DBG–11 Symbolic Debugger User's Guide

Order Number AA–HJ37C–TC

August 1991

This manual tells you how to use the DBG–11 symbolic debugging utility package. The manual describes DBG–11 commands and illustrates their use with a sample program. You should have some knowledge of assembly language programming before trying to use DBG–11.

DBG–11 is an unsupported product and subject to change without notice. The term unsupported as used in reference to a distributed module means that Digital does not guarantee that future releases will be necessarily compatible with previous versions of that module and does not guarantee that module will appear in future releases of the operating system. However, SPRs (Software Performance Reports) on the module are answered.

Revision/Update Information:	This manual supersedes AA–HJ37B–TC.
Operating Systems:	RT–11 Version 5.6
Software:	DBG–11 Version 2.0

Digital Equipment Corporation
Maynard, Massachusetts

First Printing, March 1986
Revised, August 1989
Revised, August 1991

S1521

This document was prepared using VAX DOCUMENT, Version 1.2.

Contents

Preface

Document Structure

Chapter 1 introduces the DBG–11 utility package by describing its components, features, and general requirements.

Chapter 2 describes the commands available in DBG–11.

Chapter 3 tells you how to install and use DBG–11 to debug a program. The chapter uses examples based on the sample program given in Appendix A.

Chapter 4 describes the DBGSYM symbol definition program available as part of DBG–11 under RT–11. DBGSYM lets you create a file handler of user symbol definitions that you can load and use from DBG–11.

Chapter 5 describes more advanced DBG–11 techniques that you may want to use if you debug a device handler. The chapter also covers some aspects of accessing extended memory with DBG–11 that you may find helpful in certain situations.

Appendix A contains a listing of the sample program used for illustration throughout the manual.

Appendix B lists all DBG–11 error messages.

Appendix C provides a cross-reference for all DBG–11 commands.

Conventions

The following conventions are used in this manual.

Convention	Meaning

Black print	In examples, black print indicates output lines or prompting characters that the system displays. For example:
	`.BACKUP/INITIALIZE DL0:F*.FOR DU1:WRK`
	`Mount output volume in DU1:; continue? Y`

Red print	In examples, red print indicates user input.
Braces ({ })	In command syntax examples, braces enclose options that are mutually exclusive. You can choose only one option from the group of options that appears in braces.

Convention	Meaning	
Brackets ([ ])	Square brackets in a format line represent optional parameters, qualifiers, or values.	
lowercase characters	In command syntax examples, lowercase characters represent elements of a command for which you supply a value. These include:	
	addr	An expression representing the address of a program location.
	n	An integer in the range 0 through 7.
	opt	Option identifier.
	qual	A DBG–11 command qualifier.
	symbol	An alphanumeric string of up to 6 characters, representing a 16-bit numeric value.
	val	A numeric or symbolic value, used alone or in an expression, with a maximum value of 177777_8. An expression may include arithmetic operators. If the value of an expression exceeds 177777_8, DBG–11 truncates the resulting value to the low-order 16 bits.
UPPERCASE characters	In command syntax examples, uppercase characters represent elements of a command that should be entered exactly as given.	
$\boxed{\text{RET}}$	$\boxed{\text{RET}}$ in examples represents the RETURN key. Unless the manual indicates otherwise, terminate all commands or command strings by pressing $\boxed{\text{RET}}$.	
$\boxed{\text{RETURN}}$	$\boxed{\text{RETURN}}$ in the text represents the RETURN key.	
$\boxed{\text{CTRL}/x}$	$\boxed{\text{CTRL}/x}$ indicates a control-key sequence. While pressing CTRL key, press another key. For example: $\boxed{\text{CTRL/C}}$	

Associated Documents

Basic Books

- *Introduction to RT–11*
- *Guide to RT–11 Documentation*
- *PDP–11 Keypad Editor User's Guide*
- *PDP–11 Keypad Editor Reference Card*

- *RT–11 Commands Manual*
- *RT–11 Mini-Reference Manual*
- *RT–11 Master Index*
- *RT–11 System Message Manual*
- *RT–11 System Release Notes*

Installation Specific Books

- *RT–11 Automatic Installation Guide*
- *RT–11 Installation Guide*
- *RT–11 System Generation Guide*

Programmer Oriented Books

- *RT–11 IND Control Files Manual*
- *RT–11 System Utilities Manual*
- *RT–11 System Macro Library Manual*
- *RT–11 System Subroutine Library Manual*
- *RT–11 System Internals Manual*
- *RT–11 Device Handlers Manual*
- *RT–11 Volume and File Formats Manual*

Hardware Requirements and Character Set

You can use DBG–11 on any terminal that supports hardware tab stops. DBG–11 does not simulate tabs; if you attempt to use DBG–11 on a terminal without hardware tab support, DBG–11's output display may be garbled.

You can use DBG–11 on a terminal without hardware tab stops if you use a version of DBG–11 that does I/O using the monitor terminal service, and the monitor terminal service simulates tabs.[1]

If you want to use DBG–11's hardware I/O mode (see Section 3.1), the terminal must be connected to your computer through a DL or DL-equivalent interface. DBG–11's software I/O mode has no interface restriction.

If you use DBG–11 on a video terminal that supports VT100 escape sequence processing, you can take advantage of DBG–11's graphics register display option.[2] If you use a video terminal that does not support VT100 escape sequence processing,

[1] SDS.SYS and SDSX.SYS use the monitor terminal service. Issue the DCL command SET TT NOTAB before using DBG–11.

[2] Issue the DCL command SET TT NOCRLF before enabling DBG–11's graphics register display.

you cannot take advantage of DBG–11's special video support, but you can still use the terminal to run DBG–11.

DBG–11 ignores the 8th bit when displaying characters, and uses the standard 7-bit ASCII character set in all cases. For example, DBG–11 displays both 101_8 and 301_8 as uppercase A.

Reading Path

You can use DBG–11 to debug programs under unmapped (SB and FB), single-mapped (XB and XM), and fully-mapped (ZB and ZM) monitors. The monitor mapping support determines which features, such as separate address space and processor modes, are enabled in a program. Therefore, the most useful reading path through this manual is determined by which type of monitor you are using with DBG–11.

The following table shows the most useful reading path determined by the monitor type being used.

Monitor Type	**Reading Path**
Unmapped	Read the entire manual except Chapter 5.
Single-Mapped	Read the entire manual. Parts of Chapter 5 dealing with separated I-D space and Supervisor mode will be informational only.
Fully Mapped	Read Chapters 1 through Section 3.2, and Chapter 5, then read from Section 3.3 to end of manual.

Chapter 1
Overview of DBG–11

1.1 Components

DBG–11 is a symbolic debugging package that lets you interactively debug an assembly-language program.

