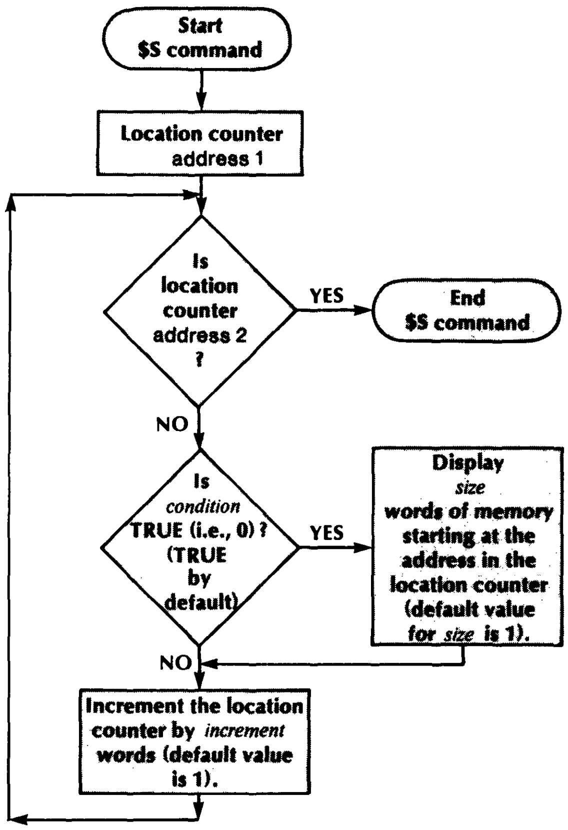
address1,address2, *[condition],[increment],[size]* $S

where:

address1	is a number, symbol, or expression that specifies the starting location for the memory display or search

address2	is a number, symbol, or expression that specifies the final location for the memory display or search
condition	is an optional expression that serves as a condition for performing memory searches. DEBUG displays a location only if *condition* is true (i.e., does not equal 0) for the current address. The default value for *condition* is true. In general, you will supply a relational expression in the *condition* argument
increment	is an optional value that specifies how much DEBUG should increment the location counter between each display or search. DEBUG will add the increment value to the location counter to determine the next address for the search or display. The default value for *increment* is 1 -- display or search consecutive 16-bit words of memory
size	is an optional value that indicates how many words of memory DEBUG will present at each address in the display or search. *size* must be between 0 and 4, inclusive. The default value is 1 -- display the one-word value at each location. Only use a *size* value of 0 when displaying instructions (see discussion in the next section)
$	represents the escape (ESC) character
S	directs the Debugger to perform a display/search operation

The following sections present a detailed description of the $S command. Figure 3-1 gives an overview of the operations DEBUG performs during a $S command.



Start $S command
Location counter address 1
Is location counter address 2 ?
YES
End $S command
NO
Is condition TRUE (i.e., 0)? (TRUE by default)
YES
Display size words of memory starting at the address in the location counter (default value for size is 1).
NO
Increment the location counter by increment words (default value is 1).

SD-02280

Figure 3-1. The $S Command

Before discussing the various uses of the $S command, we must inject a word of caution. *You may NOT interrupt a $S command without aborting your debugging session.* The only time DEBUG stops a $S command is when it encounters an illegal address. In this case, DEBUG stops the memory display or search, generates an appropriate error message, and returns a prompt.

Displaying Memory

The simplest form of the $S command is

```
address1,address2$S
```

This command directs DEBUG to display the one-word contents of each memory location from **address1** to address2, inclusive. Thus, to display the contents of locations 100 through 200, enter the command

```
_100,200$S
```

The Debugger would respond as follows (default ring is 7):

```
16000000100/ 000000
16000000101/ 000000
16000000102/ 000000
          .
          .
          .
16000000176/ 000000
16000000177/ 000000
16000000200/ 000000
```

DEBUG displays one word of memory at each location, by default. If you wish to view more words at each address, use the *size* argument. For example, the following command displays all two-word values starting at addresses 100 through 200:

```
100,200,,,2$S
```

Note that we had to enter extra commas in the $S command so DEBUG would know that 2 is a *size* argument, not a *condition* or *increment* value (see the discussion of the $S command).

After you enter this command, DEBUG would respond as follows:

```
16000000100/ 00000000000
16000000101/ 00000000000
16000000102/ 00000000000
             .
             .
             .
16000000176/ 00000000000
16000000177/ 00000000000
16000000200/ 00000000000
```

Again, the Debugger displays the two-word value starting at each one-word address from 100 to 200. That is, the Debugger presents the two-word values in locations 100 and 101, 101 and 102, 102 and 103, etc.

In many cases, you want to display consecutive, nonoverlapping two-word values. That is, you wish to view locations 100 and 101, 102 and 103, 104 and 105, etc. You may accomplish this by using the ***increment*** argument in the $S command. For example,

```
_100,200,,2,2$S
```

The *increment* of 2 directs DEBUG to increment the address by 2 words before each display. Therefore, this command directs DEBUG to display the two-word values starting at every other address from 100 to 200. DEBUG would respond to this command as follows:

```
16000000100/ 00000000000
16000000102/ 00000000000
16000000104/ 00000000000
          .
          .
          .
16000000174/ 00000000000
16000000176/ 00000000000
16000000200/ 00000000000
```

You may supply an *increment* without a *size* if you wish to view every other one-word value. For example, the command

```
_100,200,,2$S
```

directs the Debugger to display the one-word contents of every other location from address 100 through 200 as follows:

```
16000000100/ 000000
16000000102/ 000000
16000000104/ 000000
          .
          .
          .
16000000174/ 000000
16000000176/ 000000
16000000200/ 000000
```

Displaying Instructions

As we noted earlier, DEBUG always presents the contents of a location in the permanent display mode (see Chapter 8). Thus, if you are in the instruction display mode (mode 3), DEBUG presents the contents of each location as an instruction. For example, if you enter

```
_200,300$S
```

the Debugger would respond as follows:

```
16000000200/ XWLDA  0,B
16000000201/ LEF    0,77
16000000202/ LCALL  0,1,4
16000000203/ JMP    0
16000000204/ JMP    1
16000000205/ JMP    4
16000000206/ WBR    ARP
          .
          .
          .
```

The problem with this presentation is that the Debugger interprets the contents of each location as the beginning of an instruction. In reality, instructions are of variable length (1 to 4 words long) and many words are argument fields for other instructions. In the previous example, locations 203 through 205 contain the displacement value and argument count for the four-word LCALL instruction starting at location 202. However, DEBUG interpreted each of the four words as an instruction and displayed three nonexistent JMP statements.

To account for this exceptional case, the Debugger provides a special value for the *size* field of the $S command. If you specify a display *size* of 0, DEBUG will present consecutive instructions, not locations. The Debugger will only present those locations that contain the first word of an instruction. Thus, the command

```
_200,300,,,0$S
```

produces the following output:

```
16000000200/ XWLDA  0,B
16000000202/ LCALL  0,1,4
16000000206/ WBR    ARP
          .
          .
          .
```

DEBUG does not present locations 201, 203, 204, and 205 because those words are argument fields for the XWLDA and LCALL instructions.

When specifying a *size* value of 0, you must adhere to the following rules or DEBUG returns an error:

- you may not supply *condition* or *increment* values in the $S command
- the permanent display mode must be set to instruction (see Chapter 8)

Searching Memory

In addition to displaying a range of memory locations, the $S command can also search memory for all occurrences of a specific value. To perform a memory search, enter an expression in the *condition* argument to $S. The Debugger evaluates the condition for each value of the location counter from **address1** to **address2**, taking into account the *increment* and *size* arguments (again, see the $S command syntax discussion). If the value of the expression is not 0, the condition is true and DEBUG displays the contents of the current location. If the expression equals 0, the condition is false and DEBUG moves on to the next address in the $S sequence without displaying the contents of the current location (see the flowchart in Figure 3-1).

Generally, your *condition* argument will be a relational expression (i.e., will include one of the relational operators =, < >, <=, >=, <>). In addition, your expression will usually refer to the contents of the current location via one of the indirect operators, @ or # . Briefly, the @ operator extracts the one-word contents of the supplied address; # indicates the two-word value starting at the supplied address. (Chapter 8 describes all DEBUG operators.)

A few examples will help clarify the use of the *condition* argument. Suppose you want to find every occurrence of the one-word integer 7 in a given address range. You would enter the command

```
_100,200,@.=7$S
```

@ . equals the one-word value at the current location. Thus, the condition @ .=7 is true only if the contents of the current location equals 7. If locations 104, 143, 174, and 177 all contain sevens, DEBUG would respond as follows:

```
16000000104/ 000007
16000000143/ 000007
16000000174/ 000007
16000000177/ 000007
```

You may use a condition in conjunction with *size* and *increment* values. For example, to search double words of memory for values of 50 or more, enter the following command:

```
_100,200,#.>=50,2,2$S
```

# . equals the two-word value starting at the current location. Thus, the condition # . > =50 is true when the two-word value starting at the current location is greater than or equal to 50. The increment and size values of 2 direct DEBUG to search consecutive two-word values from location 100 to location 200 (i.e., the two-word values in locations 100-101, 102-103, 104-105,...,200-201). The Debugger would respond to the previous command as follows:

```
16000000106/ 00000000050
16000000124/ 00000233570
16000000142/ 00000000110
16000000144/ 00000000051
16000000174/ 0000002271
```

To search memory for a specific instruction, you must test the instruction's assembled value in the *condition* argument. For example, to search for the instruction ADD 0,0, enter the following command

```
_100,200,@.=103000$S
```

This command directs DEBUG to search the contents of each memory location from address 100 through 200 for the one-word integer 103000, the assembled value for instruction ADD 0,0. If the permanent display mode is set to instruction, the Debugger's response to the previous command might be

```
16000000105/ ADD 0,0
16000000113/ ADD 0,0
16000000126/ ADD 0,0
16000000154/ ADD 0,0
16000000171/ ADD 0,0
```

To obtain the assembled value of an instruction, enter the instruction and hit the numeric function key (F1). DEBUG will immediately return the instruction's value. Refer to the sections of Chapter 8 that describe local display commands for more information.

End of Chapter

Chapter 4
Accessing Registers

The AOS/VS Debugger allows you to examine and modify the contents of two types of registers:

- machine state registers
- Debugger registers

Machine state registers are internal variables that contain program status information. DEBUG allows you to examine and modify the following machine state registers: fixed-point accumulators, carry bit, processor status register, floating point accumulators, floating point status register, program counter, and stack control registers.

Debugger registers are internal variables that DEBUG uses when performing certain operations. You may examine and modify the following Debugger registers: display mode, proceed count, output radix, and ring.

Next, we explain how to examine and modify the contents of a register. This is a general discussion providing information common to all registers.

After the general discussion, we describe machine state and Debugger registers in more detail.

Examining and Modifying Registers

The syntax for examining a register is

[n] $W

where:

n	is an integer value; only certain register examine commands accept an argument
$	represents the escape (ESC) character (ASCII value 33_8)
W	is a single letter representing a register or class of registers (see Table 4-1)

Table 4-1 lists the commands you use to access the various registers.

Table 4-1. Register Command Summary

Registers	Command	Register(s)	Action
Machine State	$A	Accumulators and Carry Bit	Display the contents of the four fixed-point accumulators and the carry bit
	n$A	Accumulator or Carry Bit	If 0<=n<=3, open fixed-point accumulator n; if n=4, open the carry bit
	$E	Stack Registers	Display the four stack registers
	n$E	A Stack Register	Open the stack pointer (n=0), frame pointer (n=1), stack limit (n=2), or stack base (n=3) register
	$F	Floating Point Accumulators and FPSR	Display the four floating point accumulators and the floating point status register (FPSR)
	n$F	Floating Point Accumulator or FPSR	If 0<=n<=3, open floating point accumulator n; if n=4, open the first 32 bits of the FPSR; if n=5, display the last 32-bits of the FPSR (i.e., the floating point PC)
	$L	Program Counter	Open the program counter
	$V	Processor Status Register	Open the processor status register
Debugger	$G	Ring	Open the ring register
	$N	Radix	Open the output radix register
	n$Q	Proceed Count	Open the proceed count register for breakpoint n
	$T	Display Mode	Open the global display mode register

After you enter a register examine command, DEBUG displays the contents of that register or group of registers. For example, to view the current value of the program counter (PC), enter

```
_$L
```

DEBUG responds on the same line with the PC value:

```
_$L 16000000446 _
```

Similarly, to view the fixed-point accumulators and carry bit, enter

```
_$A
```

DEBUG responds by displaying the contents of all four accumulators and the carry bit as follows:

```
_$A  (AC0,AC1,AC2,AC3,CARRY)
00000000000 00000000000 00000000000 00000000000 00000000000
_
```

If you wish to view only accumulator 2 (AC2), you must precede the escape character with the value 2:

```
_2$A
```

This instructs DEBUG to display the contents of AC2 as follows:

```
_2$A 00000000000 _
```

Table 4-1 shows which register examine commands display a single register and which commands display a group of registers.

After DEBUG displays the contents of a *single* register, that register is *open;* that is, you may deposit a new value, if you wish. Simply enter the new value after the Debugger prompt.

For example, to change the program counter, open that register and deposit the new value as follows:

`_$L 16000000446 _16000000555`	Open the location counter register and deposit the value 16000000555

When modifying a register, you may enter an integer, a symbol, or an expression. If the value you enter is too large, DEBUG truncates the high-order (most significant) bits. If your entry value is too small, DEBUG pads on the left with zeros. There is one exception to these rules: if you enter a floating point number or assembly language instruction that is too large for the open register, DEBUG returns an error and does not alter the register value.

If you modify the contents of a register, you must close that register before performing other Debugging operations. Normally, you will enter a NEW LINE (↓) or carriage return (↓) to close a location.

`_4$A 00000000001 _0↓`
`_`
	Open the carry bit, deposit the value 0, and close with NEW LINE

If you do not modify the contents of a register, you can enter a new command immediately after DEBUG displays a register's value. That is, you need not use a NEW LINE or carriage return to close the register if you do not enter a new value. For example,

`_$L 16000000555 _2$A 00000000000 _`	First, we display the contents of the PC register. Then, we immediately enter the 2$A command to open AC2

Similarly, you can use the memory examine commands (/ and \ ) immediately after DEBUG displays a register. These commands close the register and open the location identified by the register's contents. For example,

```
_$L 16000000555 _ / 000177 _
```

In this example, we first direct DEBUG to display the contents of the program counter (PC) by entering the command $L. Since we close the PC register with a slash command (/), DEBUG interprets the contents of the PC as an address and opens one word of memory at that location (in this case, location 555_8). Thus, the 16-bit value starting at location 555_8 equals 177_8. Refer to Chapter 3 for more information on memory access and the memory examine commands / and \ .

Machine State Registers

You may examine and modify the following machine state registers:

- the four fixed-point accumulators (ACs)
- the carry bit
- the processor status register (PSR)
- the four floating point accumulators (FPACs)
- the floating point status register (FPSR)
- the four stack registers
- the program counter (PC)

The following sections of this chapter explain how to access these registers. This manual does *not* describe the use of the various machine state registers within your program; refer to the *ECLIPSE MV/8000 Principles of Operation* manual for that information.

The Debugger provides a set of special symbols that represent the contents of the MV/8000 machine state registers. See "Special Debugger Symbols" in Chapter 6 for a description of these symbols.

Accumulators and Carry

To examine the contents of the four fixed-point accumulators and the carry bit, enter the following command:

```
$A
```

where:

$	represents the escape (ESC) character
A	directs DEBUG to display the accumulators and carry bit

When displaying them as a group, DEBUG always presents the four fixed-point accumulators and carry in numeric mode, regardless of the current display mode.

An example command and response is

```
_$A  (AC0,AC1,AC2,AC3,CARRY)
00000000000 00000000000 00000000000 00000000000 00000000000
_
```

The $A command does *not* open the registers for modification, but simply displays their contents.

To open a specific accumulator or the carry bit, enter

```
n$A
```

where:

n	is an integer value from 0 to 4, inclusive. If n equals 4, DEBUG opens the carry bit; otherwise, DEBUG opens the accumulator specified by n (i.e., ACn)
$	represents the escape (ESC) character
A	informs DEBUG that you wish to access an accumulator or the carry bit

When displaying a single register, DEBUG uses the current display mode.

As an example, to open accumulator 1 (AC1), enter

```
_1$A
```

The Debugger will display the contents of AC1 and open that accumulator for modification. Enter a new value, if you wish, and close the register. For example,

```
_1$A 00000000000 _177↵        Open AC1, deposit the value 177, and close with NEW LINE
_
```

To open the carry bit, enter the command

```
_4$A
```

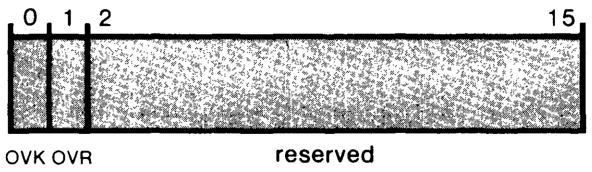
DEBUG displays the carry value and opens that bit for modification. The carry value must be either 0 or 1; if you enter a different value, DEBUG returns an error. The following example opens and modifies the carry bit:

```
_4$A 00000000001 _0↵        Open the carry bit, deposit the value 0, and close with NEW LINE
_
```

Processor Status Register

The processor status register (PSR) is a 16-bit hardware register that contains information about the state of the ECLIPSE MV/8000. The format of the PSR register is

SD-02281



The first two bits (bits 0 and 1) of the PSR contain overflow information. The OVK bit (bit 0) is the overflow mask and the OVR bit (bit 1) is the overflow indicator. Bits 2 through 15 are reserved for use by the hardware. See the *ECLIPSE MV/8000 Principles of Operations* manual for more information on the PSR.

To open the processor status register (PSR), enter the following DEBUG command:

```
$V
```

where:

$	represents the escape (ESC) character
V	directs DEBUG to open the PSR

When DEBUG displays the contents of the PSR, it always shows bits 2 through 15 as zeros since these bits are reserved for hardware use. DEBUG's display does, however, reflect the current values for bits 0 and 1 (i.e., OVK and OVR).

The following example shows a $V command and the Debugger's response:

```
_$V 00000140000 _
```

This display shows that the OVK and OVR bits are both set to 1. Note that DEBUG presents the PSR as a 32-bit value even though it is only one word (16 bits) in length. Ignore the first word of DEBUG's display.

After DEBUG displays the PSR, that register is open for modification. DEBUG only allows you to modify bits 0 and 1 (OVK and OVR). Bits 2 through 15 are reserved for hardware use and if you try to modify them, DEBUG returns an error. After you enter a new value, close the PSR register with a NEW LINE or carriage return.

The following examples show how to modify the PSR register:

_$V *00000000000* _140000↵ _	Open the PSR, set the OVK and OVR bits to 1, and close with NEW LINE
_$V *00000140000* _100000↵ _	Open the PSR, set the OVK bit to 1 and the OVR bit to 0
_$V *00000100000* _40000↵ _	Open the PSR, set the OVK bit to 0 and the OVR bit to 1
_$V *00000040000* _377↵ *?*_	ERROR: The Debugger does not allow you to modify bits 2 through 15 of the PSR register

Floating Point Accumulators and FPSR

DEBUG allows you to examine the contents of the four floating point accumulators (FPACs) and the Floating Point Status Register (FPSR) by entering the following command:

$F

where:

$	represents the escape (ESC) character
F	directs DEBUG to display the floating point accumulators and the FPSR

The floating point registers are each four-word (64-bit) values. When displaying them as a group, DEBUG presents the four floating point accumulators (FPACs) as double-precision (64-bit) floating point numbers, regardless of the current display mode. DEBUG presents the FPSR as two 32-bit integers; the first one holds the floating point status information, the second holds the floating point program counter (PC). Refer to the *ECLIPSE MV/8000 Principles of Operation* manual for more information on the floating point registers.

An example $F command and the Debugger response is

```
_$F
0.0000000000000000E+0 0.0000000000000000E+0
0.0000000000000000E+0 0.0000000000000000E+0
0000160000 00000000000
_
```

The first line of DEBUG's response shows the values of FPAC0 and FPAC1; the second line shows FPAC2 and FPAC3. The last line displays the FPSR as two 32-bit integers.

The $F command does *not* open the registers for modification, but simply displays their contents.