The components of DBG–11 are four variants of the SD pseudodevice handler (SDS.SYS, SDSX.SYS, SDH.SYS, and SDHX.SYS) and a symbol definition utility (DBGSYM.SAV). Although SDS.SYS, SDSX.SYS, SDH.SYS, and SDHX.SYS appear to be device handlers, they contain the program code to implement the DBG–11 commands and features. These files are discussed in more detail in Chapters 3 and 4.

1.2 Features

You can use DBG–11 to:

- Control program execution through breakpoint traps and single-step execution of machine instructions.
- Depending on hardware support and how the job was linked, change the displayed address space and processor mode.
- Display in numeric or mnemonic format the contents of memory locations and machine registers.
- Change the contents of memory locations and machine registers.
- Define symbol names for memory addresses.

Using DBG–11, you can execute your program gradually, setting breakpoints at selected locations or stepping through the program one PDP–11 instruction at a time.

DBG–11 gives a symbolic rather than a numeric display of machine instructions. If you use DBG–11 from a video terminal, DBG–11 can give you a continually updated display of register and stack contents.

You can examine any location in your program, such as instruction or data, word or byte, by opening the location with DBG–11. While the location is open, you can immediately change the contents. You can move forward or backward in memory to examine and modify other locations. Thus, you can test any number of modifications without rebuilding your program.

You can also define your own symbols. You can then use those symbols to refer by name to the address of a subroutine or data location, rather than using an octal address. For example, you can define the symbol START to refer to the starting address of the example program in Appendix A, and the symbol ANSWER to refer to the location where the result of the multiplication is stored.

NOTE

Any modifications you make to a program with DBG–11 are made to only the memory image of that program. That memory image is not written back to disk. Therefore, any actual changes you want to make in the program as a result of the debugging must be made to the program file image. As appropriate, you can modify the source file image and reassemble and relink or you can modify locations in the save file image by using SIPP.

1.3 Numeric Values

All numeric input to DBG–11 defaults to octal, unless the character string contains an 8 or 9, or a decimal point (.) follows the numeric value. In those cases DBG–11 assumes the value is decimal. All input to DBG–11 is independent of the output radix setting.

The following prefix operators let you enter hexadecimal, Radix–50, and ASCII character values:

_$

Up to four hexadecimal digits follow. (Loads one word.)

_%

Up to three Radix–50 characters follow. Use the underscore character (_) to enter leading or embedded spaces. (Loads one word.)

_"

Two ASCII characters follow. (Loads one word.)

_′

One ASCII character follows. (Loads one byte.)

You cannot enter a numeric value in binary (base 2) format. Convert the binary number to another base and enter the value in octal, decimal, or hexadecimal.

DBG–11 displays all numeric values based on the output radix setting (see description of the ;R command in Section 2.1), regardless of the radix used for input.

1.4 Registers and Symbols

DBG–11 can refer to registers and symbols as well as octal addresses.

Registers can be grouped into two classes:

- PDP–11 machine registers

  The machine registers hold the current contents of the user's PDP–11 general register set and processor status word. These registers are referred to symbolically as R0, R1, R2, R3, R4, R5, SP or R6, PC or R7, and PS.

- DBG–11 internal registers

  You can refer to the DBG–11 internal registers using DBG–11 commands, such as the open location commands described in Section 2.5.

  DBG–11's internal registers are:

  _ADROFF

  Maximum symbolic address offset. DBG–11 displays address values as *symbol+offset* until the offset value exceeds _ADROFF; at that point, DBG–11 displays addresses as purely numeric values.[1]

  _DATOFF

  Maximum symbolic data offset. DBG–11 displays data values as *symbol+offset* until the offset value exceeds _DATOFF; at that point, DBG–11 displays data values as purely numeric values.[1]

Symbols can be grouped into five classes:

- PDP–11 machine register symbols

  Machine register symbols consist of the eight general register symbols (R0, R1, R2, R3, R4, R5, SP, PC) and the processor status word symbol PS.

- DBG–11 permanent symbols

  DBG–11's permanent symbols are a subset of the PDP–11 instruction mnemonics defined in Appendix C of the *PDP–11 MACRO–11 Language Reference Manual*. DBG–11 does not currently support the PDP–11 instruction mnemonics for the CIS and FPP instruction sets.

- DBG–11 internal symbols

  Some DBG–11 internal symbols are used to examine or modify DBG–11 internal registers, while others represent values that can be used as arguments to some of DBG–11's commands. For example, you can use the internal symbol _ADROFF to refer to one of DBG–11's internal registers. The contents of _ADROFF determines the maximum symbolic offset DBG–11 displays from a user-defined symbol.

- Current displayed location and value symbols

  The special symbols period (.) and Q can be equated to the currently-displayed location and value, respectively. Those special symbols are described in Section 3.10.

[1] You can disable symbolic address and data display entirely, using the *qual*;T command.

- User-defined symbols

  You can use the colon (:) command to define or redefine your own set of symbols during a DBG–11 debugging session. The following rules govern the creation of user-defined symbols:

  – Symbols can be composed of only alphanumeric characters (A through Z, a through z, 0 through 9, dollar signs ($), and periods (.)).

  – The first character of a symbol must not be a number.

  – The first six characters of a symbol must be unique. A symbol can be written with more than six characters, but all characters after the sixth are ignored except to check that they are valid symbol characters. DBG–11 discards the extra characters.

  – Lowercase characters are converted to uppercase (DBG–11 is case insensitive).

1.5 Expressions

Once you define a symbol, you can use that symbol name, alone or with other numeric or symbolic values, to form an expression. An expression is a string of numbers, symbols, and operators that is interpreted as a number. For example, 3+6 is an expression; DBG–11 would interpret it as 11_8. You can use an expression to represent an absolute address, an offset, or a PDP–11 instruction.

An expression used in a DBG–11 session can contain any of the following elements:

- PDP–11 machine instructions.
- Numeric values.
- Symbolic values. You can use any user-defined symbol just as you would use its numeric equivalent.
- The current location symbol (.), described in Section 3.10.
- The value at the current displayed location symbol (Q), described in Section 3.10.
- The following binary arithmetic operators:

  – Plus sign (+), indicating addition.

  – Minus sign (–), indicating subtraction.

  – Asterisk (*), indicating multiplication.

  – Divide operator (_/), indicating division. The leading underscore differentiates it from the instruction mode examine operator.

  – Logical OR operator (!), indicating a logical OR operation.

  – Logical AND operator (&), indicating a logical AND operation.

- The following unary arithmetic operators:
  - Minus sign (–) indicates that the value(s) that follow are to be negated (2's complement).
  - Plus sign (+). Unary plus is ignored in expression evaluation, but it can be used to indicate that a value is positive.

In evaluating expressions, DBG–11 proceeds from left to right. All operators have the same precedence. You can nest expressions and force precedence within expressions by using angle brackets (< >). DBG–11 allows a maximum expression nesting depth of six. You can use the equal sign operator (=), described in Section 2.7, to display the value of an expression.