To view a specific floating point accumulator or the FPSR, enter

n$F

where:

n	is an integer value from 0 to 5, inclusive. If n=4, DEBUG opens the status portion of the FPSR (the first 32-bits); if n=5, DEBUG displays the floating point program counter portion of the FPSR (the last 32 bits); otherwise, DEBUG opens the accumulator specified by n (i.e., floating point accumulator FPACn)
$	represents the escape (ESC) character
F	informs DEBUG that you wish to access a floating point register

As an example, to open floating point accumulator 3 (FPAC3), enter

```
_3$F
```

The Debugger displays the four-word (64-bit) contents of FPAC3 on the same line:

```
_3$F 000000 00000000000 000000 _
```

When displaying the contents of a single register, DEBUG uses the current display mode (numeric, by default). To view the contents of a floating point register in floating point notation, you must set the display mode accordingly (see Chapter 8). If you are currently in floating point display mode, DEBUG responds to your register examine command as follows:

```
_3$F 0.0000000000000000E+0 _
```

After DEBUG displays the contents of a floating point accumulator, that resister is open for modification. Enter a new value, if you wish, and close the register. For example,

_3$F *0.0000000000000000E+0* _ 7.125E-6↵ _	Open FPAC3, deposit the value 7.125E-6, and close with NEW LINE

FPAC3 now contains the value 7.125E-6, as shown by the following command:

```
_3$F 7.1250000000000000E-6 _
```

To open the status portion (bits 0 through 31) of the Floating Point Status Register (FPSR), enter the command

```
_4$F
```

DEBUG presents the first 32 bits of the FPSR in the current display mode, followed by a prompt. At this point, you may modify the status value, if you wish.

_4$F *00001600000* _41600000↵ _	Open the status portion of the FPSR. Enter the value 41600000 and close with NEW LINE

To view the floating point PC (i.e., bits 32 through 61 of the FPSR), enter

```
_5$F
```

DEBUG displays the 32-bit floating point PC in the current display mode. For example (in numeric mode),

```
_5$F 000000000000 _
```

Though DEBUG presents the floating point PC value, that register is *not* open for modification. You may *not* enter a new floating point PC value. If you try to do so, DEBUG returns an error.

Stack Registers

Using the $E command, you may display the four ECLIPSE MV/8000 stack registers: stack pointer, frame pointer, stack limit, and stack base. The command syntax is

$E

where:

$	represents the escape (ESC) character
E	directs DEBUG to display the four stack registers

When displaying them as a group, DEBUG always presents the four stack registers in numeric mode, regardless of the current display mode.

An example $E command and the Debugger's response is

```
_$E        (SP,FP,SL,SB)
16000001004 00000000000 16000001077 16000001004
_
```

The $E command does *not* open registers for modification, but simply displays their contents.

To open a specific stack register, enter

n$E

where:

n is an integer value from 0 to 3, inclusive. n identifies one of the four stack registers as follows:

Value of n	Register
0	Stack Pointer (SP)
1	Frame Pointer (FP)
2	Stack Limit (SL)
3	Stack Base (SB)

$	represents the escape (ESC) character
E	informs DEBUG that you wish to open a stack register

When presenting a single register, DEBUG always uses the current display mode.

The following command opens the stack pointer:

```
_0$E
```

The Debugger will display the stack pointer and open that register for modification. Enter a new value, if you wish, and close the register. For example,

```
_0$E 16000001004 _16000002117↵      Open the stack pointer, enter the value 16000002117, and close
_                                   with NEW LINE
```

Program Counter

The program counter (PC) holds the address of the next instruction your program will execute. To open the program counter, enter

$L

where:

$	represents the escape (ESC) character
L	directs DEBUG to open the PC register

The following example shows a $L command and the Debugger's response:

```
_$L 16000000446 _
```

After DEBUG displays the program counter, that register is open for modification. Enter a new value, if you wish, and close the register.

_$L *16000000446* _500;7↲ _	Open the PC register, enter the value 500;7 (location 500 in ring 7), and close with NEW LINE

After this command, the PC register identifies address 500_8 in ring 7. Your program will now begin (or continue) execution at that location.

Since the PC register holds an address, the value you enter must include the ring value for your program in bits 1 through 3. If the PC value does *not* address the appropriate ring, DEBUG returns an error when you try to continue program execution. See "Rings" in Chapter 3 for more information.

You may wish to close the PC register using the memory examine commands / and \ . As described earlier, these characters direct DEBUG to close the PC and open the location specified by the PC value. For example,

_$L *16000000446* _ / 103000↲ _	Open the PC register. Since we close with /, DEBUG uses the PC value as an address and opens that location (i.e., location 16000000446 contains the value 103000)

Chapter 3 provides more information on memory access and the memory examine commands / and \ . That chapter also contrasts the program counter and the location counter.

Debugger Registers

The Debugger uses various internal registers when performing certain operations. You cannot access these registers from your program, but you may examine and modify them directly during the debugging session. The *Debugger registers* include

- output radix register
- global display mode register
- proceed count registers
- ring register

The following sections discuss each of these registers. We describe some in more detail elsewhere; in these cases, we provide brief descriptions in this chapter and references for more information.

Radix Register

The value in the *radix register* determines the radix (base) that DEBUG will use when presenting numeric values. By default, DEBUG presents numeric values in octal (base 8), but you may set the output radix to any base from 2 (binary) to 16 $_{10}$ (hexadecimal), inclusive.

To examine the radix register, enter

$N

where:

$	represents the escape (ESC) character
N	directs DEBUG to display the radix register in the current display mode. DEBUG always presents this value as a decimal (base 10) number

An example command and response is

```
_$N 8. _          8. equals 8₁₀
```

After DEBUG displays the current output radix, the register is open for modification. You may insert any value from 2 to 16 $_{10}$; DEBUG interprets the new value in octal unless you include a a decimal point (i.e., 16. equals 16 $_{10}$).

The following example, shows the use of the $N command:

```
_$N 8. _↲
_0$A 00000000077 _↲
_$N 8. _10.↲
_0$A 0000000063 _↲
_$N 10. _16.↲
_0$A 0000003F _↲
_$N 16. _
```

Examine the radix register (default output radix is octal) and close with NEW LINE. DEBUG displays the contents of AC0 in octal. Again, open the radix register. This time, deposit the value 10_{10}. Now, DEBUG presents the contents of AC0 in decimal. Lastly, change the output radix to hexadecimal (16 $_{10}$) and display AC0. At the end of this example, the output radix is hexadecimal

Note that the number of digits DEBUG displays varies with the radix value. For example, when the radix is octal, DEBUG requires 11 $_{10}$ digits to present a 32-bit integer. However, DEBUG presents decimal and hexadecimal 32-bit values using only 10 $_{10}$ and 8 $_{10}$ digits, respectively.

```
00000000077
0000000063
0000003F
```

The 32-bit value 63_{10} in octal, decimal, and hexadecimal. Note the number of columns DEBUG requires in each case

Global Display Mode Register

The Debugger may present data in a variety of modes (e.g., numeric, instruction, symbolic, floating point, ASCII, half-word, etc.). The *global display mode register* holds a value that indicates which mode DEBUG is currently using.

Table 4-2 lists the various display mode register values and the corresponding DEBUG display modes.

Table 4-2. Display Mode Values

Register Value		Display Mode
Decimal	Octal	
1	1	Numeric (default mode)
2	2	Numeric words
3	3	Instruction
4	4	Symbolic
5	5	Half-word
6	6	Byte pointer
7	7	ASCII
8	10	Floating point
9	11	System call
0	0	AOS/VS error message

Chapter 8 provides more information on each of these display modes.

To examine the display mode register, enter the command

$T

where:

$	represents the escape (ESC) character
T	directs DEBUG to present the global display mode register

An example of this command and the Debugger's response is

```
_$T 00000000001 _
```

After DEBUG presents the display mode register, that register is open for modification. Enter a new value from 0 to 9 $_{10}$ (0 to 11$_8$), if you wish, and close the register. For example,

```
_$T 00000000001 _5↲
_
```

Open the display mode register (current display mode is numeric). Change the global display mode to half-word (i.e., enter 5) and close the register with NEW LINE

The following example is more extensive and shows how the display mode affects Debugger output.

```
_$T 00000000001 _↲
_0$A 11420245505 _↲
_$T 00000000001 _5↲
_0$A 114 101 113 105 _↲
_$T 000 000 000 005 _7↲
_0$A LAKE _↲
_
```

The display mode register equals 1, so DEBUG displays AC0 in numeric mode. Change the display mode to half-word (5) and view AC0. Lastly, change the display mode to ASCII (7) and again view AC0

The $T command is only one of several ways you may modify the display mode. Refer to Chapter 8 for a complete description of the various display modes and the commands that invoke those modes.

Proceed Count Registers

A *proceed count* determines how many times DEBUG must encounter a breakpoint before it stops your program. Proceed counts are particularly useful when you set a breakpoint inside a loop or routine and want DEBUG to stop execution the n th time it encounters that breakpoint.

Each breakpoint has a proceed count associated with it. This value resides in that breakpoint's *proceed count register*. The default value for each proceed count register is 1; i.e., stop program execution each time DEBUG encounters the breakpoint.

To examine a proceed count register, enter the following command:

```
n$Q
```

where:

n	is an integer value that identifies a breakpoint
$	represents the escape (ESC) character
Q	directs DEBUG to open the proceed count register for breakpoint n

The following example shows a typical n$Q command and the Debugger's response:

_2$Q *00000000001* _	Open the proceed count register for breakpoint number 2

After DEBUG displays a proceed count register, that register is open for modification. Enter a new value, if you wish, and close the register. For example,

_2$Q *00000000001* _5↵ _	Open the proceed count register for breakpoint 2, deposit the value 5, and close with NEW LINE

After this command, the proceed register for breakpoint 2 equals 5. That is, DEBUG will stop program execution the fifth time it encounters breakpoint number 2.

If you enter an n$Q command and there is no breakpoint n currently defined, DEBUG returns an error.

This discussion is meant as a very brief overview of proceed count registers. "Proceed Count" in Chapter 5 provides much greater detail on the subject and includes a number of examples.

Ring Register

All AOS/VS addresses contain a ring value between 0 and 7, inclusive. This value resides in bits 1 through 3 of the address. Thus, your addresses will generally be of the form

```
16000000100
```

This number represents location 100_8 in ring 7. (Note that the value in bits 1 through 3 of the address equals 7.)

Usually, all addresses you wish to access reside in the same ring (often, ring 7). Rather than explicitly specifying the ring field in each address, you may select a default value and let DEBUG insert it for you. This default ring value resides in the *ring register*. It initially equals the ring value in the program counter.

Thus, if the ring register holds the value 7, DEBUG assumes any address that does not explicitly include a ring value refers to ring 7. For example,

```
_100/
```

Since the address 100 does not specify a ring, DEBUG inserts the value from the ring register (i.e., 7) into bits 1 through 3. In this case, DEBUG converts the entry 100 into address 16000000100.

To examine the ring register, enter the command

```
$G
```

where:

$	represents the escape (ESC) character
G	directs DEBUG to present the ring register

The following example, shows a $G command and DEBUG's response.

```
_$G 00000000007 _          Open the ring register
```

After DEBUG displays the ring register, that register is open for modification. Enter a new value, if you wish, and close the register. For example,

```
_$G 00000000007 _6↵        Open the ring register, enter the value 6, and close with NEW LINE
_
```

After this command, ring 6 is the default addressing ring.

Again, DEBUG only refers to the ring register when an address does not include a ring value. If you specify an address that includes a ring value, DEBUG uses that value and not the ring register.

This discussion is meant as a brief overview. Refer to "Rings" in Chapter 3 for more information on rings and $G.

If you are using the FED utility (not DEBUG), you may deposit a -1 value in the ring register. This allows you to examine locations in your program's preamble. Refer to Chapter 10 for more information on FED and the -1 ring register value.

End of Chapter

Chapter 5
Breakpoints and Program Restarts

A *breakpoint* is a position in your program where you wish to suspend execution. When your program encounters a breakpoint, it immediately passes control to the Debugger. You may then enter commands to examine and modify memory locations, machine state registers, and Debugger registers (see Chapters 3 and 4). After performing the desired actions, you may continue program execution at the breakpoint or at any other location in your program.

In this chapter, we explain how to set, examine, and delete breakpoints, and how to continue program execution after a breakpoint. Note that you may *not* set breakpoints in your program if you enter the Debugger via a ?DEBUG system call. Refer to "Entering the Debugger from a Program" in Chapter 2 for more information.

Table 5-1 summarizes the various breakpoint-related commands. Figure 5-1, under "Proceed Counts", is a flowchart of the Debugger's breakpoint logic.

Table 5-1. Breakpoint Command Summary

Command	Action
$B	Display all breakpoints
address$B	Set a breakpoint at location **address**
address,condition$B	Set a conditional breakpoint at location **address**
$D	Delete all breakpoints
n$D	Delete breakpoint n
$P	Continue program execution at the last breakpoint encountered
n$P	Continue program execution at the last breakpoint encountered and set the proceed count for that breakpoint to n
n$Q	Open the proceed count register for breakpoint n
$R	Resume program execution at the current program counter (PC)
address$R	Resume program execution at location **address**

Setting Breakpoints

DEBUG allows you to have 20_{10} breakpoints in your program at a time. The command for setting a breakpoint is

address$B

where:

address	is a number, symbol, or expression whose value specifies a word location where you want to set a breakpoint; you will frequently pass a label as an address argument to $B
$	represents the escape (ESC) character
B	informs DEBUG that you want to set a breakpoint

The **address$B** command directs DEBUG to set a breakpoint in your program at location **address**. During program execution, DEBUG will stop your program *before* the instruction at location **address** is executed. You can then examine and modify memory locations and registers.

Be sure you set your breakpoints at the beginning of executable instructions. Do *not* set breakpoints at the following locations:

- data entries
- instructions modified during program execution
- within multiword instructions

Also, if the **address** value does not specify a memory ring, DEBUG inserts one from the ring register (see "Rings" in Chapter 3).

An example command for setting a breakpoint is

```
_1500$B _
```

DEBUG sets a breakpoint at word address 1500_8 and returns a prompt. During execution, your program will stop and pass control to DEBUG each time it encounters the breakpoint at address 1500_8.

Often, you will supply a label in the **address** argument to $B. For example,

```
_LOC$B _)
_LOC+40$B _
```

DEBUG sets two breakpoints; one at address LOC and one at address LOC+40 (assuming LOC is a valid symbol). Chapter 6 describes symbol use in more detail.

In the above discussion, we present the simplest way to set breakpoints. Later we discuss two features that make breakpoints more powerful: conditions and proceed counts.

Displaying Breakpoints

As mentioned earlier, DEBUG allows your program to contain up to 20_{10} (24_8) breakpoints at a time. To display a list of breakpoints, their locations, and their proceed counts, enter

$B

where:

$	represents the escape (ESC) character
B	directs DEBUG to display all breakpoints, their locations, and their proceed counts

For example, suppose you set breakpoints at address 446_8, 500_8, and 540_8 as follows:

```
_446$B ↵   Set breakpoints at addresses 446, 500, and 540
_500$B ↵
_540$B _
```

If you now enter the display command, DEBUG will show you all currently set breakpoints.

```
_$B
B0  16000000446  #1
B1  16000000500  #1
B2  16000000540  #1
_
```

The first entry on each line shows the breakpoint number. B0 is breakpoint 0, B1 is breakpoint 1, etc. Since DEBUG displays the numbers in octal, it numbers breakpoints from 0 to 23_8, and B10 refers to breakpoint 10_8.

The second entry shows the address of the breakpoint. In the above example, B0 is at location 16000000446_8. (Note that DEBUG automatically inserted ring values in the addresses; see "Rings" in Chapter 3.)

If DEBUG has access to a symbol table, it will try to present the breakpoint addresses in symbolic mode. For example, suppose symbol START equals 16000000446_8 and symbol LOC equals 16000000500_8. DEBUG would present the above three breakpoints as

```
_$B
B0 START #1
B1 LOC #1
B2 LOC+40 #1 _
_
```

Refer to Chapter 6 for more information on symbols and symbol tables.

The last entry on each line indicates the proceed count for that breakpoint. In the above example, all breakpoints have a proceed count of 1 (i.e., # 1). This means that DEBUG will suspend program execution every time it encounters one of those breakpoints. "Proceed Count" later in this chapter gives more information on this subject.

Deleting Breakpoints

To delete all breakpoints from your program, enter the command

```
$D
```

where:

$	represents the escape (ESC) character
D	informs DEBUG that you wish to delete breakpoints

To delete a specific breakpoint from your program, enter

```
n$D
```

where:

n	is the number of a previously set breakpoint (as shown in a $B display)
$	represents the escape (ESC) character
D	informs DEBUG that you wish to delete a breakpoint

For example, suppose you set breakpoints at locations 446_8, 500_8, and 540_8. A $B display would show

```
_$B
B0  16000000446  #1
B1  16000000500  #1
B2  16000000540  #1
_
```

If you want to delete breakpoint B1 from your program, enter the command

```
_1$D
```

To verify the deletion, enter another $B command.

```
_$B
B0  16000000446  #1
B2  16000000540  #1
_
```

This display shows that DEBUG deleted breakpoint B1. Note that all other breakpoints retain their same identification numbers (e.g., the breakpoint at location 16000000540_8 is still breakpoint 2).

If you do not delete breakpoints from your program before leaving the debugging session, DEBUG will automatically remove them for you.

Conditional Breakpoints

When you set a breakpoint, you have the option of associating a *condition* with that breakpoint. Each time DEBUG encounters the breakpoint, it will evaluate the condition. If the condition is false (equal to 0), DEBUG will *not* stop program execution. If, on the other hand, the condition is true (not equal to 0), DEBUG will stop your program at that breakpoint.

The command for setting a *conditional breakpoint* is

address,condition$B

where:

address	is a number, symbol, or expression whose value specifies a word location where you want to set a conditional breakpoint
condition	is an expression that DEBUG evaluates when it encounters the breakpoint. If the expression is true (does not equal 0), DEBUG stops your program at the breakpoint; otherwise, program execution continues
$	represents the escape (ESC) character
B	informs DEBUG that you wish to set a breakpoint

In most cases, your condition argument will contain a relational operator -- >, <, =, >=, <=, or <>.

Consider the following example:

```
_446,@1500>=5$B
```

This command directs DEBUG to set a conditional breakpoint at address 446_8. When your program encounters the breakpoint, DEBUG will evaluate the conditional expression @ 1500 > =5. If it is true (not equal to 0), DEBUG will stop program execution at the breakpoint.

@ 1500 equals the one-word value at location 1500_8. Thus, the expression @1500 > =5 is true if the contents of word 1500_8 is greater than or equal to 5. See "Operators" in Chapter 7 for more information on the @ operator.

Our second example sets a conditional breakpoint that compares an accumulator value to a constant (the symbol < $ > represents the dollar sign character, not escape):

```
_523,2<$>A=177$B
```

This command sets a conditional breakpoint at location 523_8. The expression 2 < $ > A represents the value of AC2 (see Chapter 6 for information on including register values in expressions). Thus, when your program encounters this breakpoint, DEBUG compares the value of AC2 to the constant 177_8. If AC2 equals 177, the condition is true and DEBUG stops program execution. Otherwise, DEBUG ignores the breakpoint and program execution continues.

Proceed Counts

Each breakpoint has a *proceed count* associated with it. The proceed count is a value that indicates how many times your program must encounter the breakpoint before DEBUG gets control.

By default, DEBUG assigns each breakpoint a proceed count of 1. Each time your program encounters a breakpoint, it decrements that breakpoint's proceed count by 1 and checks the result. If the proceed count equals 0, your program stops, control passes to the Debugger, and you may enter commands.

If the result is *not* 0, program execution continues. After your program stops execution at a breakpoint, DEBUG sets that breakpoint's proceed count to 1, the default value.

Proceed counts are particularly useful when you set a breakpoint in a loop or routine but only want your program to stop execution when it encounters that breakpoint for the n^{th} time.

Figure 5-1 shows DEBUG's logic for stopping program execution at a breakpoint. Note that the flowchart includes logic for the optional breakpoint condition (see "Conditional Breakpoints" earlier in this chapter).

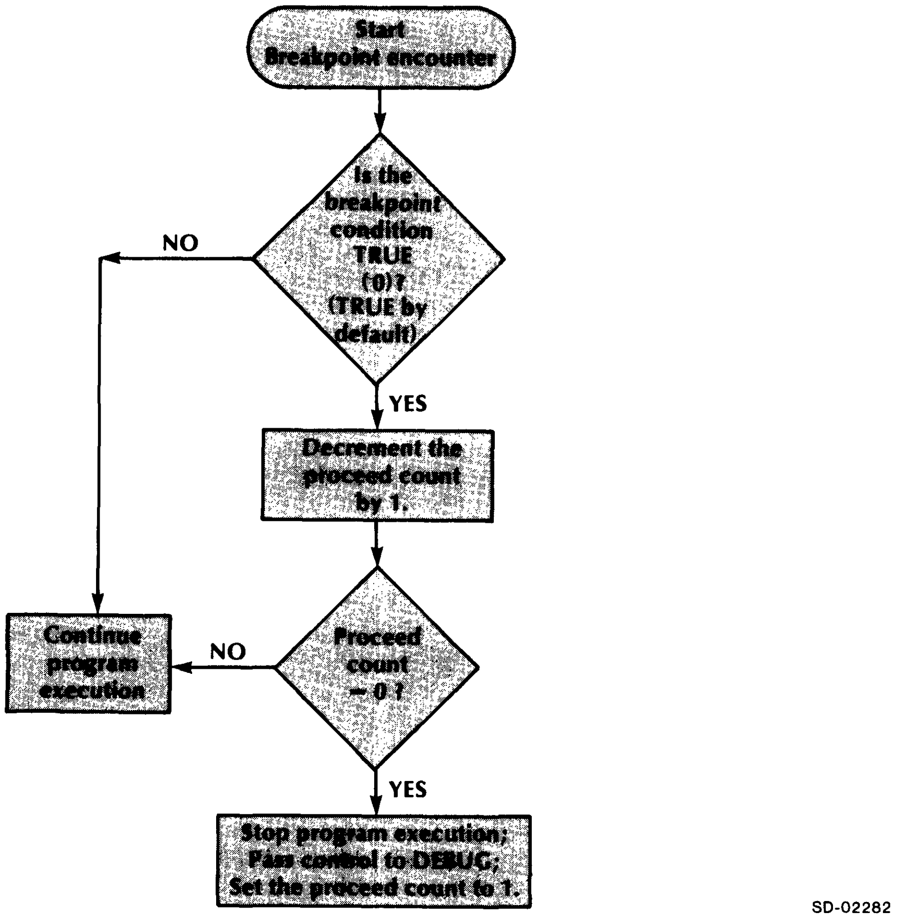

Start Breakpoint encounter
Is the breakpoint condition TRUE (0)? (TRUE by default)
NO
YES
Decrement the proceed count by 1.
Continue program execution
NO
Proceed count = 0 ?
YES
Stop program execution; Pass control to DEBUG; Set the proceed count to 1.
SD-02282

Figure 5-1. Breakpoint Logic Diagram

To determine the proceed count values for your breakpoints, enter the breakpoint display command $B. For example,

```
_$B
B0  16000000446  #1
B1  16000000500  #1
B2  16000000540  #1
_
```

The last entry on each line (e.g., # 1) indicates the proceed count for the breakpoint. In this example, B0, B1, and B2 all have proceed counts of 1.

DEBUG provides two ways to modify the proceed count for a breakpoint:

- open the proceed count register and enter a new value (via the n$Q command)
- supply a proceed count when you continue program execution at a breakpoint (via the n$P command)

The proceed count value for each breakpoint resides in a *proceed count register.* To open this register, enter the command

```
n$Q
```

where:

n	is the number of a previously set breakpoint (as shown in a $B display)
$	represents the escape (ESC) character
Q	directs DEBUG to open the proceed count register for breakpoint number n

In the previous example, breakpoint B1 has a proceed count of 1 (# 1). To open B1's proceed count register, enter

```
_1$Q
```

DEBUG responds with the proceed count for breakpoint 1. At this point that register is open and you may enter a new proceed count value, if you wish. For example,

`_1$Q 00000000001 _4↵`
`_`
	Open the proceed count register for B1 by entering 1$Q. After DEBUG displays the proceed count (1), enter the new value 4 and close the register with NEW LINE

The proceed count for B1 now equals 4. That is, your program will ignore B1 the next three times it is encountered. However, on the fourth encounter, program execution stops and control passes to DEBUG. After the program stops at B1, DEBUG resets the corresponding proceed count register to 1.

Refer to Chapter 4 for more information on the proceed count register and registers in general.

The second way to alter a proceed count is to wait until your program encounters the breakpoint. Then, restart your program with the command

```
n$P
```

DEBUG will set the proceed count for the breakpoint just encountered to n and will continue program execution at that breakpoint.

Refer to "Restarting Your Program at a Breakpoint" later in this chapter for more information on the $P command.

Program Restarts

When your program stops at a breakpoint, DEBUG presents a display similar to the following:

```
B0 16000036000
00000000000 00000000000 00000000000 00000000000 00000000000
_
```

The first line identifies the breakpoint (e.g., B0 is breakpoint number 0) and its address. The following five numbers show the values of AC0, AC1, AC2, AC3, and the carry bit, from left to right.

After you issue various DEBUG commands, you can either continue program execution or end your debugging session. The Debugger supplies two commands for restarting program execution after a breakpoint. You may

- restart your program at the breakpoint (via $P) or
- restart your program at a specified location (via $R)

If you want to end your debugging session, enter the $Z command (see "Leaving the Debugger" in Chapter 2 for more information).

If your program traps during execution, control returns to the Debugger. At this point, you can examine memory and registers but may *not* continue program execution. If you try to do so, DEBUG will terminate the debugging session. Refer to the *AOS/VS Programmer's Manual* for information on traps.

Restarting Your Program at a Breakpoint

To restart program execution at the last breakpoint encountered, issue the command

```
$P
```

where:

$	represents the escape (ESC) character
P	directs DEBUG to proceed with program execution at the last breakpoint encountered

Remember, when your program encounters a breakpoint, it stops *before* executing the instruction originally at the breakpoint address. Thus, when you issue the $P command, your program will restart at that address.

When you restart your program at a breakpoint via $P, you may also specify a proceed count for that breakpoint (see "Proceed Counts" earlier in this chapter). The syntax for this command is

```
n$P
```

where:

n	is the new proceed count for the last breakpoint encountered
$	represents the escape (ESC) character
P	directs DEBUG to continue program execution at the last breakpoint encountered

As an example, suppose your program runs through a loop 100_{10} times but you only want to DEBUG your code during the last iteration of that loop. One way to do this is to set a breakpoint at the beginning of the loop and enter the $P command each time your program stops. However, you have to issue the $P command 99_{10} times before your program finally enters the last (100th) iteration of the loop.

Alternatively, after your program stops at the breakpoint for the first time, you could enter the command

```
_99.$P
```

This command sets the proceed count for the breakpoint to 99_{10} and restarts your program at that breakpoint. Now program execution will not stop at that breakpoint until the 99th iteration of the loop. Again, see "Proceed Counts" for more information on their use.

Note that you may only issue a $P command after your program encounters a breakpoint. If you try to start your program with $P, DEBUG returns an error.

Restarting Your Program at a Location

To restart your program at the address in the program counter (PC), enter the command

```
$R
```

where:

$	represents the escape (ESC) character
R	directs DEBUG to resume program execution at the PC value

When program execution stops at a breakpoint, the program counter (PC) holds the address of that breakpoint. Thus, the $R command directs DEBUG to resume execution at the breakpoint.

Note that after a breakpoint, the $P and $R commands perform the same operation (i.e., continue program execution at the breakpoint).

If you want to continue program execution at a location other than the current PC value, you can either

- modify the address in the program counter (PC) via the $L command (see Chapter 4) and then enter $R to continue execution at that address, or
- pass an address directly in the $R command as follows:

```
address$R
```

where:

address	is a number, symbol, or expression that indicates the address in your program where you want to continue execution
$	represents the escape (ESC) character
R	directs DEBUG to resume program execution at location **address**

If you enter an **address$R** command, DEBUG will continue program execution at location **address**.

As an example, suppose your program just encountered a breakpoint. If you want to continue execution at location 1025_8 rather than at the breakpoint, enter

```
_1025$R
```

DEBUG would restart your program at location 1025_8.

Of course, you may pass symbols to $R. For example, to continue execution at label LOOP1, enter

```
_LOOP1$R
```

In the previous section, we said you could only issue the $P command after your program encounters a breakpoint. This is *not* true for the $R command; in fact, you must use this command to initiate program execution when your first enter the debugging session (see "Starting Program Execution" in Chapter 2).

End of Chapter

Chapter 6
Symbol Recognition and Definition

The AOS/VS Debugger allows you to enter symbolic values and can present data in symbolic mode. DEBUG can recognize three symbol types:

- symbols defined in your program's symbol table (i.e., in the .ST file)
- temporary symbols you define during the debugging session
- special Debugger symbols that represent the contents of the location counter and the various MV/8000 machine state registers

In the following sections, we first describe these three types of symbols and then show how to use them during the debugging session.

The section entitled "Checking a Symbol's Value" explains how to obtain any symbol's value.

Table 6-1 lists the DEBUG commands that we describe in this chapter.

Table 6-1. Symbol Command Summary

Command	Action
$I	Display the currently defined temporary symbols
symbol,value$I	Define a temporary symbol
$J	Delete all temporary symbols
symbol$J	Delete a specific temporary symbol
$X	Disable the current symbol table file
file$X	Disable the current symbol table file (if any) and enable a new one

Program Symbol Recognition

When you enter a debugging session, DEBUG automatically searches for your program's symbol table file. Normally, your program's symbol table file has the same name as your program file, less the .PR extension and with the new extension .ST.

Thus, if you invoke the Debugger from the CLI with the command

```
) DEBUG     PROG1⤶
```

the Debugger will search for file PROG1.ST and will use it as the symbol table, if available.

DEBUG recognizes your program's global symbols; that is, the symbol's you declare in .ENT pseudo-op source statements. DEBUG does *not* recognize any other symbols you define in your program. Refer to the *AOS/VS Macroassembler Reference Manual* for a description of the .ENT pseudo-op.

In addition to .ENT symbols, your program's symbol table includes all the AOS/VS memory parameter symbols (e.g., ?ZBOT, ?NMAX, ?CLOC) and certain symbols from the system library URT32.LB. The *AOS/VS Link and LFE User's Manual* contains more information on these symbols.

Disabling a Symbol Table

If you want to stop using the current symbol table, enter the command

```
$X
```

where:

$	represents the escape (ESC) character
X	directs DEBUG to close the symbol table

After you issue a $X command, DEBUG will not recognize any symbols in your program. Consider the following example,

```
_START/ 006017 ↵
_$X ↵
_START/ U? _
```

By default, DEBUG recognizes symbol START (your program declares START in an .ENT statement). After you disable the symbol table with $X, DEBUG does not recognize START

If you enter the $X command and no symbol table is currently enabled, DEBUG returns an error message.

There is no way to disable a specific symbol in your .ST symbol table. You must either enable or disable the whole set.

Enabling a Symbol Table

To enable a symbol table, enter the command

```
file$X
```

where:

file	is the complete pathname of a file that contains a symbol table; include the .ST extension when specifying the filename
$	represents the escape (ESC) character
X	informs DEBUG that you want to open a symbol table

When DEBUG encounters a file$X command, it first closes the current symbol table, if one is open. Then DEBUG opens the specified file as the new symbol table.

Consider the following example:

```
_$X ↵
_LOC1/ U? ↵
_PROG1.ST$X ↵
_LOC1/ 000077 _
```

$X disables the current symbol table; DEBUG does not recognize symbol LOC1. Open file PROG1.ST as the symbol table. DEBUG now recognizes symbol LOC1, defined in PROG1.ST

Note that we explicitly included the .ST extension when specifying the symbol table file PROG1.ST; the Debugger does *not* add this extension for you. Should you omit it, the Debugger will send an error message.

Temporary Symbols

DEBUG allows you to define up to 20_{10} symbols during the debugging session. We refer to these as *temporary symbols* since they are known only for the duration of the current debugging session.

Temporary symbol definitions reside in the *temporary symbol table.* When displaying or interpreting symbols, DEBUG refers to the temporary symbol table first; the .ST symbol table second. Thus, if you define a temporary symbol that has the same name as one in the .ST symbol table, DEBUG uses the value in the temporary symbol table.

Defining Temporary Symbols

The syntax for defining a temporary symbol is

```
symbol,value$I
```

where:

symbol	is the name of the temporary symbol you want to define; the following discussion states the rules for temporary symbol names
value	is a number, symbol, or expression that specifies a value for symbol
$	represents the escape (ESC) character
I	informs DEBUG that you are defining a temporary symbol

Each temporary symbol (i.e., symbol, in the above command syntax) may contain up to 32_{10} characters. Symbol names must conform to the following format:

a *[b...]*

where:

a	is the first character of the symbol and may be any upper- or lowercase letter, period (.), question mark (?), or dollar sign ($)
b	represents subsequent characters in the symbol and can include upper- and lowercase letters, numbers (0 through 9), period (.), question mark (?), dollar sign ($), and underscore (_)

According to this definition, the following are all legal temporary symbols:

```
LOCATION1     .START     B19_X?     ?IOD
```

The Debugger is case sensitive. Thus, the symbol 'start' is *not* the same as 'START'.

The following example defines the three temporary symbols COL, LIB, and TJ:

```
_COL,25$I_↵     Define temporary symbols COL, LIB, and TJ
_LIB,144$I_↵
_TJ,47$I_
```

DEBUG now recognizes these three symbols. You may use them anywhere you would use the corresponding numeric values. In addition, DEBUG will use them for display purposes. Later in this chapter, we discuss how you and DEBUG may use symbolic references.

Displaying Temporary Symbols

To display a list of the currently defined temporary symbols and their values, enter the command

$I

where:

$	represents the escape (ESC) character
I	directs DEBUG to display all temporary symbols and their values

Consider the following example:

```
_COL,25$I⌋          Define temporary symbols COL, LIB, and TJ
_LIB, 144$I⌋
_TJ,47$I⌋
_$I                 Display all currently defined temporary symbols
COL   00000000025
LIB   00000000144
TJ    00000000047
_
```

Deleting Temporary Symbols

To delete all temporary symbols and their definitions, enter

$J

where:

$	represents the escape (ESC) character
J	directs DEBUG to delete all temporary symbol definitions from the current debugging session

The $J command deletes all temporary symbol definitions but does not affect your program's symbol table (i.e., the .ST file).

If you wish to delete a specific temporary symbol, enter the command

symbol$J

where:

symbol	is the name of the temporary symbol you want to delete
$	represents the escape (ESC) character
J	directs DEBUG to delete symbol from the temporary symbol table

As an example, suppose symbols COL, LIB, and TJ have values 25, 144, and 47, respectively. A $I display would show

```
_$I
COL   00000000025
LIB   00000000144
TJ    00000000047
_
```

If you want to delete temporary symbol LIB, enter

```
_LIB$J
```

To verify the deletion, enter another $I display command.

```
_$I
COL    00000000025
TJ     00000000047
_
```

This display shows that DEBUG deleted symbol LIB from the temporary symbol table.

Special Debugger Symbols

The Debugger provides special symbols that you may use to represent the contents of the location counter and the various MV/8000 machine state registers. You can use these special symbols any place you can use temporary or program file (.ST) symbols.

Table 6-2 lists the special Debugger symbols (< $ > represents the dollar sign character, not the escape).

Table 6-2. Special Debugger Symbols

Symbol	Value
. (period)	The current value of the location counter
n<$>A	The contents of fixed point accumulator n (i.e., ACn) for 0<=n<=3; the contents of the carry bit for n=4
n<$>E	The current value of the MV/8000 stack pointer (n=0), frame pointer (n=1), stack limit (n=2), or stack base (n=3) register
n<$>F	The high-order (most significant) 32 bits of floating point accumulator n (i.e., FPACn) for 0<=n<=3; the first 32 bits of the FPSR for n=4; the last 32 bits of the FPSR (the floating point PC) for n=5
<$>L	The current value of the program counter (PC)
<$>V	The current value of the processor status register (PSR)

The *location counter* holds the address of the location that DEBUG most recently displayed. Refer to "Location Counter and Program Counter" in Chapter 3 for more information.

To use the value of the location counter during your debugging session, enter the single character symbol . (period). For example,

_450/ *000010* _⏎ _1$A *00000000000* _⏎ _	Examine location 450_8; the location counter now holds that address. Open AC1 and deposit the value of the location counter (i.e., deposit address value 450 into AC1
_./ *000010* _	Examine the contents of the location counter; note that ./ is equivalent to 450/ at this point in the debugging session

You will often combine the location counter symbol with the indirect operators @ and # . For example, the expression @ . equals the 16-bit value starting at the address in the location counter. See "Indirect Operators" in Chapter 7 for more information on @ and # .

In addition to the location counter, the Debugger provides special symbols that represent all of the MV/8000 machine state registers (see Table 6-2). The symbol format for entering a register value is the same as the corresponding register examine command, except you use the dollar sign instead of the escape character. In this manual, we use $ to represent the escape character (ASCII 33_8) and <$> to represent the dollar sign (ASCII 44_8).

The following examples show the use of the special register symbols:

_1$A *00000000001* _⏎ _460/ *777777* _1<$>A⏎ _460/ *000001* _⏎	Examine the value of AC1. Open location 460_8, and deposit the value of AC1; then verify the new value
_1$E *16000000730* _1<$>E+6⏎ _1$E *16000000736* _	Open the frame pointer and increase its value by 6; then check the new value
_0$A *00000000000* _3<$>A⏎ _	Open AC0 and deposit the value of AC3
_2$A *00000000000* _3<$>A+1<$>A⏎ _	Open AC2 and deposit the sum of AC3 and AC1

Refer to Chapter 4 for more information on examining and modifying the MV/8000 machine state registers during your debugging session.

Checking a Symbol's Value

To check a symbol's value, simply enter the symbol and strike the numeric function key (F1). DEBUG will immediately respond with that symbol's value.

For example,

```
_VAR5↑↑ 16000000477 _
```

In this example, we entered the symbol VAR5 and hit the numeric function key. (DEBUG echoes function keys with two uparrow characters.) After we hit the function key, DEBUG immediately displays the value of VAR5, 16000000477_8.

Refer to Chapter 8 for more information on the numeric function key and the various display modes available.

Address Symbols

The Debugger distinguishes between address symbols and other types of symbols. As described in Chapter 3, under "Rings", all AOS/VS addresses contain a 3-bit ring value from 0 to 7 (in bits 1 through 3 of the address). Thus, any symbol whose value specifies an address will have a value in bits 1 thorugh 3. The following example will help clarify this point:

Suppose you define two symbols, LOC1 and A, in your assembly language module as follows:

```
        .TITLE  MOD     ;Source module MOD.
        .ENT    LOC1,A  ;LOC1 and A are global symbols.
        .LOC    500     ;Start assigning address at absolute
LOC1:   A=500           ;location 500 (octal). LOC1 equals
        .               ;the address 500, and A equals the
        .               ;value 500.
        .
        .END
```

In this example, A receives the absolute value 500_8 (00000000500_8 in 32-bit octal notation). Label LOC1 represents the address 500_8. However, since LOC1 represents an address, its value specifies a ring (usually ring 7) in bits 1 through 3. That is, instead of 500_8, LOC1's value is really 16000000500_8 (i.e., ring 7, location 500).

An *address symbol* is a symbol whose value indicates a ring in bits 1 through 3. Address symbols may be either global symbols (in the .ST file), temporary symbols (defined by $I), or special Debugger symbols.

In the above example, LOC1 is an address symbol; A is not. Similarly, in the following example, LOC2 is an address symbol but B is not.

```
_LOC2,16000000477$I_↵
_B,477$I_
```

The distinction between address symbols and other symbols will be important in the following discussions.

Entering Symbols

DEBUG allows you to enter symbols anywhere you would use the corresponding numeric values. Thus, you can use symbols to specify addresses, to modify the contents of memory and registers, and within expressions.

When it encounters a symbol, DEBUG first looks for that symbol in the temporary symbol table. If the temporary table contains the symbol, DEBUG stops its search and uses that definition.

If DEBUG does not find the symbol in the temporary table, it then searches the .ST symbol table, if one is currently enabled. Again, if DEBUG finds the symbol in the .ST file, it uses that definition.

If DEBUG cannot find a symbol's definition in either table, you will receive an undefined symbol error (i.e., DEBUG will print U?).

The following examples show how you may enter symbols during your debugging session. Assume that all definitions reside in either the temporary or .ST symbol table.

_START/ *000177* _	Use symbol START as an address
_3$A *00000000050* _M+3↵ _	Open AC3 and deposit the value of symbol M plus 3
_LOC+12\ *00000010113* _GH↵ _	Open the two words of memory starting at location LOC+12 and deposit the value of symbol GH

Entering Symbolic Addresses

As discussed previously, all addresses must contain a ring value in bits 1 through 3. Thus, if you supply an address value that does *not* specify a ring, DEBUG automatically inserts the ring value currently in the ring register ($G).

As an example, suppose your .ST symbol table file defines two symbols: LOC and VAR. LOC is an address symbol and equals 16000000731_8 (ring 7, address 731). VAR is not an address symbol; its value is 731_8.

If you enter LOC as an address in a command, DEBUG will use LOC's value as is because it includes a ring specification. However, if you use VAR as an address, DEBUG will insert the default ring into bits 1 through 3 of VAR's value. Assuming the default ring is 7, the following two commands produce identical results:

_LOC/ *000177* _	Open location LOC (i.e., address 16000000731_8). Since LOC's value specifies a ring (7), DEBUG does not refer to the ring register
_VAR/ *000177* _	Open location VAR. Since VAR's value (731_8) does not specify a ring, DEBUG inserts the default ring from ring register (7) into bits 1 through 3. That is, DEBUG converts VAR's value from 731_8 to 16000000731_8 and opens that location

Again, DEBUG only supplies a ring value if you specify an address that does not already contain one. If your address has a ring value, or if your value is not in an address argument, DEBUG does not insert a ring. For example,

_VAR;6/ *000000* _	The expression VAR;6 equals 14000000731_8. Since this value specifies a ring (6), DEBUG does not refer to the ring register

Refer to "Rings" in Chapter 3 and "Ring Register" in Chapter 4 for more information.

Debugger's Symbolic Displays

DEBUG can display addresses and the contents of memory and registers in symbolic mode. The following two sections explain how DEBUG presents symbolic displays and how you can control them.