1.6 Numeric and Symbolic Addressing

You can use octal addresses to refer to locations within a program. However, it is often more convenient or meaningful to use symbolic addresses to refer to locations. When you use symbolic addressing, you refer to a location not by its numeric address but by an assigned name. You can combine symbolic references with other symbolic references or numeric values to form address expressions.

To use symbolic addressing with DBG–11, you must first define the symbols you want to use. Then, examine the program assembly listing and the program's map file created when you linked the program to determine the numeric addresses of the symbols you want to define. Use those values to define the symbols you have chosen with the DBG–11 symbol definition operator (:), described in Section 2.2.

Use of address expressions can make the symbol definition process easier and more meaningful. An address expression, represented throughout this manual by the lowercase word *addr*, is an expression interpreted by DBG–11 as a 16-bit numeric value. You can use an address expression to refer to a location in your program.

You can specify an address expression in either numeric or symbolic form. You can include in the address expression any of the operators and symbols described in Section 1.5.

The sample program loads beginning at location 1000 (you can determine that from the program's link map), but the listing for the program shows START at relative address 0 and VALUE1 at relative address 334. Instead of having to remember the 1000 offset all the time, you can define symbols like this:

```
1000,START:
START+102,MULPY:
START+124,PUTNUM:
START+334,VALUE1:
```

You can also define symbols using more complex expressions. The examples in Table 1–1 show how DBG–11 interprets various forms of address expressions. These examples assume a value of 1000 for the symbol START and a value of 1174 for the symbol ANSWER.

Table 1–1: Address Expression in DBG–11

Expression	DBG–11 Interpretation
START	001000
ANSWER	001174
START+5	001005
–<START+5>	176773
<ANSWER–START>_/2	000076

1.7 Processor Mode and Address Space Prompt Displays

DBG–11 displays the current processor mode and address space when running under mapped monitors. The display is omitted under unmapped monitors (SB and FB), as those monitors run only in Kernel mode Instruction space.

Under mapped monitors, the display shows the current hardware support, which can include the processor mode and current address space. For example, the display could show (U) for User mode on hardware that does not support separated I-D space. For hardware that supports separated address spaces, the display couples the current processor mode with the current address space, such as (UI) for User processor mode and Instruction address space.

Examples of various prompt displays are located throughout the manual.

Chapter 2
DBG–11 Commands

DBG–11 commands consist of one or more characters plus one or more optional arguments. When the DBG–11 command requires multiple arguments, use commas to separate them. The maximum number of arguments in any DBG–11 command is three.

DBG–11 commands can be put in groups according to function. DBG–11 includes commands that:

- Set DBG–11 operating parameters.
- Define and delete user symbols.
- Set and remove breakpoints.
- Execute the program.
- Open and close locations and display contents.
- Change the contents of locations.
- Display expression values.

Each group is discussed in detail in the following sections.

2.1 Setting DBG–11 Operating Parameters

[qual];M

Change address space/processor mode display.

DBG-11 lets you temporarily change the current address space bits in MMR3 and processor mode bits in the PSW. DBG–11 restores the initial setting automatically before beginning execution of a proceed command. You can manually restore the initial setting by specifying the command without condition.

Qualifiers to the ;M command must be specified singly. That is, do not use the construction _D!_U;M, but rather issue _D;M and then _U;M.

If *qual* is not specified, DBG–11 restores initial address space/processor mode display. If you specify *qual*, it can be one of the following:

_D

Temporarily set address space display to Data.

_I

Temporarily set address space display to Instruction (the default setting).

_K

Temporarily set processor mode display to Kernel.

_S

Temporarily set processor mode display to Supervisor.

_U

Temporarily set processor mode display to User.

[qual];R

Set/Reset output radix. If *qual* is not specified, the output radix for DBG–11 is set to octal. If you specify *qual*, it can be one of the following:

_BIN

Set output radix to 2.

_OCT

Set output radix to 8 (default).

_DEC

Set output radix to 10.

_HEX

Set output radix to 16.

The output radix is the same in both Instruction and Data address spaces.

If you are using the graphics register display (see Section 3.3.5) and set the output radix to _OCT, _HEX, or _DEC, the graphics register display will show values in the specified radix. If you set the output radix to _BIN, the graphics register display will default to octal, since there is not enough space in the graphics register display to show the values in binary. However, the instruction and location examine displays will be shown in binary.

The output radix setting has no effect on the way DBG–11 interprets numeric input. DBG–11 assumes all numeric input is octal unless otherwise instructed, as described in Section 1.3.

[qual];T

Set/Reset the following debugger functions:

- Escape key as single-step operator
- Processor mode
- Register (PC or PS) display
- Graphics register display
- Symbolic data display
- Symbolic address display

- Scope/Hard-copy rubout support

If you specify *qual*, it can be one of the following:

_ESC or _NOESC
Enable/disable support for ESCAPE key (ESC) as single-step operator (like ;S).

_K or _NOK
Enable/disable selection of Kernel mode (with ~ operator).

_S or _NOS
Enable/disable selection of Supervisor mode (with ~ operator).

_U or _NOU
Enable/disable selection of User mode (with ~ operator).

_PC or _PS
_PC sets the graphics register display such that the program counter (PC) is displayed in the lower left corner of the graphics register display (the initial setting). _PS sets the graphics register display such that the processor status (PS) word is displayed in the lower left corner.

_REG or _NOREG
Enable/disable graphics register display. Refer to Section 3.3.5 for details.

_SYM or _NOSYM
Enable/disable symbolic data display.

_ADR or _NOADR
Enable/disable symbolic address display.

_RUB or _NORUB
Enable/disable video mode rubout support. Video mode rubout (_RUB) backspaces and erases the previous character from the screen when you press the RUBOUT key. Hard-copy mode rubout (_NORUB) echoes a backslash (\ ) followed by the character or characters deleted, then a closing backslash.

The qualifiers _K, _S, _U, _PC, and _PS should be used singly in a command and not coupled with each other or other qualifiers.

If you omit *qual* before ;T, DBG–11 sets all functions to their defaults. By default (SET TT SCOPE), the functions are set to _NOESC, _K, _S, _U, _PC, _NOREG, _SYM, _ADR, and _RUB. If you have issued the command SET TT NOSCOPE, the functions are the same except _NORUB replaces _RUB.

[qual];V

Enable/Disable trap handling.

DBG–11 can intercept five types of traps: traps to 4, traps to 10, traps caused by memory protection violations, TRAP instructions, and IOT instructions. By default, DBG–11 intercepts any of those traps and notifies you by displaying a message, as shown in Table 2–1.

Table 2–1: Trap Interception by DBG–11

Type of Trap	Message Displayed
Trap to 4	T4: addr
Trap to 10	T10: addr
Memory protect	MP: addr
TRAP instruction	TR: addr
IOT instruction	IO: addr

The [*qual*];V command modifies the way DBG–11 handles traps though vectors 4 and 10; *qual* can be _T4 or _T10. If you specify a qualifier (_T4 or _T10), DBG–11 attempts to pass any trap though 4 or 10 to the trap-handling routine assigned by the RT–11 .TRPSET directive. If you have issued a _T4;V or _T10;V command but have not assigned a trap-handling routine with the .TRPSET directive, DBG–11 displays its normal trap messages when traps to 4 or 10 occur.