Displaying Symbolic Addresses

The Debugger always tries to present addresses in symbolic mode; that is, as a symbol plus a numeric value:

addr-symbol *[+offset]*

where:

addr-symbol	is an address symbol whose definition resides in either the temporary symbol table or the .ST symbol table file (see "Address Symbols" earlier in this chapter)
offset	is a numeric value that, when added to **addr-symbol**, equals the address DEBUG is displaying

As an example, suppose your program's .ST symbol table contains the following definitions:

Symbol	**Octal Value**
X	00000000575
BEGIN	16000000573
LOC1	16000000576
LOC2	16000000600

If you step through memory one word at a time starting at address BEGIN, the Debugger would present addresses as follows:

```
_BEGIN/ 000000 _↓        Open location BEGIN (i.e., 16000000573). Since we close with carriage return,
BEGIN+1/ 000000 _↓       DEBUG displays the next location. DEBUG presents that address as BEGIN+1.
BEGIN+2/ 000000 _↓       When it presents location 16000000576, DEBUG uses symbol LOC1 rather than
LOC1/ 000000 _↓          displaying BEGIN+3
LOC1+1/ 000000 _↓
LOC2/ 000000 _↓
_
```

Note that DEBUG presents each address as a symbol plus a numeric offset (if the offset is 0, DEBUG does not display it). DEBUG uses the symbol whose value is closest to, but not greater than, the address DEBUG is displaying.

When presenting addresses, DEBUG only uses symbols whose values specify the correct ring. That is, DEBUG searches for address symbols with the same ring value as the address it wishes to present. In the above example, symbol X's value does not include a ring, so DEBUG ignores X when presenting addresses. BEGIN, LOC1, and LOC2, on the other hand, are all address symbols (in ring 7), so DEBUG does refer to them when displaying addresses.

Our next example shows a $S memory display command. Assume the following symbol table configurations:

Temporary Symbol Table

Symbol	Octal Value
T1	00000001050
TEMP1	16000001045
TEMP2	16000001050

.ST Symbol Table

Symbol	Octal Value
Y	00000001047
ADDR1	16000001045
ADDR2	16000001047

Suppose you want to view the one-word values in locations 1045_8 through 1052_8. You could issue the command

```
_1045,1052$S
```

DEBUG first converts the numbers 1045 and 1052 into 32-bit address values by inserting a ring value from the ring register into bits 1 through 3 of the address (see "Rings" in Chapter 3). If the ring register specifies ring 7, DEBUG converts the above command to

```
_16000001045,16000001052$S
```

As DEBUG presents each address between 16000001045 and 16000001052, it searches the symbol table for a value that is less than or equal to that address. DEBUG then adds a numeric offset, if necessary.

Thus, DEBUG would respond to the $S command with the following display:

```
_1045,1052$S        Display the one-word value between addresses 1045 and 1052, inclusive
TEMP1/ 000000
TEMP1+1/ 000000
ADDR2/ 000000
TEMP2/ 000000
TEMP2+1/ 000000
TEMP2+2/ 000000
_
```

Note that DEBUG uses the symbol TEMP1 rather then ADDR1 when displaying location 16000001045. DEBUG checks the temporary symbol table before the .ST symbol table and, therefore, encounters TEMP1 before ADDR1 in its search for the address.

Also note that DEBUG does not use symbols T1 and Y when presenting addresses. As mentioned earlier, DEBUG only uses address symbols whose values specify the correct ring.

As mentioned at the beginning of this section, DEBUG always tries to present addresses in symbolic mode. The Debugger presents address as numbers only if there is no address symbol with an appropriate value (i.e., in the correct ring and less than or equal to the address DEBUG is displaying).

The simplest way to have DEBUG present numeric addresses is to disable your .ST symbol table file (via a $X command) and delete all temporary symbols (via $J). With no symbols available, DEBUG will present all addresses in numeric mode.

Note that the current display mode does *not* affect how DEBUG presents addresses. See Chapter 8 for more information on display modes.

Displaying the Contents of Memory and Registers in Symbolic Mode

In the previous section, we showed that DEBUG always presents addresses in symbolic mode, if possible. However, when presenting the contents of memory and registers, DEBUG uses the *global display mode* to present values.

When you enter a debugging session, the display mode is set to numeric (mode 1). DEBUG presents the contents of memory and registers as numeric values, by default. You may, however, direct DEBUG to present memory and registers in symbolic mode.

Chapter 8 describes the various modes and explains how to invoke them. The current discussion describes features of DEBUG's symbolic display mode (mode 4).

When DEBUG is in the symbolic display mode, it presents the contents of memory and registers as a symbol plus a numeric offset; that is

symbol *[+offset]*

where:

symbol	is a symbol defined in either the temporary symbol table or the .ST symbol table file
offset	is a numeric value that, when added to symbol, equals the value DEBUG is displaying

As an example, suppose your symbol tables contain the following symbol definitions:

Temporary Symbol Table		**.ST Symbol Table**	
Symbol	**Octal Value**	**Symbol**	**Octal Value**
T1	00000000177	X	00000000205
T2	00000001045	Y	00000001045
T3	00000002077	Z	00000001557

If you want to view the contents of AC0, AC1, AC2, and memory locations 400 and 401, you would enter

```
_0$A 00000000177 ↵
_1$A 00000001047 ↵
_2$A 00000001557 ↵
_400/ 000207 ↵
_401/ 003000 _
```

By default, DEBUG presents the contents of registers and memory in numeric mode

If you now set the display mode to symbolic (see Chapter 8), DEBUG responds to the register and memory examine commands as follows:

```
_0$A T1 ↵
_1$A T2+2 ↵
_2$A Z ↵
_400/ X+2 ↵
_401/ T3+1 _
```

Now DEBUG presents the contents of memory and registers as symbols plus numeric offsets

When DEBUG presents values in symbolic mode, it uses the symbol whose value is closest to, but not greater than, the value DEBUG is displaying. DEBUG then adds a numeric offset to that symbol, if necessary.

Note that DEBUG uses symbol T2 rather than X in the above example. DEBUG always searches the temporary symbol table before the .ST symbol table file and uses the first symbol it finds, if there is a conflict.

Refer to Chapter 8 for more information on display modes and the commands that invoke the symbolic display mode (mode 4).

End of Chapter

Chapter 7 Debugger Expressions

An *expression* is a combination of integers, symbols, and operators. The Debugger allows you to enter expressions any place you can enter numeric values (e.g., as numeric arguments to DEBUG commands or as new values for the contents of memory and registers).

The Debugger represents numeric values in two words (32 bits) of memory. Thus, you may use unsigned numeric values from 0 through $4{,}294{,}967{,}295_{10}$ (i.e., 0 to 2^{32} -1). Signed values can range from $-2{,}147{,}483{,}648_{10}$ to $+2{,}147{,}483{,}647_{10}$.

The Debugger performs all arithmetic operations with 32 bits of precision.

Expression Syntax

The general syntax for Debugger expressions is

[u-operator] operand *[b-operator[u-operator]operand]...*

where:

u-operator	is a unary operator. Unary operators require only one operand, which must appear immediately after the operator
operand	may be an integer, symbol, or another expression
b-operator	is a binary operator. Operands must both precede and follow every binary operator in your expression

You may *not* include spaces within a Debugger expression.

According to the above definition, the following are all legal expressions:

```
START+30    -6*5+7       A*-B+C
```

The following sections describe the various operands and operators you may include in your expressions.

Operands

The Debugger allows you to use the following two types of operands within expressions:

- integers
- symbols

You may *not* use floating point numbers or assembly language instructions within expressions; rather, they are expressions by themselves and cannot be used with operators. (See "Special Expressions" at the end of this chapter.)

Integers

Integers, or constants, are whole numbers. The input format for an integer is

d *[d...][.]*

where:

d	is a digit in the range 0 through 7 for octal integers, or 0 through 9 for decimal integers
	is an optional decimal point. If you include a decimal point, DEBUG interprets the integer in decimal (base 10); otherwise, DEBUG interprets the integer in octal (base 8). Do *not* place a digit after the decimal point

According to this definition, the following are all legal integers:

```
137      17349.    60771
```

The following are all *illegal* integers; the first one contains an illegal character (i.e., A), the second includes a digit after the decimal point, and the third contains a digit that is illegal for octal integers (i.e., 9):

```
21A7      177.3     259007
```

Symbols

You may include any legal symbol in a DEBUG expression:

- symbols defined in the currently enabled symbol table (usually your program's .ST file)
- temporary symbols defined during the current debugging session
- special Debugger symbols that specify the contents of the location counter and the various MV/8000 machine state registers

The following expressions show the use of symbolic operands in expressions (< $ > represents the dollar sign character, not the escape):

```
LOC+50  F*2&B        1<$>A+177
```

If you use an unknown or illegal symbol in an expression, DEBUG returns an error.

Refer to Chapter 6 for a complete discussion on the three types of symbols.

Operators

Table 7-1 lists the operators that the AOS/VS Debugger recognizes:

Table 7-1. DEBUG Operators

	Operator	Operation
Arithmetic Operators	+	Addition or unary plus
	-	Subtraction or unary minus
	*	Multiplication
	|	Division
Logical Operators	~	Logical NOT (unary operator)
	!	Inclusive OR
	!!	Exclusive OR
	&	Logical AND
Relational Operators	>	Greater than
	<	Less than
	=	Equal to
	<=	Less than or equal to
	>=	Greater than or equal to
	<>	Not equal to
Indirect Operators	@	Extract the one-word (16-bit) contents of the supplied address (unary operator)
	#	Extract the two-word (32-bit) contents of the supplied address (unary operators)
ASCII Character Operators	'	Return the ASCII code for the following character (unary operator)
	"	Return the ASCII codes for the following two characters (unary operator)
Half-word Compression Operator	]	Pack values into consecutive 8-bit bytes
Ring Field Operator	;	Insert a value from 0 to 7 into the ring field (bits 1 through 3) of an address

There are two general classes of operators:

- binary operators
- unary operators

Binary operators require two operands; one before and one after the operator. For example, the following expressions contain binary operators:

```
5+3      6&4      A>=B    17]15
```

Unary operators require only one operand, which must appear immediately after the operator. Thus, a unary operator may either begin an expression or follow another operator within an expression. The following characters may function as unary operators:

```
+   -   ~   @   #   '   "
```

Examples of expressions that use unary operators are

```
-5   +4   6*-3   5!~2
```

The following sections describe the various DEBUG operators, both unary and binary.

Arithmetic Operators

Table 7-2 lists the four arithmetic operators.

Table 7-2. Arithmetic Operators

Operator	Operation
+	Addition or unary plus
-	Subtraction or unary minus
*	Multiplication
|	Division

These operators perform standard mathematical operations. Note that the vertical bar (ASCII 174_8) functions as the division operator. (Remember, / is the one-word memory examine command.)

The following expressions show the use of the arithmetic operators:

Expression	**Octal Value**
3+2	5
10-3	5
2*3	6
6|2	3
5+2+1	10

In the above examples, + and - function as binary operators. You may also use them as unary operators, in which case they indicate the sign of the following expression (positive or negative).

Logical Operators

The Debugger provides four logical operators; see Table 7-3.

Table 7-3. Logical Operators

Operator	Operation
~	Logical NOT (complement)
!	Inclusive OR
!!	Exclusive OR
&	Logical AND

To perform a logical operation, DEBUG inspects the bit pattern of the operand(s).

The *logical NOT operator (~)* is a unary operator (i.e., requires only one operand). This operator directs DEBUG to complement each bit of the operand. That is, the result in a given bit position is 1 if the operand contains a 0 in that bit position; the result is 0 if the operand contains a 1 in that bit. For example, the logical NOT operation for the expression ~5 is

```
Bit representation of 5:  00 000 000 000 000 000 000 000 000 000 101
--------------------------------------------------------------------
Result of ~ operation:    11 111 111 111 111 111 111 111 111 111 010
```

Thus, the value of ~5 is 37777777772_8.

The *inclusive OR operator (!)* directs DEBUG to compute the logical OR of two operands. The result in a given bit position is 1 if either or both operands contain a 1 in that bit position. Otherwise, the result in that bit position is 0. The following example shows how DEBUG evaluates the expression 6!3:

```
Bit representation of 6:  00 000 000 000 000 000 000 000 000 000 110
Bit representation of 3:  00 000 000 000 000 000 000 000 000 000 011
--------------------------------------------------------------------
Result of ! operation:    00 000 000 000 000 000 000 000 000 000 111
```

Thus, the value of the expression 6!3 is 7_8 (i.e., 111_2).

The *exclusive OR operator (!!)* directs DEBUG to compute the logical XOR of two operands. DEBUG returns a 1 in a given bit position if one and only one of the operands contains a 1 in that bit position. If both operands contain 0s or 1s in the same bit position, DEBUG returns a 0 in that bit. For example, DEBUG evaluates the expression 6!!3 as follows:

```
Bit representation of 6:  00 000 000 000 000 000 000 000 000 000 110
Bit representation of 3:  00 000 000 000 000 000 000 000 000 000 011
--------------------------------------------------------------------
Result of !! operation:   00 000 000 000 000 000 000 000 000 000 101
```

The value of 6!!3 is 5_8 (i.e., 101_2).

The *logical AND operator (&)* directs DEBUG to return a 1 in a given bit position only if both operands contain a 1 in that bit position. Otherwise, DEBUG returns a 0. The following example shows how DEBUG evaluates 6&3:

```
Bit representation of 6:  00 000 000 000 000 000 000 000 000 000 110
Bit representation of 3:  00 000 000 000 000 000 000 000 000 000 011
------------------------------------------------------------------------
Result of & operation:    00 000 000 000 000 000 000 000 000 000 010
```

The value of 6&3 is 2_8 (i.e., 10_2).

Note that the Debugger operates on all 32 bits of the operand(s) when performing a logical operation. For instance, in our logical NOT example, all leading zeros became ones.

More examples of logical expressions follow:

Expression	Octal Value
~17	37777777760
30!20	30
30!!20	10
30&20	20
~5!100	37777777772
177&17	17

Relational Operators

Relational operators direct the Debugger to compare the unsigned values of two operands. If the comparison is true, the value of the expression is -1; if the comparison is false, the expression's value is 0.

You will find relational operators particularly useful when performing conditional operations; for example, when setting conditional breakpoints (see Chapter 4) or performing memory searches (see Chapter 3).

Table 7-4 lists the six relational operators.

Table 7-4. Relational Operators

Operator	Operation
>	Greater than
<	Less than
=	Equal to
>=	Greater than or equal to
<=	Less than or equal to
<>	Not equal to

The following examples show how DEBUG evaluates relational expressions:

Expression	Octal Value	
5>3	-1	(True)
7=3	0	(False)
17>=17	-1	(True)
107<>41	-1	(True)
6>7	0	(False)
104<=277	-1	(True)

The Debugger always presents integer values as unsigned. Thus, when displaying a true relation, DEBUG writes '37777777777', not '-1'. The bit representations for the these two integers is identical.

Indirect Operators

The Debugger provides two unary *indirect operators:* the commercial at sign (@) and the number sign (#). The indirect operators require you to supply one operand -- an address. The value of an indirect expression is the contents of that memory location.

Using the @ and # operators, you may extract either a one- or two-word value starting at the specified address. Table 7-5 summarizes the two indirect operators.

Table 7-5. Relational Operators

Operator	Operation
@	Extract the one-word (16-bit) value starting at the supplied address
#	Extract the two-word (32-bit) value starting at the supplied address

As an example, assume memory contains the following values:

Memory (Word) Location	16-bit Contents
500	000010
501	000020
502	000600
600	100777
601	000001

The following expressions show how DEBUG evaluates indirect operators given the above memory configuration:

Expression	Octal Value	Comment
@500	10	The 16-bit value starting at address 500_8 equals 10
@501	20	The 16-bit value starting at address 501_8 equals 20
@500+1	11	The 16-bit value at address 500_8 is 10; add 1 and the result is 11
@(500+1)	20	The 16-bit value at address 501_8 is 20
#500	2000020	The 32-bit value at address 500_8 equals 2000020_8
#600	20177600001	The 32-bit value at address 600_8 equals 20177600001_8
@(@502)	100777	Double indirection: the parenthetical expression equals 600_8 (i.e., the 16-bit contents of location 502_8); the 16-bit contents of address 600_8 equals 100777_8

Since the operand you pass to an indirect operator represents an address, DEBUG requires that value to specify an address ring. That is, bits 1 through 3 of each address in your program contain a ring value from 0 to 7. Therefore, each address value DEBUG uses must also contain a ring value.

If the address value you supply to DEBUG does not specify a ring in bits 1 through 3, DEBUG will use the default ring value stored in the ring register ($G). Refer to "Rings" in Chapter 3 for more information on rings and the ring register.

ASCII Character Operators

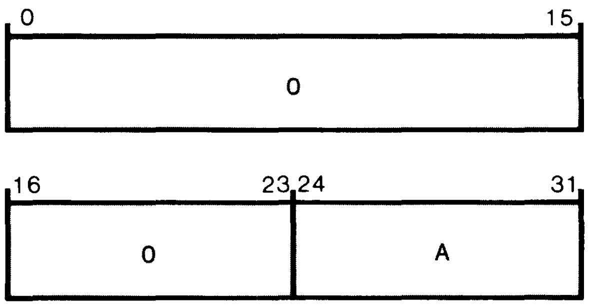
The Debugger provides two unary operators that convert characters to their 8-bit ASCII values. Table 7-6 describes these ASCII character operators.

Table 7-6. ASCII Character Operators

Operator	Operation
‘	Return the ASCII code for the following character
“	Return the ASCII codes for the following two characters

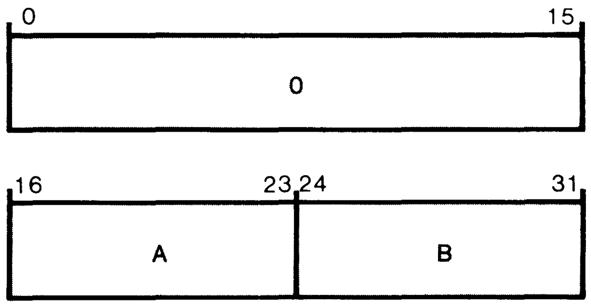
The ASCII character operators are unique in that they require character operands, *not* numeric values. The examples later in this section illustrate this rule.

When DEBUG encounters the operator ‘, it stores the ASCII code for the following character in the low-order 8 bits of the integer representation (bits 24-31 in 32-bit notation). For example, DEBUG stores the expression ‘A as follows:



SD-02283

The “ operator directs DEBUG to store the ASCII codes for the next two characters in consecutive bytes. In a two-word value, the two ASCII codes will reside in the low-order bytes (bits 16 through 23 and 24 through 31). Thus, the Debugger represents the expression “AB as



SD-02284

DEBUG always strips the parity bit from the character before returning its ASCII value. Thus, the first bit of each character byte always equals 0.

If you pass too many characters to the ASCII operators, DEBUG returns an error.

The following expressions show how DEBUG evaluates the ASCII character operators:

Expression	**Octal Value**	**Comment**
'A	101	Return the ASCII code for A (i.e., 101_8)
'B	102	The ASCII code for B is 102_8
"AB	40502	Place the ASCII codes for A and B in consecutive bytes
'5	65	The ASCII code for the character 5 is 65_8
"32	31462	Place the ASCII codes for 3 (63_8) and 2 (62_8) in consecutive bytes
"A/	40457	Store the ASCII codes for A (101_8) and / (57_8) in consecutive bytes
"ABC	ERROR	The " operator accepts ony 2 characters
'AB	ERROR	The ' operator accepts only 1 character

As mentioned earlier, the operators ' and " require characters, not numeric values, as operands. In the expressions '5 and "32, DEBUG uses the ASCII codes for the characters 5, 3, and 2.

In the expression "A/, we supplied a slash to an ASCII character operator. DEBUG treated this character as any other; that is, DEBUG stored its ASCII value. DEBUG did *not* interpret / as the one-word memory examine command.

In general, DEBUG allows you to pass any character to the ASCII operators -- there are no restrictions. In all cases, DEBUG simply stores the ASCII value for the character (with the parity bit set to 0).

Appendix B lists the ASCII codes for all characters.

Half-word Compression Operator

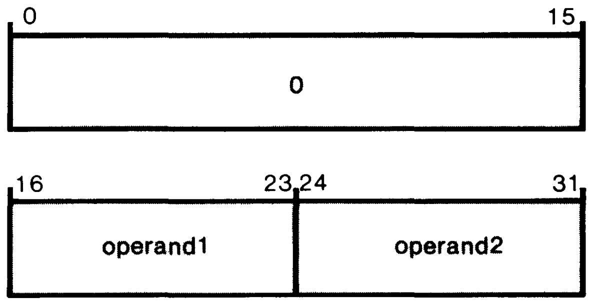
The *half-word compression operator (])* allows you to pack values into consecutive 8-bit bytes of memory. The syntax for using this operator is

operand1]operand2

When DEBUG encounters the] operator, it performs the following operation:

operand1 $*$ 400_8 + (operand2 $*$ 377_8)

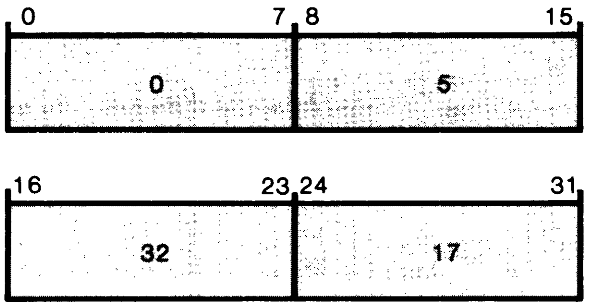
That is, DEBUG represents the expression operand1]operand2 as follows:



SD-02285

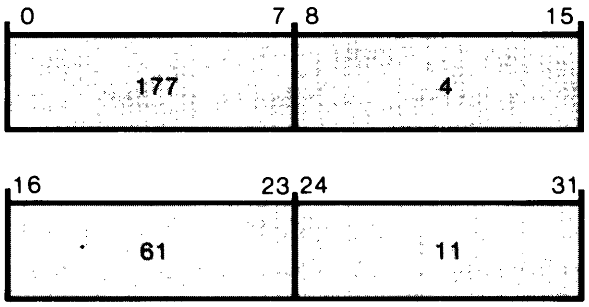
Since the] operator places operand values in 8-bit bytes, you will normally use values between 0 and 377_8, inclusive (or signed values between -200_8 and +177_8). If an operand value is too large, DEBUG will truncate the value to 8 bits before performing the] operation.

If you wish to pack more than two values, apply the] operator more than once. For example, DEBUG represents the expression 5]32]17 as follows:



SD-02286

Similarly, DEBUG stores 177]4]61]11 in the following manner:



SD-02287

The following expressions show the use of the half-word compression operator (]):

Expression	**Octal Value**
1]1	401
3]27	1427
14]31	6031
3]3]3	601403
15]7]1	3203401
22]11]6]2	2202203002
1]2]3]4	00100401404

Ring Field Operator

AOS/VS organizes memory into a hierarchy of 8 *rings*, numbered 0 to 7. Bits 1 through 3 of each AOS/VS memory address indicates one of these 8 ring values.

As an example, to indicate the 66^{th} word in ring 7, you enter the value 16000000066_8. Note that bits 1 through 3 of the address specify the value 7.

Rather than explicitly enter the 32-bit address (e.g., 16000000066), you may use the ring field operator (;). The *ring field operator* inserts a ring value into bits 1 through 3 of a memory address.

The syntax for using the ring field operator is

address;ring

Thus, to represent the address value 16000000066_8 (ring 7, address 66), you could enter the expression 66;7.

Since the ring field is only 3 bits long, the ring value you supply to the ; operator should be between 0 and 7, inclusive. If you supply a ring value that is too large, DEBUG will truncate the value to 3 bits before placing it in the address.

Also, if the address operand you supply to the ; operator already has a ring value in bits 1 through 3, DEBUG replaces it with the new ring value (i.e., the value in the ring argument).

The following expressions show how DEBUG interprets the ring field operator (;):

Expression	**Octal Value**
100;7	16000000100
17001;6	14000017001
742;7	16000000742
342111;5	12000342111
1044;7	16000001044

Refer to "Rings" in Chapter 3 for more information.

Operator Precedence

All Debugger operators are of equal precedence. Thus, DEBUG evaluates the operators in an expression in order from left to right.

You may use parentheses to impose operator priority. The Debugger always evaluates parenthetical expressions first.

The following examples show how DEBUG evaluates expressions with and without parentheses:

Expression	**Octal Value**
5-2+1	4
5-(2+1)	2
6<4+3	3
6<(4+3)	-1 (true)
4!5+2	7
4!(5+2)	7
"AB+100*2	101404
"AB+(100*2)	40702
3]2*10	14020
3](2*10)	1420

Special Expressions

In addition to arithmetic expressions that use the operators described above, DEBUG allows you to use the following two types of *special expressions:*

- floating point numbers
- assembly language instructions

We refer to these as special expressions because DEBUG does not allow you to use floating point numbers or assembly language instructions as operands in expressions. Rather, they are expressions by themselves. That is, you may enter a floating point number or instruction any place you can enter an expression, but you cannot perform operations on them.

When entering instructions and floating point numbers in registers, be sure the value is less than or equal to two words (32 bits) in length. If the value is too long, DEBUG will return an error and will *not* modify the register. For example, if you open accumulator 0 (AC0) and enter the instruction LWLDA 0,1, DEBUG will return an error because LWLDA is a three-word instruction and AC0 is only two words long.

If, on the other hand, you enter an instruction or floating point number in an open memory location, DEBUG will modify as many words of memory as are necessary to represent the value. "Legal Entry Value" in Chapter 3 provides more information on entering instructions and floating point numbers in memory.

Floating Point Numbers

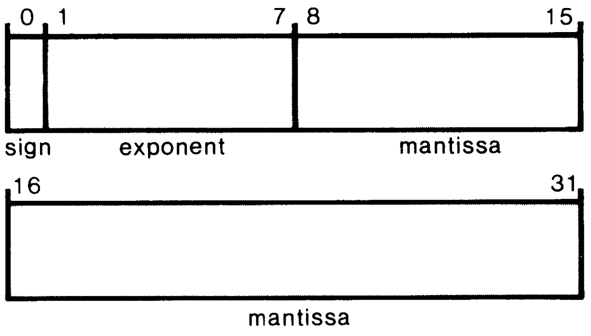
DEBUG allows you to enter *floating point numbers* anywhere you can use an expression. Floating point numbers have three components:

- A *sign* (positive or negative)
- A fractional part called the *mantissa;* DEBUG normalizes all floating point numbers such that the mantissa is always greater than or equal to 1/16 and less than 1
- An *exponent,* expressed in excess-64 representation (the *AOS/VS Macroassembler Reference Manual* explains excess-64 representation)

You can specify two types of floating point numbers:

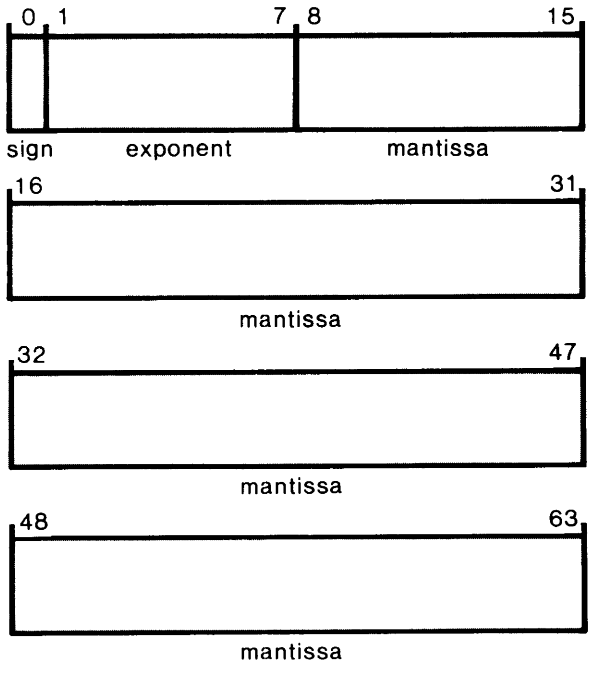
- single precision (2 words)
- double precision (4 words)

DEBUG represents *single precision floating point numbers* in two words of memory as follows:



SD-02288

Double precision floating point numbers reside in four words of memory:



SD-02289

Use the E and D characters to designate single and double precision floating point values, respectively. The following is the input format for *floating point numbers:*

[sign] d *[d...]* .d *[d...]* $\left\{\begin{matrix}E\\D\end{matrix}\right\}$ *[sign]* d *[d]*

where:

sign	indicates the sign of a value (positive or negative) and is one of the following characters: + or -. If the sign appears at the beginning of the number, then it defines the sign of that number. If a sign appears after the letter E or D, it defines the exponent's sign. If you do not supply a sign, DEBUG assumes the value is positive
d	is a digit in the range 0 through 9. The Debugger always interprets floating point numbers in decimal (base 10)
	is a decimal point (period). You must enter at least one digit before and one digit after the decimal point
E	specifies single precision (2 word) floating point number representation
D	specifies double precision (4 word) floating point number representation

According to the above syntax, you must conform to the following rules when entering floating point numbers:

- You must include a decimal point
- You must both precede and follow the decimal point with digits
- You must specify either single (E) or double (D) precision representation
- You must follow the E or D with one or two digits

Also, DEBUG evaluates all floating point numbers in decimal (base 10).

The following examples show legal floating point numbers and their octal values:

Floating Point Expression	**Octal Value**
1.1E1	040660 000000
5.6E1	041070 000000
6.4937D11	045227 030567 155200 000000
-3.400005D-02	137613 041642 166377 071070
25433.E-2	041376 052172
254.33E0	041376 052172
0.25433E03	041376 052172

The integers under "Octal Value" each represent one-word values (i.e., the values are in numeric word display mode -- see Chapter 8 for more information on this display mode).

Remember, a floating point number is the equivalent of an entire expression; do *not* use floating point numbers as operands within other expressions.

Instructions

Anywhere you can enter an expression, you can also enter an *MV/8000 assembly language instruction.* DEBUG will compute the assembled value of the instruction and use that value accordingly.

If you enter a memory reference (MRI) instruction and do not specify an addressing mode (or index), DEBUG will try to use PC relative addressing by default (i.e., index value of 1). If DEBUG cannot resolve the address relative to the program counter (PC), DEBUG will then try to use absolute addressing (i.e., index value 0).

As with floating point numbers, instructions are the equivalent of complete expressions; you may *not* use them as operands within other expressions.

Checking an Expression's Value

To check an expression's value, simply enter the expression and strike the numeric function key (F1). DEBUG will immediately respond with that expression's value.

For example,

```
_500+22↑↑ 00000000522 _
```

In this example, we entered the expression 500+22 and hit the numeric function key (DEBUG echoes function keys with two uparrow characters). After we hit the function key, DEBUG evaluated the expression 500+22 and displayed its value, 00000000522_8.

Similarly, to check the value of a floating point number or an assembly language instruction, enter the expression and strike the numeric function key:

```
_XWLDA 0,2↑↑ 20302200002 _
```

Refer to Chapter 8 for more information on the numeric function key and the various display modes available.

End of Chapter

Chapter 8
Debugger Display Modes

The AOS/VS Debugger can display data in a variety of different modes. In the following sections, we first describe the various *display modes*, and then show how to invoke them.

The 10 Display Modes

The Debugger can display data in any of the 10 modes listed in Table 8-1.

Table 8-1. Display Modes

Mode	Mode Number	Description
Numeric	1	Present each value as an octal number (default display mode)
Numeric Word	2	Present each value as a series of one-word (16-bit) octal numbers
Instruction	3	Interpret each value as an instruction, complete with arguments
Symbolic	4	Present each value as a symbol plus a numeric offset
Half-word	5	Present each value as a series of half-word (8-bit) octal numbers
Byte-pointer	6	Present each value as a byte pointer
ASCII	7	Interpret each 8-bit quantity as an ASCII code and present the corresponding character
Floating point	8	Display each value as a floating point number
System call	9	Interpret each value as a system call code and print the corresponding system call
AOS/VS error message	0	Interpret each value as an AOS/VS error code and print the corresponding error message

Please note that DEBUG uses the display modes when presenting the contents of memory and registers, but does *not* use the modes when displaying memory addresses. The Debugger always tries to present addresses in symbolic mode. See "Displaying Symbolic Addresses" in Chapter 6 for more information.

Numeric Mode (1)

By default, the Debugger presents values in the *numeric display mode* (mode 1); that is, as octal integers.

Depending on the inherent length of the value (i.e., 1 to 4 words), DEBUG displays the value as follows:

Length of Value	**Representation in Numeric Mode**
One word (16 bits)	A single one-word octal integer
Two words (32 bits)	A single two-word octal integer
Three words (48 bits)	Two octal integers: a one-word value followed by a two-word value
Four words (64 bits)	Three octal integers: a one-word value, a two-word value, and another one-word value

As an example, suppose memory contains the following values:

Memory (Word) Location	**Contents**
500	000000
501	000001
502	000002
503	000003
504	000004
505	000005
506	000006
507	000007
510	000010

If you are in the numeric display mode, DEBUG responds to memory display commands as follows:

```
_5000/ 000000 _↓
16000000501/ 000001 _↓
16000000502/ 000002 _↲
_
```

Examine single words of memory; DEBUG displays them as one-word octal integers

```
_500\ 00000000001 _↓
16000000502\ 00000400003 _↓
16000000504\ 00001000005 _↲
_
```

Examine double words of memory; DEBUG presents them as two-word octal values

```
_500,505,,3,3$S
16000000500/ 000000 00000200002
16000000503/ 000003 00001000005
_
```

Display the three-word values between addresses 500 and 505; DEBUG presents each three-word value as a one-word integer followed by a two-word integer

```
_500,507,,4,4$S
16000000500/ 000000 00000200002 000003
16000000504/ 000004 00001200006 000007
_
```

Display the four-word values between addresses 500 and 507: DEBUG presents each four-word value as a series of three integers -- one-word, two-words, and one-word in length

Note that the Debugger does not present three- and four-word values as single octal numbers. Rather, DEBUG presents the first word as as seperate integer from the rest of the value.

Most three- and four-word values represent assembly language instructions. Thus, when displaying them as numbers, you usually want to view the first word of the instruction (the opcode) as a separate value from the instruction's arguments. The numeric mode provides a convenient way to view instructions and their arguments as octal values.

Numeric Word Mode (2)

Using the *numeric word display mode* (mode 2), you can direct the Debugger to present values as a series of one-word (16-bit) octal integers.

For example, assume memory contains the following values:

Memory (Word) Location	**Contents**
500	000000
501	000001
502	000002
503	000003
504	000004

If you are in numeric word display mode, DEBUG presents data as a series of one-word integers as follows:

```
_500/ 000000 _↓                   Examine one-word memory locations
16000000501/ 000001 _↓
16000000502/ 000002 _↵
_

_5000\ 000000 000001 _↓           Examine two-word memory locations
16000000502/ 000002 000003 _↵
_

_0$A 000000 000000 _↵             Examine AC0 and AC1
_1$A 000000 000000 _↵
_
```

Instruction Mode (3)

When set to the *instruction display mode* (mode 3), the Debugger presents values as MV/8000 assembly language instructions, complete with arguments (e.g., accumulators, displacements, etc.).

When presenting data as an instruction, DEBUG interprets the first word as the instruction's operation code (or opcode, for short). From this value, DEBUG can determine how long the instruction is (i.e., from 1 to 4 words).

When you examine memory via the / and \ commands, DEBUG displays as many words of memory as are necessary to present the complete instruction. That is, after interpreting the first word of the instruction (the opcode), DEBUG will use as many subsequent words of memory as required by that instruction. As an example, consider the following memory examine commands (instruction display mode):

```
_700/ XWLDA 0,1 _↓                When in instruction mode, the / and \ commands display the same values.
16000000702/ LCALL 0,1,4 _↓       Note that stepping forward through memory displays consecutive
16000000706/ WADD 0,1 _↵          instructions, not consecutive locations
_

_700\ XWLDA 0,1 _↓
16000000702\ LCALL 0,1,4 _↓
16000000706\ WADD 0,1 _↵
_
```

Refer to "Examining Locations in Various Modes" and "Displaying Instructions" in Chapter 3 for more information on examining memory in the instruction display mode.

When you are in the instruction display mode, the Debugger presents all registers in numeric mode, by default. This is a desirable feature since you rarely want to view register values as instructions.

If you *do* want to display a register as an instruction, first open the register (as described in Chapter 4). After DEBUG displays the register's value, redisplay that value in the instruction mode using a local display command (we describe local and global display commands later in this chapter).

When displaying values of definitive length (e.g., registers), DEBUG may have to truncate or pad the value to produce an instruction. DEBUG truncates and pads values as follows:

- If the first word (opcode) indicates an instruction that is *shorter* than the data value, DEBUG truncates on the right (i.e., DEBUG uses the most significant words)
- If the first word (opcode) indicates an instruction that is *longer* than the data value, DEBUG pads on the right with zeros

Three final notes on the instruction display mode:

- The Debugger presents all I/O instructions as LEF instructions
- DEBUG never prints MRI index values of 0 or 1
- When displaying MRI instructions, DEBUG presents displacement values in symbolic mode if the addressing index is 0 or 1 (i.e., PC relative or absolute addressing); if the addressing index is 2 or 3 (i.e., AC relative addressing), DEBUG displays the address as an octal integer

Symbolic Mode (4)

Using the *symbolic display mode* (mode 4), you can direct the Debugger to present data as a symbol plus a numeric offset.

When in the symbolic mode, DEBUG searches for a symbol whose value is less than or equal to the data value. DEBUG then adds a numeric offset to that symbol, if necessary. For example, suppose the Debugger recognizes the following symbols:

Symbol	**Octal Value**
VAR1	00000000500
VAR2	00000000510
VAR3	00000000512

Using these symbol definitions, DEBUG presents values as follows:

Octal Value	**Symbolic Representation**
00000000505	VAR1+5
00000000507	VAR1+7
00000000510	VAR2
00000000511	VAR2+1
00000000512	VAR3
00000000513	VAR3+1
00000000470	470

Note that DEBUG uses the symbol whose value is closest to, but not greater than, the value DEBUG is displaying.

When searching for symbol values, DEBUG checks the temporary symbol table first and your program's .ST symbol table second. Chapter 6 contains more information on these two symbol tables.

If DEBUG cannot find a symbol less than or equal to the data value, it presents the value as an octal integer. In the last line in the above example, there is no symbol value less than or equal to 470, so DEBUG presents that value as a number.

Similarly, if there are no symbols defined, DEBUG presents values in numeric mode.

Refer to Chapter 6, especially "Debugger's Symbolic Displays", for more information on symbols, symbol tables, and symbolic displays.

Half-word Mode (5)

In *half-word display mode* (mode 5), the Debugger presents data as a series of half-word (8-bit) octal integers.

For example, assume AC0 holds the value 11420245505_8 and memory contains the following values:

Memory (Word) Location	Contents
1000	000000
1001	000001
1002	000002
1003	000003

When the Debugger is in the half-word display mode, it responds to commands as follows:

```
_1000/ 000 000 _↓                  Examine one-word memory locations
16000001001/ 000 001 _↓
16000001002/ 000 002 _↓
_

_1000\ 000 000 000 001 _↓          Examine two-word memory locations
16000001002\ 000 002 000 003 _↓
_

_0$A 114 101 113 105 _             Examine accumulator 0 (AC0)
```

Again, each three-digit octal integer represents the value of an 8-bit byte.

Byte-pointer Mode (6)

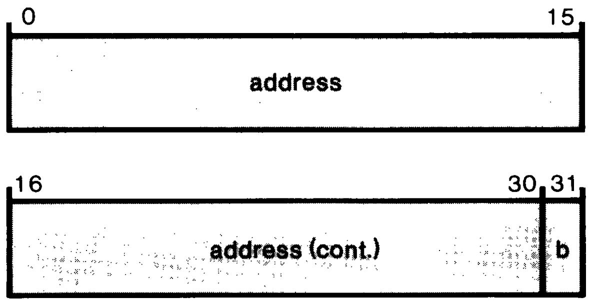
A *byte pointer* is a value that specifies the address of an 8-bit byte in memory. The format for a byte pointer is

2*address+b

where:

address	is the address of the word that contains the byte
b	is a one-bit value: 0 specifies the left byte of the the word at location **address**; 1 indicates the right byte

According to this format, the internal representation of a byte pointer is



SD-02290

When the Debugger is in *byte-pointer display mode* (mode 6), it interprets data as byte pointers; that is, DEBUG displays values as two integers: an address value and a one-bit byte indicator. If DEBUG is presenting a 32-bit value, the first 31 bits are the address and the last bit (bit 31) is the byte indicator. Similarly, in a 16-bit value, the first 15 bits represent the address and the last bit (bit 15) is the byte indicator.

As an example, suppose your accumulators contain the following values:

Accumulator	**Octal Value**
AC0	34000001000
AC1	34000001001
AC2	00000002136
AC3	00000002137

If you are using the byte-pointer display mode, DEBUG presents the accumulators as follows:

```
_0$A 16000000400 0 ↵   Examine the accumulators in byte-pointer mode; DEBUG presents each
_1$A 16000000400 1 ↵   accumulator as two integers: an address followed by a one-bit value
_2$A 00000001057 0 ↵
_3$A 00000001057 1 _
```

ASCII Mode (7)

The *ASCII display mode* (mode 7) directs DEBUG to interpret data as a series of ASCII characters. The Debugger will interpret each 8-bit byte as an ASCII code and will display the corresponding character, if possible. DEBUG does *not* consider the parity bit (the leftmost bit) when interpreting a byte as an ASCII character code.

For example, suppose the four accumulators contain the following values:

Accumulator	**Octal Value**
AC0	11420245505
AC1	10220246104
AC2	06114431464
AC3	00101000101

If you are using the ASCII display mode, DEBUG will present the accumulators as follows:

```
_0$A LAKE ↵             Examine the four accumulators in the ASCII display mode
_1$$A BALD ↵
_2$A 1234 ↵
_3$A <1><4><0>A_
```

In each of these examples, DEBUG interprets the two-word accumulator value as a series of four ASCII characters.