Specify ;V with no qualifier to reenable handling by DBG–11 of traps to 4 and 10.

Your program can intercept traps caused by IOT and TRAP instructions by filling in vector locations 20 and 34, respectively.

DBG–11 does not intercept EMT traps. EMT instructions execute normally, calling the monitor's EMT processing routine. If an error occurs while the monitor is processing an EMT, the monitor may stop the current job and exit to command level without returning control to DBG–11. If you want to debug a routine called by an EMT, you have to put a breakpoint within the routine itself; you cannot single-step through the EMT instruction to get to the routine.[1]

2.2 Defining and Deleting User Symbols

Because DBG–11 uses a single symbol definition file, each symbol is defined (or deleted) in both Instruction and Data address spaces.

[addr,]symbol:

Symbol definition operator. DBG–11 defines or redefines the specified symbol to be equal to the value *addr*. If *addr* is value 0, the symbol is defined as zero but the symbol is not used as the base to an offset in address or value displays.

If you omit *addr*, the value of the symbol is set equal to the value of the current location symbol (.), described in Section 3.10. This command closes any open location, defines the symbol, and prompts for the next command.

[1] If you are using a hardware I/O version of DBG–11 (SDH.SYS or SDHX.SYS), you can set bit 4 (020) in kernel address location 32. This sets the T-bit when the processor executes an EMT instruction, which will allow you to trace the execution of an EMT.

Any symbol that you define remains defined until you delete it using the ;K command or unload the SD handler. You can define a maximum of 20_{10} symbols.

symbol;K

Deletes the definition of the specified symbol from DBG–11's user symbol table.

2.3 Setting, Removing, and Displaying Breakpoints

All breakpoint operations are performed in I-space.

[addr][,n];B

Set/Reset breakpoints. This command has the following forms:

;B
Removes all breakpoints from the user program. (0;B is equivalent.)

,n;B
Removes breakpoint *n* from the user program, where *n* can be 1 through 10_8.

addr[,n];B
Sets breakpoint *n*, where *n* can be 1 through 10_8, in the user program at address *addr*. If *n* is omitted, DBG–11 uses the lowest-numbered available breakpoint. You can have up to 8 breakpoints set in your program at a time, numbered 1 through 10_8.

;D
Displays a table of the breakpoints.

2.4 Executing the Program

[addr];G

Sets BPT instructions at all breakpoints that have been set in your program, restores the processor status word and user program registers, and starts your program at the specified address or at the current saved PC value if *addr* is omitted.

Provided the monitor has not been damaged, DBG–11 exits to the monitor prompt (.) if you specify an address of 0 (0;G).

[val];P

Proceeds with user program execution from the current breakpoint location and stops when the next breakpoint location is encountered. (Program must be at the breakpoint location when issued.)

If you specify *val*, DBG–11 proceeds with program execution from the current breakpoint location and stops at that breakpoint again only after encountering it a total of *val* times. The default value for *val* is 1. Other breakpoints that may be set are not counted or affected; the program continues to stop when it encounters other set breakpoints.

You can set a different *val* count for each active breakpoint. For example, when you stop at breakpoint 1 you can type 10;P, and when you stop at breakpoint 2 you can type 25;P.

[val];S

Single step. Executes one instruction at the current PC value, and displays the address and the next symbolic PDP–11 instruction to be executed. Breakpoints are not recognized or displayed during single-step operations. If *val* is specified, DBG–11 executes *val* instructions, displaying each address and symbolic PDP–11 instruction as it is executed.

Setting the operating parameter _ESC;T lets you use the ESCAPE key to single-step each instruction.

2.5 Opening Locations and Displaying Contents

Locations are opened and contents displayed in the current address space and processor mode.

[addr]"

Word mode ASCII operator. Interprets and displays the contents of the currently open (or last opened) location as two ASCII characters. If the contents of a byte is a nonprinting character (octal range 000 through 037), DBG–11 displays the character as ^X, where X is determined by adding 100_8 to the contents of the byte. For example, 003 displays as ^C, since 003 + 100 equals 103, the ASCII code for C. The nonprinting ASCII code 177_8 is treated as a special case and displayed as ^?.

If " is preceded by *addr*, the value *addr* is taken as the address of a word location to be opened and displayed.

[addr]'

Forced byte mode ASCII operator. Interprets and displays the contents of the currently open (or last opened) location as one ASCII character.

If the contents of a byte is a nonprinting character (octal range 000 through 037), DBG–11 displays the character as ^X, where X is determined by adding 100_8 to the contents of the byte. For example, 003 displays as C, since 003 + 100 equals 103, the ASCII code for C. The nonprinting ASCII code 177_8 is treated as a special case and displayed as ^?.

If ' is preceded by *addr*, the value *addr* is taken as the address of a byte location to be opened and displayed.

[addr]%

Word mode Radix–50 operator. Interprets and displays the contents of the currently open (or last opened) location as three Radix–50 characters. If the location does not contain a valid Radix–50 string, DBG–11 displays three question marks (???).

If % is preceded by *addr*, the value *addr* is taken as the address of a location to be opened and displayed.

[addr]/

Instruction mode operator. Displays the contents of the currently open (or last opened) location as a symbolic PDP–11 instruction. If the location does not contain a PDP–11 instruction that DBG–11 recognizes, DBG–11 displays the contents as a numeric value, using the current default output radix as set by the [qual];R command.

If / is preceded by *addr*, the value *addr* is taken as the address of a word location to be opened and displayed.

If you attempt to open a DBG–11 internal register using the / instruction mode operator, DBG–11 defaults to symbolic data mode.

[addr][

Symbolic data mode operator. Displays the contents of the currently open (or last opened) location as a symbolic data word.

If [is preceded by *addr*, the value *addr* is taken as the address of a word location to be opened and displayed.

[addr] |

Numeric data mode operator. Displays the contents of the currently open (or last opened) location as a numeric data word, using the current output radix.

If | is preceded by *addr*, the value *addr* is taken as the address of a word location to be opened and displayed.

[addr]\

Forced byte mode operator. Displays the contents of the currently open (or last opened) location as a byte using the current output radix and redefines Q, the last displayed value symbol, as the contents of this byte.

If \ is preceded by *addr*, the value *addr* is taken as the address of a byte location to be opened and displayed.

2.6 Changing the Contents of Locations

Before you use any of the commands described in this section, you must open a memory location using one of the open location commands described in Section 2.5.

[val]RET

Closes the currently open location (if any) and prompts for the next command. If RET is preceded by *val*, the value *val* replaces the contents of the currently open location before it is closed. If *val* is specified and no location is currently open, DBG–11 displays the error message *?DBG–W–No location open*.