If the ASCII code in a byte does not represent a printable character, DEBUG prints out the octal value of the byte (including the parity bit) inside angle brackets, < > . In the last line of the above example, DEBUG presents the first three bytes of AC3 as octal numbers because their values do not represent printable ASCII characters.

Appendix B lists the ASCII characters and their octal values.

Floating Point Mode (8)

The *floating point display mode* (mode 8) instructs the Debugger to present data as single precision (two-word) or double precision (four-word) floating point numbers. DEBUG presents floating point values as follows:

[sign] d.d *[d...]* $\left\{\begin{matrix}\text{E}\\\text{D}\end{matrix}\right\}$ *[sign]* d *[d]*

where:

sign	indicates the sign of a value (positive or negative) and is one of the following characters: + or -. If the sign appears at the beginning of the number, then it defines the sign of that number. If the sign appears after the letter E or D, it defines the exponent's sign. If DEBUG does not indicate a sign, the value is positive
d	is a digit in the range 0 through 9. The Debugger presents floating point numbers in decimal (base 10)
	is a decimal point
E	specifies single precision (two-word) floating point number representation
D	specifies double precision (four-word) floating point number representation

When presenting floating point numbers, DEBUG conforms to the following rules:

- DEBUG always places one digit to the left of the decimal point and either 7 or 16 digits to the right, depending on the precision of the floating point number
- All floating point numbers are decimal (i.e., base 10)
- DEBUG always presents the exponent after the letter E or D, even if its value is 0

When in floating point mode, DEBUG displays two-word values as single precision floating point numbers and four word values as double precision floating point numbers. For example,

```
_0$A 7.1791535E+13 ↵
_1$A 6.5970912E+1 ↵
_2$A 0.0000000E+0 ↵
_
```

DEBUG presents fixed-point accumulators (two-word values) as single precision floating point numbers

```
_0$F 1.8231491520767509E-63 ↵
_1$F 4.0250000000000000E+2 ↵
_2$F 0.0000000000000000E+0 ↵
_
```

DEBUG presents the floating point accumulators (four-word values) as double precision floating point numbers

When examining memory in floating point display mode, the / and \ commands take on new meaning. The / command directs DEBUG to present *two* words of memory (not one) as a single precision floating point number. The \ command displays *four* words of memory (not two) as a double precision floating point value. For example,

```
_500/ 1.7507244E-40 _↓                      In floating point mode, / directs DEBUG to present two words
16000000502/ -7.8076683E+43 _↓              of memory
16000000504/ 5.8790504E-2 _⅃
_

_570\ 1.7507260661445691E-40 _↓             The \ command directs DEBUG to present four words of
16000000574\ 5.8790518001246290E-2 _↓       memory
16000000600\ 8.1445004799992258E-30 _⅃
_
```

As discussed in Chapter 3, once you examine a memory location, that location is open for modification (i.e., you can enter a new value). Normally, you can enter the new value in any of the various modes. However, when examining memory in floating point mode, you should modify locations only by entering floating point numbers. If you enter a different type of value, you may receive unexpected results.

For more information on floating point values, refer to "Floating Point Numbers" in Chapter 7.

System Call Mode (9)

The *system call display mode* (mode 9) instructs DEBUG to interpret each value as a system call code and to print the name of the corresponding system call.

System calls are predefined macros that perform common assembly language operations. Each system call name begins with a question mark and contains uppercase letters (e.g., ?OPEN, ?READ, ?WRITE). The *AOS/VS Programmer's Manual* describes each system call in detail.

Each system call occupies two entries in your program file. The first entry is a jump instruction that passes control to the system call handling routine (JSR for 16-bit programs and XJSR for 32-bit programs). The second entry contains a system call code -- a one-word value that identifies a particular system call. Files SYSID.16.SR and SYSID.32.SR, supplied with your AOS/VS software package, list the numeric code for each system call.

When in system call display mode, DEBUG interprets each value as one of the system call codes. If the code is valid, DEBUG prints the corresponding system call. If the code does not represent a system call, DEBUG returns an error.

As an example, suppose your source code contains the following statements:

```
        .TITLE  MOD
        .NREL
        .ENT    LOC     ;Declare symbol LOC in an .ENT
                 .      ;statement so DEBUG will recognize
                 .      ;it (see Chapter 6 for information
                 .      ;on symbol recognition).
LOC:    ?OPEN           ;Issue on ?OPEN system call at
                 .      ;location LOC. (?OPEN opens a
                 .      ;file.)
                 .
        .END
```

As mentioned above, all system calls in 32-bit programs expand to three words in your program file (i.e., a two-word XJSR instruction followed by a one-word system call code). Thus, the two-word value at location LOC represents an XJSR instruction; the following word (at location LOC+2) contains the numeric code for an ?OPEN system call.

To verify this, inspect location LOC+2 in the system call display mode (mode 9):

```
_LOC+2/ ?OPEN
```

DEBUG recognizes the one-word value at LOC+2 as the system call code for ?OPEN.

AOS/VS Error Message Mode (0)

The *AOS/VS error message display mode* (mode 0), directs the Debugger to interpret values as AOS/VS error codes and to print the corresponding error messages.

As an example, suppose AC0 contains the value 00000000025_8. If you are using the AOS/VS error message display mode, DEBUG will present the contents of AC0 as follows:

```
_0$A FILE DOES NOT EXIST_
```

DEBUG interprets the contents of AC0 (i.e., 25_8) as a CLI error code and prints the corresponding error message.

The Debugger's AOS/VS error message response for a value n is the same response you receive when you issue the CLI command

```
) MESSAGE n↵
```

where n is a numeric error code.

The *AOS and AOS/VS Command Line Interpreter (CLI) User's Manual* presents more information on the MESSAGE command and AOS/VS errors.

Setting Display Modes

The Debugger supplies several different ways to invoke the various display modes. Generally, you can issue two types of display mode commands:

- global
- local

The *global display mode* is the display mode that DEBUG is currently using to present data. When you enter a debugging session, the global display mode is numeric (mode 1).

You can change the global display mode in two ways:

- issue a global display mode command
- enter a new value in the global display mode register (via $T)

After you change the global display mode, DEBUG presents data in the new mode. You can alter the global display mode as often as you wish.

As an alternative to changing the global display mode, you can enter a *local display mode command.* A local display command directs DEBUG to present the last value it typed out in a new display mode. You can also use local display commands to obtain the value of a symbol or expression. Local display commands do *not* alter the global display mode.

The local and global display commands vary depending on whether your keyboard has *function keys* (blank keys at the top of the keyboard). Data General's DASHER D1 (6052), D2 (6053), D3, D4, D5, and D200 video display terminals all have function keys. The DASHER D100 terminal and hard-copy devices TP1 and TP2 do not.

The following sections explain how to change local and global display modes for keyboards with and without function keys; only read the section that describes your keyboard. Table 8-2 summarizes the various display mode commands.

Regardless of whether your keyboard has function keys, you can alter the global display mode by directly modifying the display mode register. Refer to the last section of this chapter, "Display Mode Register", for information.

Table 8-2. Display Mode Commands for Various Keyboards

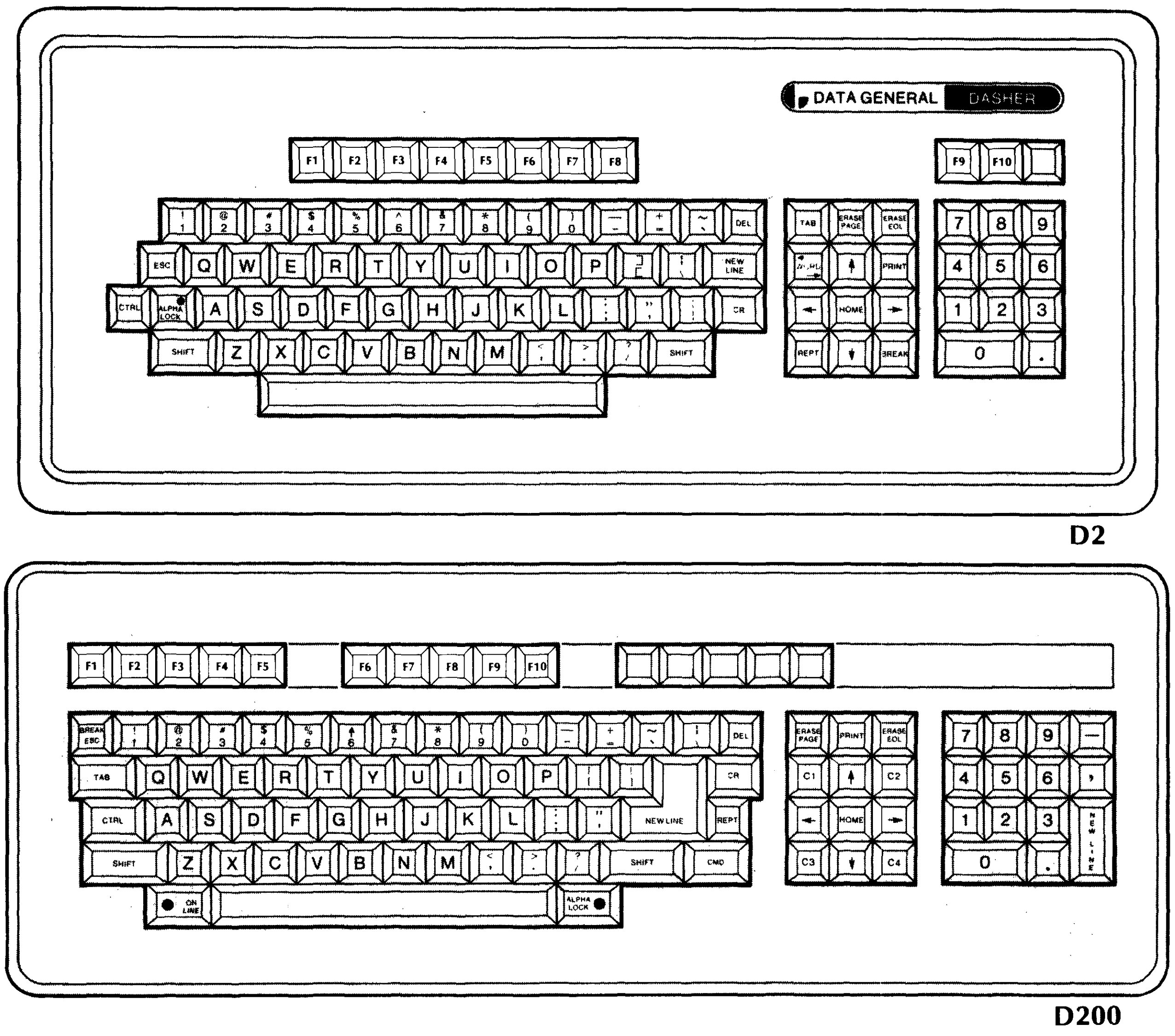
Display Mode	Keyboard Commands		
	D2 & D200	D1	Keyboards Without Function Keys
Local *			
Numeric (1)	F1	F1	TAB 1
Numeric Word (2)	F2	F2	TAB 2
Instruction (3)	F3	F3	TAB 3
Symbolic (4)	F4	F4	TAB 4
Half-word (5)	F5	F5	TAB 5
Byte-pointer (6)	F6	F6	TAB 6
ASCII (7)	F7	F7	TAB 7
Floating point (8)	F8	F8	TAB 8
System Call (9)	F9	TAB 9	TAB 9
AOS/VS Error Message (0)	F10	TAB 0	TAB 0
Global **			
Numeric (1)	CTRL-F1	CTRL-F1	TAB TAB 1
Numeric Word (2)	CTRL-F2	CTRL-F2	TAB TAB 2
Instruction (3)	CTRL-F3	CTRL-F3	TAB TAB 3
Symbolic (4)	CTRL-F4	CTRL-F4	TAB TAB 4
Half-word (5)	CTRL-F5	CTRL-F5	TAB TAB 5
Byte-pointer (6)	CTRL-F6	CTRL-F6	TAB TAB 6
ASCII (7)	CTRL-F7	CTRL-F7	TAB TAB 7
Floating point (8)	CTRL-F8	CTRL-F8	TAB TAB 8
System Call (9)	CTRL-F9	TAB TAB 9	TAB TAB 9
AOS/VS Error Message (0)	CTRL-F10	TAB TAB 0	TAB TAB 0

* F1 through F10 represent the function keys on the DASHER™ D200 terminal; they are numered from left to right (see Figure 8-1).

** In addition to the commands in this table, you can set the global display mode by entering a new value in the display mode register ($T).

Keyboards With Function Keys

Figure 8-1 shows the location of the function keys on the D2 and D200 keyboards. In the figure, we number the leftmost function keys F1 through F10, though they are blank on your keyboard.



SD-02307

Figure 8-1. Function Keys

The AOS/VS Debugger documentation package includes keyboard templates that fit over the DASHER function keypads. Figure 8-2 shows the keyboard templates over the DASHER D2 and D200 function keys. Note that the templates associate each function key with a particular display mode.

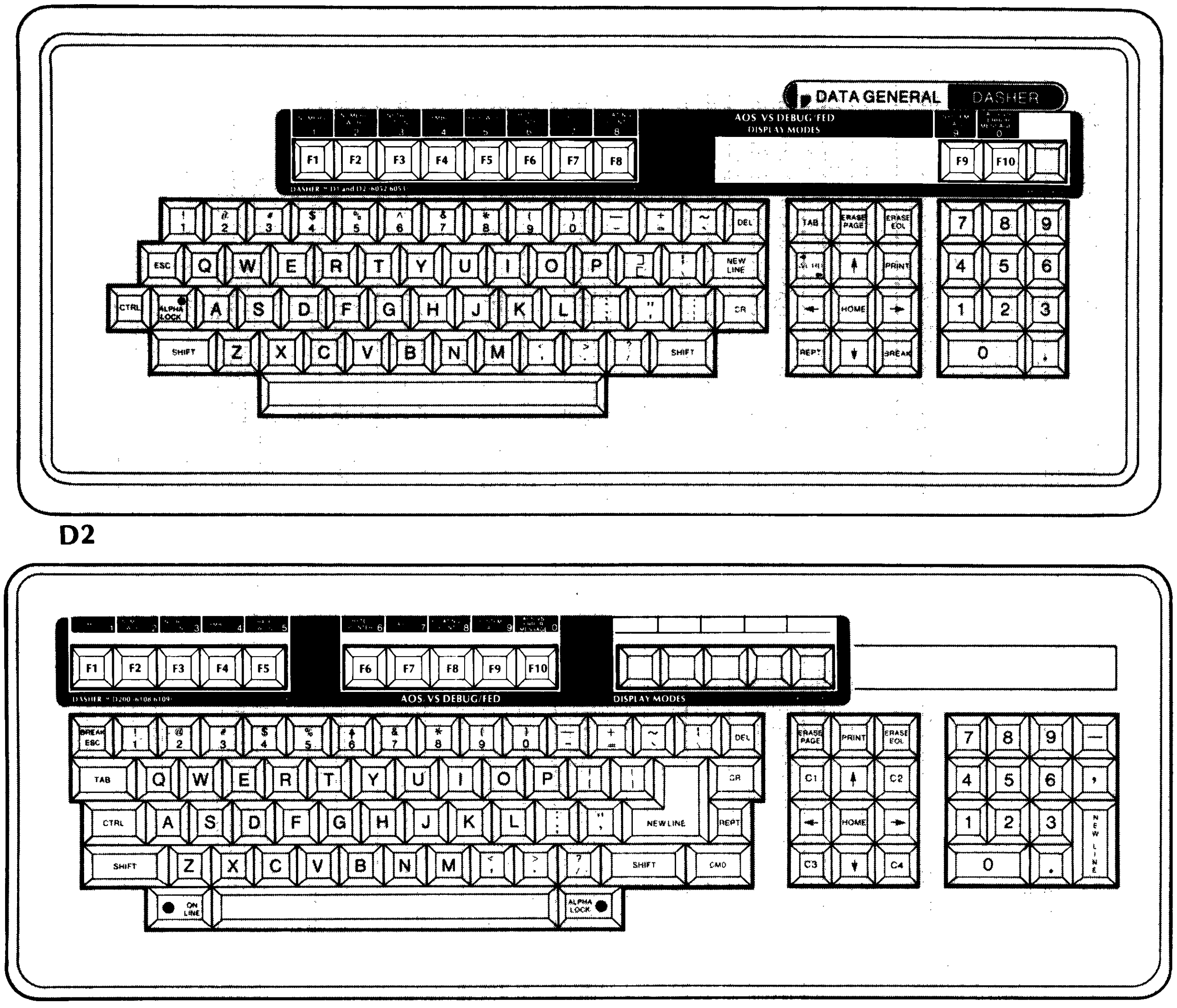


Figure 8-2. AOS/VS Debugger Keyboard Templates SD-02308

Using the function keys, you can easily alter the local and global display modes, as described in the following sections.

If you are using a DASHER D1 keyboard, you have only eight function keys (F1 through F8). When issuing display commands, you can use the function keys for display modes 1 through 8 (as on the D2 and D200 keyboards). However, to use the system call and AOS/VS error message modes (modes 9 and 0), you must either issue the display commands for keyboards that do not have function keys or modify the display mode register (see "Keyboards Without Function Keys" and "The Global Display Mode Register" later in this chapter). Generally, you will not use the system call and AOS/VS error message display modes during your debugging session. Thus, the missing function keys on the D1 keyboard should not cause any inconvenience.

Local Display Commands

You will use *local display commands* when you want to

- view the last value DEBUG presented in a new display mode
- obtain the value of a number, symbol, expression, or instruction in a particular display mode

To issue a local display command, simply press the function key that specifies the desired display mode. DEBUG will immediately reply with the appropriate value.

Local display mode commands do *not* affect the global display mode in any way.

In the following examples, we illustrate the local display mode commands. Note that DEBUG echoes each function key as two uparrow characters (↑↑).

In our first example, we examine two-word locations in memory. The global display mode is set to numeric (mode 1), but we want to view certain locations in other modes.

```
_500\ 2112220100 _↓
16000000502\ 22302200003 _↓
16000000504\ 20302200001 _↑↑ 101411 000001 _↑↑ XWLDA 0,1 _↓
16000000506\ 20426254063 _↓
16000000510\ 05051430253 _↓
16000000512\ 12424440520 _↑↑ 050 246 060 253 _↑↑ TRAP _↓
16000000514\ 05051230252 _↓
_
```

DEBUG presents the two-word values in the global display mode (i.e., numeric, by default). After DEBUG displays the value at location 504_8, we hit the numeric word function key (F2). DEBUG immediately presents the two-word value at location 504 as two 1-word octal integers (e.g., in numeric word mode). We then hit the instruction (F3) function key and DEBUG interprets the same value as an instruction.

Finally, we close that two-word location and continue stepping through memory. Note that the local display commands do not alter the global display mode; that is, DEBUG continues to present values in numeric mode.

At location 512_8, we again issue local display commands. First, we press the half-word function key (F5) and then the ASCII function key (F7). Each time, DEBUG presents the two-word value at location 512 in the specified display mode.

At the end of the above example, the global display mode is still numeric.

Our second set of examples show how to use local display commands to evaluate symbols, expressions, and instructions in various modes. Simply enter the symbol, expression, or instruction and press the appropriate function key. DEBUG evaluates the entry and presents that value in the specified mode.

```
_6*3+42↑↑ 000064 _↑↑ 000032 0 _

_LOC↑↑ 16000036000 _↑↑ 070000 036000 _↑↑ 160 000 074 000 _

_32.6E02↑↑ 10362740000 _↑↑ 103 313 300 000 _

_100;7↑↑ 16000000100 _

_XWLDA 0,1377,2↑↑ 30302201377 _

_LCALL 0,1,14777↑↑ 127311 21777777777 001377 _
```

On the first line, we enter the expression 6*3+42 and hit the numeric function key (F1). DEBUG immediately computes the expression's value and displays it in numeric mode. We then hit the byte-pointer function key (F6) and DEBUG presents the value in byte-pointer notation.

In this manner, you can use the local mode commands to view any number, symbol, expression, or instruction in any display mode. Again, local display commands do *not* alter the global display mode.

Global Display Commands

The *global display mode* is the display mode DEBUG is currently using to present data. When you enter a debugging session, the global display mode is set to numeric (mode 1). You can change the global mode by issuing a *global display command* as follows:

- hold down the control key (CTRL) and simultaneously press the appropriate function key (i.e., the function key that represents the desired display mode)

After issuing a global display command, DEBUG presents all data in that mode until you change the global mode again.

The following examples show how to use the global display mode commands. Note that DEBUG echoes each CTRL-function key command sequence as two uparrow characters (↑↑).

```
_500\ 00000000000 _↓                  By default, the global display mode is numeric (mode 1)
16000000502\ 00000000001 _↓
16000000504\ 00000000002 _↵
_↑↑_500\ 000 000 000 000 _↓           Change the global display mode to half-word (mode 5)
16000000502\ 000 000 000 001 _↓
16000000504\ 000 000 000 002 _↵
_↑↑_500\ 000000 000000 _↓             Change the global display mode to numeric mode (mode 2)
16000000502\ 000000 000001 _↓
16000000504\ 000000 000002 _↵
_
```

In this example, we examine the two-word values between locations 500 and 506 in three different display modes. To alter the global display mode, we held the CTRL key down and pressed the appropriate function key (DEBUG echoes these commands as ↑↑).

In the example above, on the fourth line, we changed the global mode to half-word by holding down the CTRL key and pressing the half-word function key (F5). DEBUG then presented all data in half-word mode until we issued another global display command.

As discussed in the previous section, you may override the global display mode at any time by issuing a local display command. Local display commands do *not* change the global mode.

Instead of issuing global display commands, you can alter the global display mode by modifying the display mode register. The last section of this chapter provides more information on how to do this.

Keyboards Without Function Keys

In the next two sections, we explain how to issue local and global display commands on keyboards that do *not* have function keys (i.e., blank keys at the top of the keyboard).

These commands are also valid for keyboards that include function keys. However, if you have function keys, you will usually use the commands described in the previous sections of this chapter instead.

Local Display Commands

Issue a *local display command* when you want to

- view the last value DEBUG presented in a new display mode
- obtain the value of a number, symbol, expression, or instruction in a particular display mode

Local display mode commands do *not* alter the global display mode.

To issue a local display command on a keyboard without function keys, enter

TAB n

where:

TAB	represents the TAB character (ASCII 11_8)
n	is a single digit between 0 and 9, inclusive. This digit identifies a particular display mode (see Table 8-1)

The numeric digit you enter in the n argument identifies a particular display mode. Table 8-1 lists the display modes and the corresponding numbers. Do *not* place a space between the TAB character and the digit n.

After you enter a TAB n command, DEBUG immediately presents the appropriate value. The following examples illustrate the use of the local display mode commands. Note that DEBUG echoes display commands as two uparrow characters (↑↑).

In our first example, we examine two-word locations in memory. The global display mode is set to numeric (mode 1), but we want to view the contents of certain locations in other modes.

```
_500\ 21122200100 _↓
16000000502\ 22302200003 _↓
16000000504\ 20302200001 _↑↑ 101411 000001 _↑↑ XWLDA 0,1 _↓
16000000506\ 20426254063 _↓
16000000510\ 05051430253 _↓
16000000512\ 12424440520 _↑↑ 050 246 060 253 _↑↑ TRAP _↓
16000000514\ 05051230252 _↓
_
```

DEBUG presents the two-word values in the global display mode (i.e., numeric, by default). After DEBUG display the value at location 504_8, we enter the command TAB 2. DEBUG immediately presents the two-word value at location 504 as two 1-word octal integers (i.e., in numeric word mode). We then enter TAB 3 (instruction mode) and DEBUG presents the same value as an instruction.

Finally, we close location 504 and continue stepping through memory. Note that the local display mode commands do not alter the global display mode; that is, DEBUG continues to present values in numeric mode.

At location 512_8, we again issue local display commands. First, we enter a half-word display command (TAB 5) and then an ASCII display command (TAB 7). Each time, DEBUG presents the two-word value at location 512 in the specified mode.

At the end of the above example, the global display mode is still numeric.

Our second set of examples show how to use local display commands to evaluate symbols, expressions, and instructions in various modes. Simply enter the symbol, expression, or instruction and enter the appropriate TAB n command. DEBUG evaluates the entry and presents that value in the specified mode.

```
_6+3*42↑↑ 000064 _↑↑ 00032 0 _

_LOC↑↑ 16000036000 _↑↑ 070000 036000 _↑↑ 160 000 074 000 _

_32.6E02↑↑ 10362740000 _↑↑ 103 313 300 000 _

_100;7↑↑ 16000000100 _

_XWLDA 0,1377,2↑↑ 30302201377 _

_LCALL 0,1,14777↑↑ 12731 21777777777 001377 _
```

On the first line, we enter the expression 6*3+42 and then hit TAB and the digit 1. DEBUG immediately computes the expression's value and displays it in numeric mode. We then enter TAB 6 and DEBUG presents the value in byte-pointer mode.

In this manner, you can use the local mode commands to view any number, symbol, expression, or instruction in any display mode. Again, local display commands do *not* alter the global display mode.

Global Display Commands

The *global display mode* is the display mode DEBUG is currently using to present data. When you enter a debugging session, the global display mode is set to numeric (mode 1). You can change the global mode by issuing a *global display command* as follows:

```
TAB TAB n
```

where:

TAB	represents the TAB character (ASCII 11_8)
n	is a single digit between 0 and 9, inclusive. This digit identifies a particular display mode (see Table 8-1)

According to this format, you change the global display mode by entering two TAB characters followed by a single digit. Do *not* place any spaces in the command.

The single digit n identifies one of the ten display modes. Table 8-1 lists the display modes and the corresponding numbers you should enter in the n argument.

After issuing a global display command, DEBUG presents all data in that mode until you change the global mode again.

The following examples show how to use the global display mode commands. Note that DEBUG echoes each TAB TAB n command as two uparrow characters (↑↑).

```
_500\ 00000000000 _↓                  By default, the global display mode is numeric (mode 1)
16000000502\ 00000000001 _↓
16000000504\ 00000000002 _↲
_↑↑_500\ 000 000 000 000 _↓           Change the global display mode to half-word (mode 5)
16000000502\ 000 000 000 001 _↓
16000000504\ 000 000 000 002 _↲
_↑↑_500\ 000000 000000 _↓             Change the global display mode to numeric word (mode 2)
16000000502\ 000000 000001 _↓
16000000504\ 000000 000002 _↲
_
```

In this example, we examine the two-word values between locations 500 and 506 in three different display modes. To alter the global display mode, we entered two consecutive TAB characters followed by a single numeric digit (DEBUG echoes each of these display commands as ↑↑).

In the example above, on the fourth line, we changed the global mode from numeric to half-word by typing TAB TAB 5. DEBUG presented all data in half-word mode until we issued another global display command.

As discussed in the previous section, you may override the global display mode at any time by issuing a local display command. Local display commands do *not* change the global display mode.

Instead of issuing global display commands, you can alter the global display mode by modifying the display mode register. The next section explains this.

The Global Display Mode Register

In addition to using global display commands, DEBUG allows you to change the global display mode by modifying the *display mode register.* The display mode register holds a value from 0 to 10_{10} (0 to 12_8) that indicates the current global mode.

Table 8-3 lists the values that DEBUG associates with the various display modes.

Table 8-3. Display Mode Register Values

Display Mode	Register Value	
	Decimal	Octal
Numeric (default mode)	1	1
Numeric word	2	2
Instruction	3	3
Symbolic	4	4
Half-word	5	5
Byte-pointer	6	6
ASCII	7	7
Floating point	8	10
System call	9	11
AOS/VS error message	0	0

To examine the display mode register, issue the command

```
$T
```

DEBUG will immediately display the register value. At this point, the register is *open* for modification; you may change the global display mode by entering a value from 0 to 9_{10} (0 to 11_8).

The following example shows how to use the $T commands:

```
_$T 00000000001 _↵
_500\ 00000000000 _↵
_$T 00000000001 _2↵
_500\ 000000 000000 _↵
_$T 000000 000002 _5↵
_500\ 000 000 000 000 _↵
_
```

The global mode equals 1 (numeric) so DEBUG presents the two-word value at address 500 as a single integer. Change the global mode to numeric word (mode 2) and again view the value at location 500. Lastly, change the global display mode to half-word (mode 5) and view the same location

There is no difference between modifying the display mode register and issuing a global display command (described earlier). Both instruct the Debugger to present data in a new global mode.

Refer to Chapter 4 for general information on examining and modifying registers. That chapter also contains a section specifically devoted to the display mode register.

End of Chapter

Chapter 9 Debugging 16-Bit Programs

The AOS/VS Debugger allows you to debug 16-bit programs. *16-bit programs* use 16-bit ECLIPSE instructions and are usually designed to run under Data General's 16-bit Advanced Operating System (AOS). By relinking them, you can execute 16-bit programs under the 32-bit AOS/VS operating system. Refer to the *AOS/VS Link and Library File Editor User's Manual* for information on linking 16-bit programs for use under AOS/VS.

In most cases, debugging a 16-bit program is identical to debugging a 32-bit program. However, you should be aware of certain exceptions and considerations. These pertain to the following:

- accumulators
- the stack
- symbolic displays
- overlays

The following sections explain these four subjects.

Accumulators

When debugging a 16-bit program, remember that 16-bit Arithmetic and Logic (ALC) instructions operate on the low-order (least significant) 16 bits of the fixed-point accumulators. That is, 16-bit ALC instructions only affect bits 16 through 31 of the 32-bit ECLIPSE MV/8000 accumulators.

Even though your instructions only access the last 16 bits of the accumulators, DEBUG always presents accumulators as full 32-bit values. Thus, when viewing an accumulator value, you will usually ignore the first 16 bits of DEBUG's display.

The Stack

The 32-bit ECLIPSE MV/8000 hardware provides four 32-bit stack registers for your use (see the $E command in Chapter 4). Data General's 16-bit computers do *not* have hardware stack registers, so 16-bit programs use absolute locations 40_8 through 43_8 to hold stack information. Thus, when examining stack parameters in a 16-bit program, you will usually reference locations 40 through 43_8, *not* the MV/8000 stack registers.

Symbolic Displays

In Chapter 6, we describe how DEBUG presents values in symbolic mode (i.e., as a symbol plus a numeric offset). In that discussion, we said that DEBUG only uses symbols whose values specify the correct ring when presenting addresses. This is *not* the case when you are debugging 16-bit programs.

When presenting symbolic values in 16-bit programs, DEBUG only uses the low-order (least significant) 16 bits of each symbol value. When DEBUG is searching for a symbol to match a particular value, it strips the symbol value to 16 bits before the comparison. Thus, DEBUG only uses bits 16 through 31 of the symbol's value when trying to find a match.

Refer to Chapter 6 for more information on symbols and DEBUG's symbolic displays.

Overlays

The AOS/VS Debugger supports 16-bit programs that include overlays. There are, however, certain restrictions:

- When using an overlay symbol as an address in a memory examine or display command, be sure the corresponding overlay is memory resident; if not, DEBUG returns an error
- When presenting addresses in symbolic mode, DEBUG only uses symbols that identify locations currently in memory
- Do *not* set breakpoints within movable resources or reference symbols that define addresses in movable resources

The first restriction on debugging overlayed programs involves the use of symbols that define addresses within an overlay (for example, labels in an overlay). The Debugger will always recognize symbols defined in your overlay, whether the overlay is in memory or not. Thus, you can always use overlay symbols to modify register values, as terms in an expression, etc. However, when using an overlay symbol as an address in a memory examine or display command, be sure that the corresponding overlay is memory resident. An example will help clarify this point.

Suppose you define an overlay area that starts at location 1000_8 and one overlay destined for that area defines label LOC at address 1050_8. DEBUG always recognizes symbol LOC, whether LOC's overlay is in memory or not. However, if you want to examine the value at address LOC, you must be sure that LOC's overlay is resident. If you enter the command

```
_LOC/
```

DEBUG opens location 1050_8 only if LOC's overlay is resident. That is, entering `LOC/` only shows you the value at address LOC if LOC's overlay is in memory. Otherwise, DEBUG returns an error.

The second overlay consideration pertains to DEBUG's presentation of addresses. As discussed in Chapter 6, DEBUG always tries to present addresses in symbolic mode (i.e., as a symbol plus a numeric offset). When presenting symbolic addresses, DEBUG only uses symbols that define locations currently in memory. In the previous example, DEBUG will only present address 1050_8 as LOC if LOC's overlay is in memory. If LOC's overlay is *not* resident, DEBUG will use a different symbol, if possible, or simply present the address as a numeric value.

Lastly, you can *not* set breakpoints within movable resources or reference symbols that define addresses within a movable resource. *Movable resources* are overlays that are position independent; that is, they need not reside at the same address each time your program calls them into memory.

Refer to the *AOS/VS Link and Library File Editor User's Manual* and the *AOS/VS Programmer's Manual* for more information on overlays.

End of Chapter

Chapter 10
The AOS/VS File Editor (FED)

The *AOS/VS File Editor (FED) utility* allows you to examine or modify locations in AOS/VS disk files. Using FED, you can inspect any kind of file; you are not limited to executable program files (as you are with DEBUG).

All changes you make to a disk file during the editing session are permanent. That is, FED copies all modifications into the disk file.

FED Commands

FED uses a subset of the DEBUG commands. When you edit a disk file with FED, that file is *not* being executed. Thus, all DEBUG commands that control breakpoints, program execution, and machine state registers are invalid under FED. More specifically, when using FED, you may *NOT*

- set, examine, or delete breakpoints ($B, $D)
- issue program restart or proceed commands ($P, $R), or examine proceed count registers ($Q)
- examine or modify the following machine state registers:
  - accumulators and carry ($A)
  - stack ($E)
  - floating point registers ($F)
  - program counter ($L)
  - processor status register ($V)
- use the special DEBUG symbols that represent the machine state registers (you may, however, use the location counter symbol .)

All other Debugger commands function under FED as they do under DEBUG. Table 10-1 lists and describes the FED commands.

Table 10-1. FED Commands

Command	Description	Reference
address/	Display the one-word value at word location **address**	Chapter 3
address\	Display the two-word value at word location **address**	Chapter 3
$C	Push to the CLI	Chapter 2
$G	Open the ring register	Chapters 3, 4
$H	Help command: list the various topics that FED can provide information about	Chapter 2
keyword$H	Help command: provide information about the topic identified by **keyword**	Chapter 2
$I	Display all currently defined temporary symbols	Chapter 6
symbol,value$I	Define a temporary symbol	Chapter 6
$J	Delete all temporary symbols	Chapter 6
symbol$J	Delete a specific temporary symbol	Chapter 6
$N	Open the output radix ring	Chapter 4
$S	Display or search a range of memory locations	Chapter 3
$T	Open the global display mode register	Chapters 4, 8
$X	Disable the current symbol table file	Chapter 6
file$X	Disable the current symbol table file (if any) and enable a new one	Chapter 6
$Y	Disable the current log file	Chapter 2
file$Y	Disable the current log file (if any) and enable a new one	Chapter 2
$Z	Terminate the FED editing session	Chapter 2
$?	Display a diagnostic error message for the last error	Chapter 2
;comment)	Enter the character string **comment** in the current log file	Chapter 2

In addition to the commands in Table 10-1, all local and global display mode commands and all Debugger expressions are valid under FED (see Chapter 8). Refer to the appropriate sections of this manual for complete descriptions of the various commands.

FED Operating Procedures

To execute the FED utility, enter the following CLI command:

```
) XEQ   FED [/switch]...   pathname
```

where:

XEQ	is the CLI command that executes a program (the single character X is a legal abbreviation for XEQ)
FED	is the name of the File Editor program (less the .PR extension)
/switch	is one or more optional FED command switches (see Table 10-2 below)
pathname	is the pathname of the file you want to edit

Table 10-2 lists the switches you can include on the FED command line. Table 10-2. FED Command Switches

Table 10-2. FED Command Switches

Switch	Description
/I=filename	Use the commands in *filename* for the editing session. With this switch, you can build a file of FED commands and apply them all at once when you execute FED (i.e., you can run FED in batch mode)
/L=filename	Use *filename* as the FED log file. That is, send all FED commands and responses to *filename*. If *filename* does not exist, FED will create it; otherwise, FED appends the new information to the existing one Note that the /L= switch performs the same operation as the file$Y command. You will normally use the */L=* switch in conjunction with the */I=* switch (i.e., when running FED in batch mode)
/N	Do not attempt to open a symbol table (.ST) file. (See Chapter 6 for information on symbol tables.)
/P	Treat the disk file as a program file (see "Program and User Data Files" below)
/R	Allow only read access
/S=filename	Use *filename* as the symbol table file. This switch performs the same operation as the file$X command. You will normally use this switch in conjunction with the */I=* switch (i.e., when running FED in batch mode). (See Chapter 6 for information on symbol tables.)
/U	Treat the disk file as a user data file (see "Program and User Data Files" below)
/X	Treat the disk file as an AOS/VS system file. Only use this switch if you are applying a patch released by Data General Corp. (see "AOS/VS System Files" below)

If you are executing FED interactively (i.e., without the /I= switch), FED will display the following prompt (same as DEBUG):

```
_
```

This means FED has successfully loaded your file and you may issue commands.

At the end of your editing session, issue the $Z command. FED will return control to the CLI.

If you specify a command file when invoking FED (with the /I= switch), you will *not* receive a FED prompt. Rather, FED opens your disk file and performs the operations specified in the command file. After editing your file, FED returns control to the CLI.

Your /I command file should contain FED commands in the normal format. To modify a location or register, enter the appropriate command followed by the new value. Then, close the location. For example, to place a new value at location 500_8, enter the following line in your command file:

_500/ *000000* _7) Open location 500 and deposit the value 7. Close with NEW LINE

FED returns control to the CLI when it encounters a $Z command or the end of your command file.

Program and User Data Files

When you enter a file editing session, FED automatically determines whether your file is a program file or a user data file. FED presents and interprets addresses differently depending on which file type you are editing.

A *program file* contains executable code and is of file type PRG or PRV (program filenames usually end with the .PR extension). AOS/VS program files contain a 20000_8 word *preamble* before the executable code. This preamble contains information that AOS/VS requires at program execution time. For example, the preamble indicates where in memory the various parts of your program will reside. Figure 10-1 shows how a program file appears on disk.

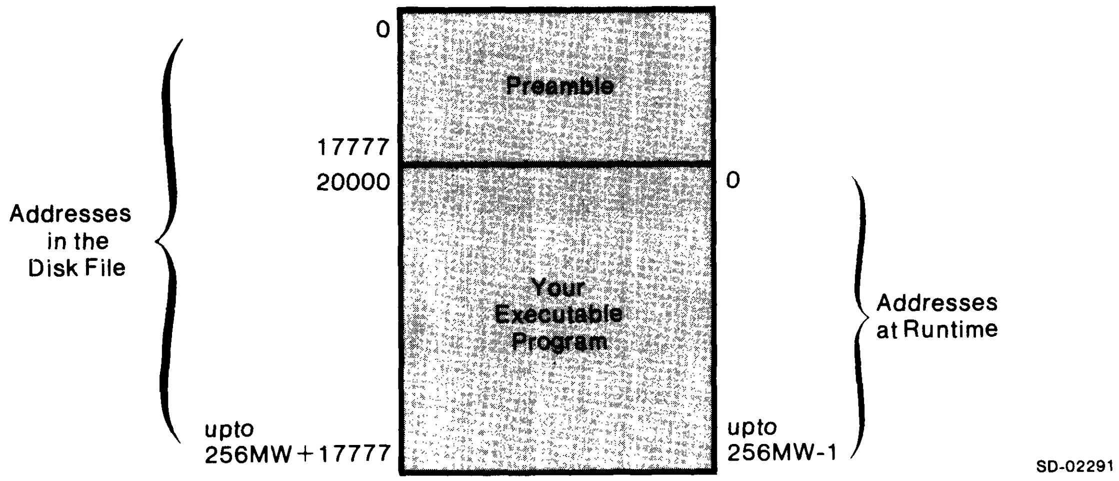


Figure 10-1. A Program File on Disk

When you execute your program file, the preamble is *not* loaded into your logical address space. Thus, for addressing purposes, location 0 is the first word *after* the preamble.

Similarly, when you edit a program file, FED interprets the first word after the preamble as location 0. In addition, FED presents shared and unshared addresses as they will appear at runtime. In short, FED calculates addresses as if you were executing the program file. This is the same addressing scheme that DEBUG uses.

If your file is *not* a program file, FED assumes it is a *user data file.* User data files do not require any special addressing considerations. That is, when editing a user data file, FED interprets location 0 as the very first word in the file. Addresses proceed in strict sequential order through the file.

Normally, FED automatically determines the file type you are editing. However, you may force FED to interpret your file as a program file or a user data file by issuing the /P or /U switches, respectively.

As a final note, when in program file (/P) mode, FED will search for the corresponding global symbol table (.ST) file and open it, if available. FED does *not* search for a symbol table when editing user data files. See the /N and /S= switch descriptions and Chapter 6 for more information on enabling and disabling symbol tables.

Addressing the Preamble

Usually, when you edit a program file, you do not want to examine locations in the preamble. However, if you wish, you may inspect the preamble by using one of the following techniques:

- issue the /U switch when editing the program file
- enter a -1 value in the ring register

If you use FED's /U switch when editing a program file, FED opens your file as if it was a user data file (see above discussion). Thus, location 0 represents the first word in the preamble, *not* the first word after the preamble.

Also, when you edit a program file using the /U switch, addresses proceed in strict sequential order through the file. Thus, there are no unused locations between the unshared and shared portions of your file. Rather, the shared locations begin immediately after the unshared locations.

As an alternative method for addressing the preamble, you can edit your file in program file (/P) mode and place the special ring value of -1 in the ring register. FED will then interpret all addresses that do not specify a ring as locations in the preamble.