[val]LF

Closes the currently open location (if any), opens the next sequential location (word, byte, or instruction depending on the output mode in effect) using the current memory mapping, and displays its contents. If LF is preceded by *val*, the value *val* replaces the contents of the currently open location before it is closed.

[val]^

Closes the currently open location (if any), opens the immediately preceding location (word, byte, or instruction depending on the output mode in effect) using the current memory mapping, and displays its contents. If ^ is preceded by *val*, the value *val* replaces the contents of the currently open location before it is closed.

If you open location 0 and type ^, DBG–11 does not wrap around memory to location 177776; the current location remains at location 0.

Backing up while in instruction mode may occasionally give incorrect results because certain combinations are ambiguous.

[val]@

Closes the currently open location (if any), then uses the contents of the closed location as the address to open in I-space. Selects I-space as the current space and opens that location. If @ is preceded by *val*, the value *val* replaces the contents of the currently open location before it is closed.

[val]#

Closes the currently open location (if any), then uses the contents of the closed location as the address to open in D-space. Selects D-space as the current space and opens that location. If separated I-D space is not enabled or the hardware does not support address separation, the command functions like [val]@.

If # is preceded by *val*, the value *val* replaces the contents of the currently open location before it is closed.

[val]~

The ~ (tilde) operator is a rotating toggle between processor modes. The rotation is governed by the PDP–11 architecture and is Kernel to Supervisor to User. Initially, all three modes can be toggled. Valid modes can be disabled or enabled by the _[NO]K;T, _[NO]S;T, and _[NO]U;T commands.

Assuming all modes are valid, if currently in Kernel mode, closes the current location (if any) and opens the same location (if any) in Supervisor mode. If currently in Supervisor mode, closes the current location (if any) and opens the same location (if any) in User mode. If currently in User mode, closes the current location (if any) and opens the same location (if any) in Kernel mode.

If ~ is preceded by *val*, opens the specified location in the other mode.

[val]‘

The ‘ (grave accent) operator is a toggle between address spaces. If separated address space is not enabled or not supported by hardware, it terminates the current line and closes any open location.

Assuming separated address space is valid, if currently in D-space, closes the current location (if any) and opens the same location (if any) in I-space. If currently in I-space, closes the current location (if any) and opens the same location (if any) in D-space.

If ‘ is preceded by *val*, opens the specified location in the other address space.

2.7 Displaying Expression Values

[val]=

Interprets and displays the numeric value of expression *val* using the current output radix. If *val* is not specified, the value of the last displayed value symbol (Q) is displayed.

2.8 Special Characters

DBG–11 recognizes some characters that perform editing functions or control terminal output. These include:

`RUBOUT`
Deletes the last character typed.

`CTRL/C`
Ignored while using DBG–11; type 0;G to return to monitor, provided the monitor has not been damaged.

`CTRL/Q` ***(XON)***
Reenables terminal output after output was suspended with `CTRL/S` (XOFF).

`CTRL/S` ***(XOFF)***
Suspends output to the terminal until a `CTRL/Q` (XON) character is received.

CTRL/U

Causes DBG–11 to ignore the current command line, closes any open location, and prompts for the next command.

CTRL/W

Refreshes the graphics register display, closes any open location, and prompts for the next command. If the graphics register display is not enabled, this command acts like a CTRL/U.

Chapter 3
Using DBG–11 to Debug a Program

This chapter uses the sample program in Appendix A to illustrate the use of most of the DBG–11 commands described in Chapter 2. The examples are not comprehensive or complete, but they should give you enough information to let you expand your use of DBG–11 to more complex debugging tasks. If you have difficulty following any command explanation, assemble and link the sample program (with a link map), run it with DBG–11, and try typing the example exactly as given; that may help you to understand more easily what the command does.

Most of the examples assume you have defined the user symbols START, MULPY, PUTNUM, VALUE1, and RESULT, as described in Section 3.4. If you try the examples on your computer but have not defined those symbols, or if you have defined additional user symbols or changed the contents of locations, the offset values or contents of locations that DBG–11 displays may be different from those shown in the examples. However, the results should be equivalent.

Some of the examples assume that only certain symbols are defined and that the symbols have particular values. Those examples are clearly identified in the text preceding them.

All the examples in this chapter, except where indicated, were produced using DBG–11 with a ZM monitor and the example program running in User mode. If you are not using a fully-mapped (ZB or ZM) monitor, the prefix symbols will be different. However, the debugging process is the same. Refer to the end of Section 3.2 for more details.

3.1 The Parts of DBG–11

DBG–11 includes four variants of an SD pseudodevice handler and the symbol definition utility DBGSYM.SAV described in Chapter 4. The four device handlers supplied are:

SDS.SYS
Software I/O version for use with unmapped monitors. SDS.SYS does all its terminal input and output by using .TTYIN, .TTYOUT, and .PRINT requests.

SDH.SYS
Hardware I/O version for use with unmapped monitors. SDH.SYS does all its terminal input and output by directly accessing the console terminal CSR.

SDSX.SYS
Equivalent to SDS.SYS, for use with mapped monitors.

SDHX.SYS

Equivalent to SDH.SYS, for use with mapped monitors.

The hardware I/O versions of DBG–11 (SDH.SYS and SDHX.SYS) are suitable for most debugging jobs. Use the software I/O versions of DBG–11 (SDS.SYS and SDSX.SYS) only if you cannot use the hardware I/O versions to debug your particular application.

3.2 Getting Started

Perform one or more of the following steps. If the SD handler is already on your system device, steps 1 through 3 may not be necessary.

1. Copy the appropriate version of DBG–11 (SDH.SYS, SDS.SYS, SDHX.SYS, or SDSX.SYS) to SD.SYS or SDX.SYS on your system volume so the monitor will recognize the file as a device handler. Use SDH.SYS or SDS.SYS with the unmapped monitors; SDHX.SYS or SDSX.SYS with the mapped monitors. It is best to copy rather than rename the file so that you always retain an original copy of each of the four different files.
2. Issue SET commands to configure the SD handler to match your system. You will probably need to type the SET SD SYSGEN command. If you want to change the default settings of _ADROFF and _DATOFF, you can change those through SET commands as well.

   Table 3–1 lists the SET commands for the SD handler and briefly describes their function.
3. Install the SD handler using the INSTALL command. If you were already using a different version of the SD handler, you will need to use the REMOVE command to remove the old handler before you install the new one.
4. Assemble and link the application program that you want to debug. An example application is located in Appendix A that assembles and links under all distributed RT–11 monitors. Assembling and linking the example program, DBGDEM.MAC, is described further in this section. Be sure to specify a link map to determine symbol locations.
5. Place a BPT (breakpoint) instruction at the starting address of your program by using SIPP or by including the BPT instruction in your program source and assembling it in.

   If you want to put a BPT instruction at the starting address of a .SAV file using SIPP, first use SIPP to examine location 40 of block 0 of the .SAV file. That value is the starting address of the file. Next, use SIPP to look at the starting address location. Write down the contents of the starting location so you can remember the value later; then replace that value with a BPT instruction (000003).
6. Load the SD handler into memory using the LOAD command. The size of SD in low memory is determined by your monitor. For mapped systems, SD occupies somewhat more than 1K of low memory. In unmapped systems, SD is over 4K words long, so if your program is large you may need to unload other jobs or

handlers to make room for DBG–11. The SD handler must be named SD; you cannot rename the SD handler and load it under another name.

When you load the SD handler, it prints a version number and some option information in parentheses, in the form

```
DBG Vxx.xx - RT-11  (  opt  opt  opt  )
```

Table 3–2 lists the possible options that may appear.

7. Run the program you want to debug, using the R, RUN, FRUN, SRUN, V, or VRUN command. If you used SIPP to put a BPT instruction at the starting location of the program, use DBG–11 to change that location back to its original value before proceeding.

Table 3–1: SET Commands to Configure DBG–11 (SD Handler)

SET Command	Function
SET SD ADROFF=*val*	Sets default value for DBG–11 internal symbol _ADROFF. If you do not use this command, the default value for _ADROFF is 1000_8. This option changes the disk-resident copy of SD, but has no effect on any copy of SD already in memory. The new default value remains in effect until you issue another SET SD ADROFF command or issue the _ADROFF[val] from within DBG–11.
SET SD BREAK	Executes a BPT instruction and transfers control to DBG–11. Unlike the other SET commands, SET SD BREAK does not configure or change SD in any way; SET SD BREAK just gives you a way to get into DBG–11. You must load the SD handler before typing this SET command. Refer to Section 5.1 for a discussion of using SET SD BREAK to debug a device handler.
SET SD CSR=*val*	Sets CSR for terminal input and output. This command applies only to SDH.SYS and SDHX.SYS, the hardware I/O versions of the SD handler. Use this command if you want to use a terminal with a nonstandard CSR as the debugging terminal. This SET condition changes the disk-resident copy of SD, but has no effect on any copy of SD already in memory. The new value remains in effect until you issue another SET SD CSR command. DBG–11 assumes the following default CSR address for hardware I/O mode: CSR = 777560 That value is correct for standard PDP–11 console terminals and for the debugging (printer) port on CTI Bus-based computers. Any terminal to be used by DBG–11 for hardware I/O mode must be connected to a DL or DL-equivalent interface.

Table 3–1 (Cont.): SET Commands to Configure DBG–11 (SD Handler)

SET Command	Function
SET SD DATOFF=*val*	Sets default value for DBG–11 internal symbol _DATOFF. If you do not use this command, the default value for _DATOFF is 100_8. This SET condition changes the disk-resident copy of SD, but has no effect on any copy of SD already in memory. The new value remains in effect until you issue another SET SD DATOFF command or issue the _DATOFF[val] command from within DBG–11.
SET SD [NO]ESC	Enables or disables using the ESCAPE key as the single-step operator (instead of ;S). You can override the SET command condition from within DBG–11 by using the _ESC;T or NOESC;T operating parameters, described in Section 2.1. The _ESC;T or NOESC;T parameter remains in effect until you reload the SD handler.
SET SD [NO]K	Enables or disables Kernal mode for the ~ (tilde) operator. You can override the SET command condition from within DBG–11 by using the _K;T or _NOK;T operating parameters, described in Section 2.1. The _K;T or _NOK;T parameter remains in effect until you reload the SD handler.

SET SD PC
SET SD PS
	Control whether the PC or PS is displayed in the graphics register display, described in Section 3.3.5.
	
	SET SD PC displays the program counter (PC) register as default in the register display. SET SD PS displays the processor status (PS) word register as default in the register display.
	
	You can override either SET command condition from within DBG–11 by using the _PS;T or _PC;T operating parameter. Both parameters are described in Section 2.1. The _PS;T or _PC;T parameter remains in effect until you reload the SD handler.

SET SD [NO]REG	Enables or disables the DBG–11 register display, described in Section 3.3.5, and makes it the default display mode. You can override the SET command condition from within DBG–11 by using the _REG;T or _NOREG;T operating parameter, described in Section 2.1. The _REG;T or _NOREG;T parameter remains in effect until you reload the SD handler.
SET SD [NO]S	Enables or disables Supervisor mode for the ~ (tilde) operator. You can override the SET command condition from within DBG–11 by using the _S;T or _NOS;T operating parameter, described in Section 2.1. The _S;T or _NOS;T parameter remains in effect until you reload the SD handler.
SET SD SYSGEN	Matches SYSGEN parameters of handler to those of monitor so that monitor will install handler without error. Use this SET command if you get the error message *?KMON–F–Conflicting SYSGEN options* when you try to install the handler. Unless you change the monitor or the version of SD you are using, you need to type this command only once.

Table 3–1 (Cont.): SET Commands to Configure DBG–11 (SD Handler)

SET Command	Function
SET SD [NO]U	Enables or disables User mode as default for the ~ (tilde) operator. You can override the SET command parameter from within DBG–11 by using the _U;T or _NOU;T operating parameter, described in Section 2.1. The _U;T or _NOU;T parameter remains in effect until you reload the SD handler.

The option codes in Table 3–2 appear in the identifying header that is displayed on the terminal when you load the SD handler.

Table 3–2: SD Handler Option Identification Codes

Identifier	Meaning
GRH	This version of DBG–11 contains support for the VT100 graphics register display (see Section 3.3.5).
PRO	This version of DBG–11 supports terminal I/O through the printer port of a CTI Bus-based processor. DBG–11 with this support will also do terminal I/O through the console DL serial line on a standard PDP–11 processor.
SD:	This version of DBG–11 is the SD handler implementation.
SOFT	Software I/O. This version of DBG–11 uses system directives to do its terminal I/O.
HARD	Hardware I/O. This version of DBG–11 does terminal I/O by going directly through the console CSR in the I/O page. If you want hardware I/O support on a CTI Bus-based processor, the PRO option must also be enabled. The distributed hardware I/O versions of the SD handler (SDH.SYS and SDHX.SYS) have the PRO option enabled, so they can do hardware I/O on any PDP–11 or CTI Bus-based computer.

The following sequence of commands runs the mapped-memory software I/O version of DBG–11 with the sample background application program shown in Appendix A. (The commands to copy SDSX.SYS, REMOVE SD, SET SD SYSGEN, and INSTALL SD are not necessary if you have already configured and installed SD.)