Consider the following example:

_400/ *103000* _↵ _	Examine location 400_8. FED automatically inserts the value from the ring register in bits 1 through 3 of the address
_$G *00000000007* _-1↵ _	Open the ring register and change the default ring value to -1
_400/ *000000* _↵ _	FED now interprets address 400_8 as a location in your program file's preamble

In short, if the ring register contains -1, all addresses that do not explicitly include a ring value refer to the preamble.

When the ring register contains -1, your addresses must be between 0 and 17777_8, inclusive. FED returns an error for any address outside this range because the preamble is only 20000_8 words long.

Also, note the following rules regarding the -1 ring value:

- You may *not* use a -1 ring value with the ring field operator (;), only in the ring register
- You may *not* use a -1 ring value when using the DEBUG utility, only with FED

Refer to Chapter 3 for more information on rings; Chapter 4 describes the ring register and $G command.

AOS/VS System Files

In addition to program files and user data files, FED can edit *AOS/VS system files.* System files contain the AOS/VS operating system programs and are organized differently from your .PR program files.

The only time you will edit an AOS/VS system file is when applying a patch supplied by Data General Corp. A *patch* is a new section of code that replaces or modifies an existing portion of the system file. To apply a patch, use the /X and /I= switches as follows:

```
) XEQ    FED/X/I=patch-file    system-file
```

The /X switch informs FED that you are editing a system file (not a program file or user data file). The /I= switch directs FED to edit the system file according to the commands in patch-file. After this command, system-file will contain all the edits specified in the patch file.

Again, only edit AOS/VS system files when applying patches supplied by Data General Corp.

End of Chapter

Appendix A
DEBUG Error Messages

This appendix lists the various error messages that DEBUG may return. Refer to "Error Responses" in Chapter 2 for more information on DEBUG errors and error messages.

ALL BREAKPOINTS ALLOCATED

ASSEMBLY FORMAT ERROR

ASSEMBLY VALUE ERROR

ATTEMPT TO REFERENCE MORE THAN ONE OVERLAY IN AN EXPRESSION

ATTEMPT TO REFERENCE RELOCATABLE RESOURCE IN EXPRESSION

EXPRESSION FORMAT ERROR

EXPRESSION REFERENCE TO NON-RESIDENT LOCATION

ILLEGAL BREAK CHARACTER SEQUENCE

ILLEGAL MEMORY ADDRESS SPECIFIED

ILLEGAL OCTAL DIGIT IN NUMBER

INTERNAL CONSISTENCY ERROR

INVALID SYMBOL TABLE ADDRESS

LINE TOO LONG

MULTIPLE BREAKPOINTS AT THE SPECIFIED ADDRESS

NO OPEN LOCATION

NON-RESIDENT OVERLAY ADDRESS SPECIFIED

UNABLE TO PROCEED - MUST USE THE START COMMAND ($R)

UNDEFINED BREAKPOINT ENCOUNTERED

UNDEFINED SYMBOL

UNRECOGNIZED COMMAND FORMAT

VALUE OUT OF RANGE

WRITE ACCESS DENIED

End of Appendix

Appendix B
ASCII Character Set

To find the *octal* value of a character, locate the character, and combine the first two digits at the top of the character's column with the third digit in the far left column.

LEGEND:

Character code in decimal
EBCDIC equivalent hexadecimal code
Character

	10_	
0	64	@
	7C	

OCTAL	00_			01_			02_			03_			04_			05_			06_			07_		
0	0	00	NUL	8	16	BS (BACK-SPACE)	16	10	DLE ↑P	24	18	CAN ↑X	32	40	SPACE	40	4D	(	48	F0	0	56	F8	8
1	1	01	SOH ↑A	9	05	HT (TAB)	17	11	DC1 ↑Q	25	19	EM ↑Y	33	5A	!	41	5D	)	49	F1	1	57	F9	9
2	2	02	STX ↑B	10	15	NL (NEW LINE)	18	12	DC2 ↑R	26	3F	SUB ↑Z	34	7F	" (QUOTE)	42	5C	*	50	F2	2	58	7A	:
3	3	03	ETX ↑C	11	0B	VT (VERT TAB)	19	13	DC3 ↑S	27	27	ESC (ESCAPE)	35	7B	#	43	4E	+	51	F3	3	59	5E	;
4	4	37	EOT ↑D	12	0C	FF (FORM FEED)	20	3C	DC4 ↑T	28	1C	FS ↑\	36	5B	$	44	6B	, (COMMA)	52	F4	4	60	4C	<
5	5	2D	ENQ ↑E	13	0D	RT (RETURN)	21	3D	NAK ↑U	29	1D	GS ↑]	37	6C	%	45	60	-	53	F5	5	61	7E	=
6	6	2E	ACK ↑F	14	0E	SO ↑N	22	32	SYN ↑V	30	1E	RS ↑↑	38	50	&	46	4B	. (PERIOD)	54	F6	6	62	6E	>
7	7	2F	BEL ↑G	15	0F	SI ↑O	23	26	ETB ↑W	31	1F	US ↑←	39	7D	' (APOS)	47	61	/	55	F7	7	63	6F	?

OCTAL	10_			11_			12_			13_			14_			15_			16_			17_		
0	64	7C	@	72	C8	H	80	D7	P	88	E7	X	96	79	` (GRAVE)	104	88	h	112	97	p	120	A7	x
1	65	C1	A	73	C9	I	81	D8	Q	89	E8	Y	97	81	a	105	89	i	113	98	q	121	A8	y
2	66	C2	B	74	D1	J	82	D9	R	90	E9	Z	98	82	b	106	91	j	114	99	r	122	A9	z
3	67	C3	C	75	D2	K	83	E2	S	91	8D	[	99	83	c	107	92	k	115	A2	s	123	C0	{
4	68	C4	D	76	D3	L	84	E3	T	92	E0	\	100	84	d	108	93	l	116	A3	t	124	4F	|
5	69	C5	E	77	D4	M	85	E4	U	93	9D	]	101	85	e	109	94	m	117	A4	u	125	D0	}
6	70	C6	F	78	D5	N	86	E5	V	94	5F	↑ or ^	102	86	f	110	95	n	118	A5	v	126	A1	~ (TILDE)
7	71	C7	G	79	D6	O	87	E6	W	95	6D	← or _	103	87	g	111	96	o	119	A6	w	127	07	DEL (RUBOUT)

SD-00217 Character code in octal at top and left of charts.

↑ means CONTROL

End of Appendix

Appendix C
DEBUG Command Summary

Table A-1 is a command summary for the AOS/VS assembly language debugger (DEBUG). The commands are divided into logical categories. For each command, we provide a brief description of its purpose and a reference for additional information.

We use the following notation in the command summary:

Symbol	Meaning
$	The escape (ESC) character
address	An AOS/VS memory address
file	The complete pathname to a file
CTRL	The control key
f-key	Function key (i.e., a blank key at the top of the keyboard)
n	An integer value
symbol	The name of a symbol (see Chapter 6 for rules on legal symbol names)
TAB	The horizontal tab character

For a complete summary of the conventions and notation we use in this manual, please refer to the Preface.

For a summary of the available File Editor (FED) commands, refer to Table 10-1 (Chapter 10).

Command	Description	Reference
Memory Access Commands		
address/	Display the one-word value starting at word location **address** and open that word for modification	Chapter 3
address\	Display the two-word value starting at word location **address** and open those two words for modification	Chapter 3
) (NEW LINE)	Close the open location	Chapter 3
↓ (carriage return)	Close the open location and open the subsequent location	Chapter 3
↑ (uparrow)	Close the open location and open the previous location	Chapter 3
$S	Display or search a range of memory locations	Chapter 3
Commands that access MV/8000 machine state registers		
$A	Display the contents of the four fixed-point accumulators and the carry bit	Chapter 4
n$A	If 0<=n<=3, open fixed-point accumulator n; if n=4, open the carry bit	Chapter 4
$E	Display the four stack registers	Chapter 4
n$E	Open the stack pointer (n=0), frame pointer (n=1), stack limit (n=2), or stack base (n=3) register	Chapter 4
$F	Display the four floating point accumulators and the floating point status register (FPSR)	Chapter 4
n$F	If 0<=n<=3, open floating point accumulator n; if n=4, open the first 32 bits of the FPSR; if n=5, display the last 32 bits of the FPSR (i.e., the floating point PC)	Chapter 4
$L	Open the program counter (PC)	Chapters 3, 4
$V	Open the processor status register (PSR)	Chapter 4
Commands that access debugger registers		
$G	Open the ring register	Chapters 3, 4
$N	Open the output radix register	Chapter 4
n$Q	Open the proceed count register for breakpoint n	Chapters 4, 5
$T	Open the global display mode register	Chapters 4, 8
Commands that control program execution		
$B	Display all breakpoints	Chapter 5
address$B	Set a breakpoint at location address	Chapter 5
address,condition$B	Set a conditional breakpoint at location address	Chapter 5
$D	Delete all breakpoints	Chapter 5
n$D	Delete breakpoint n	Chapter 5
$P	Continue program execution at the last breakpoint encountered	Chapter 5
n$P	Continue program execution at the last breakpoint encountered and set the proceed count for that breakpoint to n	Chapter 5
n$Q	Open the proceed count register for breakpoint n	Chapters 4, 5
$R	Start or resume program execution at the current program counter (PC)	Chapters 2, 5
address$R	Start or resume program execution at location address	Chapters 2, 5
Commands related to symbol use		
$I	Display the currently defined temporary symbols	Chapter 6
symbol,value$I	Define a temporary symbol	Chapter 6
$J	Delete all temporary symbols	Chapter 6
symbol$J	Delete a specific temporary symbol	Chapter 6
$X	Disable the current symbol table file	Chapter 6
file$X	Disable the current symbol table file (if any) and enable a new one	Chapter 6
Display mode commands		
(f-key)	Redisplay the last value in the specified display mode (for consoles with function keys)	Chapter 8
CTRL-(f-key)	Modify the global display mode (for consoles with function keys)	Chapter 8
TAB-n	Redisplay the last value in the specified display mode (for consoles without function keys)	Chapter 8
TAB-TAB-n	Modify the global display mode (for consoles without function keys)	Chapter 8
General use commands		
$C	Push to the CLI	Chapter 2
$H	Help command: list the various topics that DEBUG can provide information about	Chapter 2
keyword$H	Help command: provide information about the topic identified by **keyword**	Chapter 2
$Y	Disable the current log file	Chapter 2
file$Y	Disable the current log file (if any) and enable a new one	Chapter 2
$Z	Terminate the DEBUG session	Chapter 2
$?	Display a diagnostic error message for the last error	Chapter 2
;comment)	Enter the character string comment in the current log file	Chapter 2

DEBUG Command Summary

Command	Description	Reference
Memory Access Commands		
address/	Display the one-word value starting at word location **address** and open that word for modification	Chapter 3
address\	Display the two-word value starting at word location **address** and open those two words for modification	Chapter 3
) (NEW LINE)	Close the open location	Chapter 3
↓ (carriage return)	Close the open location and open the subsequent location	Chapter 3
↑ (uparrow)	Close the open location and open the previous location	Chapter 3
$S	Display or search a range of memory locations	Chapter 3
Commands that access MV/8000 machine state registers		
$A	Display the contents of the four fixed-point accumulators and the carry bit	Chapter 4
n$A	If 0<=n<=3, open fixed-point accumulator n; if n=4, open the carry bit	Chapter 4
$E	Display the four stack registers	Chapter 4
n$E	Open the stack pointer (n=0), frame pointer (n=1), stack limit (n=2), or stack base (n=3) register	Chapter 4
$F	Display the four floating point accumulators and the floating point status register (FPSR)	Chapter 4
n$F	If 0<=n<=3, open floating point accumulator n; if n=4, open the first 32 bits of the FPSR; if n=5, display the last 32 bits of the FPSR (i.e., the floating point PC)	Chapter 4
$L	Open the program counter (PC)	Chapters 3, 4
$V	Open the processor status register (PSR)	Chapter 4
Commands that access debugger registers		
$G	Open the ring register	Chapters 3, 4
$N	Open the output radix register	Chapter 4
n$Q	Open the proceed count register for breakpoint n	Chapters 4, 5
$T	Open the global display mode register	Chapters 4, 8
Commands that control program execution		
$B	Display all breakpoints	Chapter 5
address$B	Set a breakpoint at location address	Chapter 5
address,condition$B	Set a conditional breakpoint at location address	Chapter 5
$D	Delete all breakpoints	Chapter 5
n$D	Delete breakpoint n	Chapter 5
$P	Continue program execution at the last breakpoint encountered	Chapter 5
n$P	Continue program execution at the last breakpoint encountered and set the proceed count for that breakpoint to n	Chapter 5
n$Q	Open the proceed count register for breakpoint n	Chapters 4, 5
$R	Start or resume program execution at the current program counter (PC)	Chapters 2, 5
address$R	Start or resume program execution at location address	Chapters 2, 5
Commands related to symbol use		
$I	Display the currently defined temporary symbols	Chapter 6
symbol,value$I	Define a temporary symbol	Chapter 6
$J	Delete all temporary symbols	Chapter 6
symbol$J	Delete a specific temporary symbol	Chapter 6
$X	Disable the current symbol table file	Chapter 6
file$X	Disable the current symbol table file (if any) and enable a new one	Chapter 6
Display mode commands		
(f-key)	Redisplay the last value in the specified display mode (for consoles with function keys)	Chapter 8
CTRL-(f-key)	Modify the global display mode (for consoles with function keys)	Chapter 8
TAB-n	Redisplay the last value in the specified display mode (for consoles without function keys)	Chapter 8
TAB-TAB-n	Modify the global display mode (for consoles without function keys)	Chapter 8
General use commands		
$C	Push to the CLI	Chapter 2
$H	Help command: list the various topics that DEBUG can provide information about	Chapter 2
keyword$H	Help command: provide information about the topic identified by keyword	Chapter 2
$Y	Disable the current log file	Chapter 2
file$Y	Disable the current log file (if any) and enable a new one	Chapter 2
$Z	Terminate the DEBUG session	Chapter 2
$?	Display a diagnostic error message for the last error	Chapter 2
;comment)	Enter the character string comment in the current log file	Chapter 2

DEBUG Command Summary

Index

Within this index, the letter "f" means "and the following page"; "ff" means "and the following pages". Also, primary references are listed first.

Data General

TIPS ORDER FORM
Technical Information & Publications Service

BILL TO:	SHIP TO: (if different)
COMPANY NAME ______	COMPANY NAME ______
ADDRESS ______	ADDRESS ______
CITY ______	CITY ______
STATE ______ ZIP ______	STATE ______ ZIP ______
ATTN: ______	ATTN: ______

QTY	MODEL #	DESCRIPTION	UNIT PRICE	LINE DISC	TOTAL PRICE

(Additional items can be included on second order form) [Minimum order is $50.00]

Tax Exempt # ______
or Sales Tax (if applicable)

TOTAL	
Sales Tax	
Shipping	
TOTAL	

METHOD OF PAYMENT

☐ Check or money order enclosed
For orders less than $100.00

☐ Charge my ☐ Visa ☐ MasterCard
Acc't No. ______ Expiration Date ______

☐ Purchase Order Number: ______

SHIP VIA

☐ DGC will select best way (U.P.S or Postal)

☐ Other:
☐ U.P.S. Blue Label
☐ Air Freight
☐ Other ______
______

NOTE: ORDERS LESS THAN $100, INCLUDE $5.00 FOR SHIPPING AND HANDLING.

Person to contact about this order ______ Phone ______ Extension ______

Mail Orders to:

Data General Corporation
Attn: Educational Services/TIPS F019
4400 Computer Drive
Westboro, MA 01580
Tel. (617) 366-8911 ext. 4032

______
Buyer's Authorized Signature Date
(agrees to terms & conditions on reverse side)

______
Title

______
DGC Sales Representative (If Known) Badge #

DISCOUNTS APPLY TO MAIL ORDERS ONLY

012-1780

educational services

CUT ALONG DOTTED LINE

DATA GENERAL CORPORATION TECHNICAL INFORMATION AND PUBLICATIONS SERVICE TERMS AND CONDITIONS

Data General Corporation ("DGC") provides its Technical Information and Publications Service (TIPS) solely in accordance with the following terms and conditions and more specifically to the Customer signing the Educational Services TIPS Order Form shown on the reverse hereof which is accepted by DGC.

1. **PRICES**
   Prices for DGC publications will be as stated in the Educational Services Literature Catalog in effect at the time DGC accepts Buyer's order or as specified on an authorized DGC quotation in force at the time of receipt by DGC of the Order Form shown on the reverse hereof. Prices are exclusive of all excise, sales, use or similar taxes and, therefore are subject to an increase equal in amount to any tax DGC may be required to collect or pay on the sale, license or delivery of the materials provided hereunder.

2. **PAYMENT**
   Terms are net cash on or prior to delivery except where satisfactory open account credit is established, in which case terms are net thirty (30) days from date of invoice.

3. **SHIPMENT**
   Shipment will be made F.O.B. Point of Origin. DGC normally ships either by UPS or U.S. Mail or other appropriate method depending upon weight, unless Customer designates a specific method and/or carrier on the Order Form. In any case, DGC assumes no liability with regard to loss, damage or delay during shipment.

4. **TERM**
   Upon execution by Buyer and acceptance by DGC, this agreement shall continue to remain in effect until terminated by either party upon thirty (30) days prior written notice. It is the intent of the parties to leave this Agreement in effect so that all subsequent orders for DGC publications will be governed by the terms and conditions of this Agreement.

5. **CUSTOMER CERTIFICATION**
   Customer hereby certifies that it is the owner or lessee of the DGC equipment and/or licensee/sub-licensee of the software which is the subject matter of the publication(s) ordered hereunder.

6. **DATA AND PROPRIETARY RIGHTS**
   Portions of the publications and materials supplied under this Agreement are proprietary and will be so marked. Customer shall abide by such markings. DGC retains for itself exclusively all proprietary rights (including manufacturing rights) in and to all designs, engineering details and other data pertaining to the products described in such publication. Licensed software materials are provided pursuant to the terms and conditions of the Program License Agreement (PLA) between the Customer and DGC and such PLA is made a part of and incorporated into this Agreement by reference. A copyright notice on any data by itself does not constitute or evidence a publication or public disclosure.

7. **DISCLAIMER OF WARRANTY**
   DGC MAKES NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY AND FITNESS FOR PARTICULAR PURPOSE ON ANY OF THE PUBLICATIONS SUPPLIED HEREUNDER.

8. **LIMITATIONS OF LIABILITY**
   IN NO EVENT SHALL DGC BE LIABLE FOR (I) ANY COSTS, DAMAGES OR EXPENSES ARISING OUT OF OR IN CONNECTION WITH ANY CLAIM BY ANY PERSON THAT USE OF THE PUBLICATION OF INFORMATION CONTAINED THEREIN INFRINGES ANY COPYRIGHT OR TRADE SECRET RIGHT OR (II) ANY INCIDENTIAL, SPECIAL, DIRECT OR CONSEQUENTIAL DAMAGES WHATSOEVER, INCLUDING BUT NOT LIMITED TO LOSS OF DATA, PROGRAMS OR LOST PROFITS.

9. **GENERAL**
   A valid contract binding upon DGC will come into being only at the time of DGC's acceptance of the referenced Educational Services Order Form. Such contract is governed by the laws of the Commonwealth of Massachusetts. Such contract is not assignable. These terms and conditions constitute the entire agreement between the parties with respect to the subject matter hereof and supersedes all prior oral or written communications, agreements and understandings. These terms and conditions shall prevail notwithstanding any different, conflicting or additional terms and conditions which may appear on any order submitted by Customer.

DISCOUNT SCHEDULES

DISCOUNTS APPLY TO MAIL ORDERS ONLY.

LINE ITEM DISCOUNT

5-14 manuals of the same part number - 20%
15 or more manuals of the same part number - 30%

DISCOUNTS APPLY TO PRICES SHOWN IN THE CURRENT TIPS CATALOG ONLY.

Data General

TIPS ORDERING PROCEDURE:

Technical literature may be ordered through the Customer Education Service's Technical Information and Publications Service (TIPS).

1. Turn to the TIPS Order Form.
2. Fill in the requested information. If you need more space to list the items you are ordering, use an additional form. Transfer the subtotal from any additional sheet to the space marked "subtotal" on the form.
3. Do not forget to include your MAIL ORDER ONLY discount. (See discount schedules on the back of the TIPS Order Form.)
4. Total your order. (MINIMUM ORDER/CHARGE after discounts of $50.00.)

   If your order totals less than 100.00, enclose a certified check or money order for the total (include sales tax, or your tax exempt number, if applicable) plus $5.00 for shipping and handling.
5. Please indicate on the Order Form if you have any special shipping requirements. Unless specified, orders are normally shipped U.P.S.
6. Read carefully the terms and conditions of the TIPS program on the reverse side of the Order Form.
7. Sign on the line provided on the form and enclose with payment. Mail to:

   TIPS
   Educational Services – M.S. F019
   Data General Corporation
   4400 Computer Drive
   Westboro, MA 01580
8. We'll take care of the rest!

educational services

Data General Corporation, Westboro, MA 01580

093-000246-01