The LINK command syntax is determined by the monitor you use to run DBG–11. The following syntax is appropriate for a single-mapped (XB or XM) monitor. If you are using a fully-mapped (ZB or ZM) monitor, see Chapter 5 for command syntax.

```
.COPY SDSX.SYS SDX.SYS
.REMOVE SD
.SET SD SYSGEN
.INSTALL SD
.MACRO DBGDEM
.LINK/MAP:DBGDEM DBGDEM
.LOAD SD

DBG Vxx.xx - RT-11  ( SOFT  SD:  GRH )

.R DBGDEM
```

When the processor encounters a BPT instruction in your program, control transfers to the SD handler and DBG–11 displays its initial prompt message. If you are using an unmapped (SB or FB) monitor and debugging the sample program in Appendix A (or another program with a BPT instruction at location 1000), the initial prompt looks like this:

```
BE: 1000
DBG>
```

The characters BE signify breakpoint exception and 1000 is the address of the BPT instruction. A breakpoint exception is caused by the execution of a BPT instruction that was not placed in the program by the *,n;B* command, but rather placed in the program source or patched into the program image by SIPP. Because such a breakpoint is placed in the program outside of DBG–11, it does not appear in the breakpoint tables and is not displayed by the ;D command.

If you are using a mapped monitor in User mode on a processor that does not support separated address space, the prompt looks like this:

```
BE:(U) 1000
DBG>
```

Here, (U) indicates the current processor mode.

If you are using a mapped monitor in User mode on a processor that supports separated I-D address space, the prompt looks like this:

```
BE:(UI) 1000
DBG>
```

Here, (UI) indicates the current processor mode and address space as User Mode and Instruction address space.

Under unmapped (SB, FB) monitors, the processor runs in only Kernel mode, so DBG–11 omits the address mode indicator from the address displays. Under mapped (Xx and Zx) monitors on processors that do not support separated address space, possible values are (U) for User mode, (K) for Kernel mode, and (S) for Supervisor mode. On processors that do support separate address spaces, the current address space (always I for breakpoint displays) is coupled with the processor mode.

If you have a video terminal that supports VT100 escape sequences and you have enabled the graphics register display, Section 3.3.5, DBG–11 puts the register display on the screen before displaying the DBG> prompt.

The DBG> prompt indicates that DBG–11 is ready to accept a command.

3.3 Setting DBG–11's Operating Parameters

You may want to modify some of DBG–11's default parameter settings before you begin to debug your program. These parameters determine the number base that DBG–11 uses to display numeric values, the type of rubouts done, and the range of address and data symbolic offsets. You can change some of them by using SET commands, or you can use the [*qual*;T] command while you are running DBG–11.

3.3.1 Types of Rubouts

DBG–11 tests the terminal SET option status word pointed to by the monitor fixed offset $TCFIG for the default SET TT SCOPE condition. If the default condition exists, DBG–11 does a backspace/space/backspace sequence for rubouts.

If you have issued the DCL command SET TT NOSCOPE, DBG–11 echoes a backslash followed by the characters that are deleted, then a closing backslash.

From within DBG–11, you can issue the DBG–11 command _RUB;T or _NORUB;T to change DBG–11's default rubout setting. If you issue a _[NO]RUB;T command, the in-memory copy of SD changes. Any change remains in effect until you unload and reload the SD handler.

The following command disables video mode rubouts and enables hardcopy rubouts:

```
DBG>_NORUB;T
```

If you want to reenable video mode rubouts, type _RUB;T.

3.3.2 Default Radix

By default, DBG–11 displays all numeric values in octal. If you prefer, DBG–11 can display numeric values in binary, decimal, or hexadecimal.

You can use the [*qual*];R command to change DBG–11's output radix, choosing a value for *qual* from the following list:

_BIN;R
Set output radix to 2.

_OCT;R
Set output radix to 8 (default).

_DEC;R
Set output radix to 10.

_HEX;R
Set output radix to 16.

If you change the output radix with ;R, the change remains in effect until you type another ;R command or unload and load the handler.

The following commands set the output radix to 10, then back to 8:

```
DBG>_DEC;R
DBG>_OCT;R
```

Except where otherwise indicated, all examples in this chapter assume that the default radix is set to octal.

3.3.3 Symbolic Address Display

The DBG–11 qualifiers _ADROFF and _DATOFF control DBG–11's display of symbolic addresses.

By default, DBG–11 attempts to display an address as a symbol or as a positive offset from a symbol. For example, if you define the symbols START and RESULT for the sample program and then examine locations 1000 and 1002, DBG–11 will display the addresses as START and START+2 rather than as octal values, and display the symbol RESULT in the source field of the MOV instruction at location START+2:

```
DBG>1000,START:
DBG>START+230,RESULT:
DBG>1000/
(U) START        /      BPT     LF
(U) START+2      /      MOV     #RESULT,R5       RET
DBG>
```

You can disable symbolic display of addresses by using the _NOADR;T command. The previous example would then look like this:

```
DBG>_NOADR;T
DBG>1000/
(U) 1000         /      BPT     LF
(U) 1002         /      MOV     #RESULT,R5       RET
DBG>
```

To reenable symbolic address display, use the _ADR;T command:

```
DBG>_ADR;T
```

You can establish a range for DBG–11's symbolic address offsets. By default, DBG–11 uses offsets of up to 1000_8 when displaying symbolic addresses. DBG–11 might display the symbolic address START+762 or START+776, for example, if there were no intervening higher-valued symbols from which DBG–11 could define a positive offset. You can limit or extend DBG–11's symbolic address offset range by modifying the internal register _ADROFF. The following sequence assumes that the contents of _ADROFF is at least 16_8 and that you have defined the user symbols listed in Section 3.4:

```
DBG>START/
(U) START        /      BPT     LF
(U) START+2      /      MOV     #RESULT,R5        LF
(U) START+6      /      MOV     VALUE1,R0         LF
(U) START+12     /      MOV     VALUE1+2,R1       LF
(U) START+16     /      CALL    MULPY   RET
DBG>
```

Now change the contents of _ADROFF to 4_8 and repeat:

```
DBG>_ADROFF|
_ADROFF          |      1000     4 RET
DBG>START/
(U) START        /      BPT       LF
(U) START+2      /      MOV      #RESULT,R5        LF
(U) 1006         /      MOV      VALUE1,R0         LF
(U) 1012         /      MOV      VALUE1+2,R1       LF
(U) 1016         /      CALL     MULPY      RET
DBG>
```

When _ADROFF is equal to 4, DBG–11 displays an address as an octal value rather than as a symbol plus an offset, if the address has an offset greater than 4 from a defined symbol.

You can change the initial (default) value of _ADROFF by using the SET SD ADROFF=*val* command (see Table 3-1).

3.3.4 Symbolic Data Display

The DBG–11 internal symbols _DATOFF and _SYM are similar to _ADROFF and _ADR, respectively. They let you control the way DBG–11 displays symbolic data. The symbol _DATOFF defines the range of symbolic data offsets, and _SYM enables and disables symbolic data display.

The following sequence of DBG–11 commands illustrates the effect of _SYM and _NOSYM. This example assumes that START and RESULT are the only user symbols defined.

First, disable symbolic data display:

```
DBG>_NOSYM;T
DBG>START+2/
(U) START+2      /     MOV     #1374,R5        RET
DBG>
```

Now, enable symbolic data display and examine the same location again:

```
DBG>_SYM;T
DBG>START+2/
(U) START+2      /     MOV     #RESULT,R5      RET
DBG>
```

The next series of DBG–11 commands illustrates the effect of _DATOFF. The example assumes that the initial (default) value of _DATOFF is 100_8 and START is the only user symbol defined.

```
DBG>START+2/
(U) START+2      /     MOV     #1374,R5        RET
DBG>_DATOFF[
_DATOFF          [     100     400 RET
DBG>START+2/
(U) START+2      /     MOV     #START+374,R5   RET
DBG>
```

When location START+2 is opened the first time, the source argument for the MOV instruction displays as an octal value because there is no symbol defined with a value within _DATOFF bytes of the source argument value from which DBG–11 can define a positive offset. By changing the contents of _DATOFF from 100 to 400, however, the symbol START with a value of 1000 falls within _DATOFF bytes of the value 1374. Opening location START+2 again now displays the value 1374 as START+374, since the offset 374 is less than the value of _DATOFF.

You can change the initial (default) value of _DATOFF by using the SET SD DATOFF=*val* command (see Table 3-1).

3.3.5 Graphics Register Display

If you have a video terminal that supports VT100 escape sequences, you can use DBG–11's real-time register display feature. When you enable the graphics register display, DBG–11 continuously displays on the upper portion of your terminal screen the contents of your machine's registers, as well as the contents of the first four words of your program's stack. DBG–11 updates the screen display every time a register's value changes.

Issue the DCL command SET TT NOCRLF before enabling DBG–11's graphics register display. DBG–11 automatically uses the graphics register display if you issue the DCL command SET SD REG. If you do not issue the monitor SET SD REG command, you can turn on the display from within DBG–11 by typing the command _REG;T in response to the DBG–11 prompt:

```
DBG>_REG;T
```

If you want to disable the graphics register display, you can issue the DCL command SET SD NOREG before you load the SD handler, or you can type the DBG–11 command _NOREG;T in response to the DBG–11 prompt.

When the graphics register display is enabled, DBG–11 displays its DBG> prompt below the register display and waits for you to type a command:

```
+------- R0 -------+------- R1 -------+------- R2 -------+------- R3 -------+------- R4 -------+------- R5 -------+
|           020056 |           000056 |           000000 |           000000 |           155166 |           102046 |
|          056,040 |          056,000 |          000,000 |          000,000 |          166,332 |          046,204 |
| ".          %EE8 |   ".^@     % AF  | "^@^@   %        | "^@^@   %        | "vZ         %48F |  "&^D   %UEO     |
+------------------+------------------+------------------+------------------+------------------+------------------+
| PC =      001000 | SP =      001000 | (SP) =000003     | 2(SP)=016700     | 4(SP)=000144     | 6(SP)=016701     |
+------------------+------------------+------------------+------------------+------------------+------------------+


DBG>
```

The boxes in the top row show the contents of registers R0 through R5. The first line in each box gives the current value of each register in octal (the default), or in hexadecimal or decimal if you have changed the output radix using the [*val*];R command. (If you set the output radix to binary, the values in the boxes are still shown as octal.)

The second line in each box gives the current byte values of each register using the current output radix. Binary radix again defaults to octal for this display.

The third line shows the contents of the registers as two ASCII characters on the left and as three Radix–50 characters on the right. DBG–11 displays the control character codes 000 through 037_8 as ^X when they appear in the ASCII display. The character displayed as X has the ASCII value of code + 100, where code is a value in the range 000 through 037_8. For example, 004 (or 204, since DBG–11 ignores the eighth bit) displays as ^D, because 004 + 100 equals 104, the ASCII code for D. Similarly, DBG–11 displays 000 as ^@, because 000 + 100 is the ASCII code for the at-sign. The nonprinting ASCII code 177_8 is treated as a special case and displayed as ^?.

The second row of boxes displays the current values of the program counter (PC), the stack pointer (SP), and the top four values on the stack. You can issue the command _PS;T from within DBG–11 to change display of the program counter to display of the processor status (PS) word.

3.4 Defining and Deleting Symbols

After you set DBG–11's operating parameters, you will probably want to define some symbols as your first step in using DBG–11. DBG–11 accepts up to 20 user-defined symbols.

Look at the link map, DBGDEM.MAP, created when you assembled and linked the example in Appendix A:

```
RT-11 LINK  V05.25      Load Map        Tuesday 18-Dec-90 12:27  Page 1
DBGDEM.SAV      Title:  DBGDEM  Ident:

Section  Addr   Size    Global  Value   Global  Value   Global  Value

 . ABS.  000000 001000 = 256.   words  (RW,I,GBL,ABS,OVR)
 INST    001000 000170 = 60.    words  (RW,I,LCL,REL,CON)
                        START   001000  MULPY   001102  PUTNUM  001124
 DATA    001170 000324 = 106.   words  (RW,D,LCL,REL,CON)
                        VALUE1  001334  VALUE2  001336  ANSWER  001340
                        X       001342  EQ      001362  RESULT  001374

Transfer address = 001000, High limit = 001512 = 421.   words
```

Use the colon character (:) to define a symbol, with the following syntax:

```
<addr>,<symbol>:
```

For example, you can define the following symbols according to the locations indicated in the link map:

```
DBG>1000,START:
DBG>1102,MULPY:
DBG>1124,PUTNUM:
DBG>1334,VALUE1:
DBG>17,DEF:
DBG>1374,RESULT:
DBG>
```

Occasionally you may want to delete a symbol definition that you have made. Use the ;K command to delete symbol definitions from DBG–11's symbol table. For example, to delete the symbol DEF that was just defined in the previous example, use the command

```
DBG>DEF;K
```

3.5 Examining Symbols

You can use the equal sign (=) command to look at the value of any user-defined symbol. Type the name of the symbol followed by an equal sign; DBG–11 displays the numeric value of the symbol in the current output radix.

The following commands display the values of symbols defined in Section 3.4, using various output radixes:

```
DBG>START= 1000
DBG>_DEC;R
DBG>START= 512
DBG>_BIN;R
DBG>START= 1000000000
```

In this example, DBG–11 displays the value of START in octal, since that is DBG–11's default output radix. The command _DEC;R changes the radix to decimal, so DBG–11 now displays the value of START as 512_{10}. The command _BIN;R changes the output radix to binary, so repeating the previous command displays the value of START as a binary number.

You can also use the equal sign command to look at the value of . , the current location symbol, and the value of Q, the last displayed value symbol. These symbols are explained in more detail in Section 3.10.

3.6 Setting, Displaying, and Removing Breakpoints

You can set up to eight breakpoints in your program to interrupt program execution and transfer control back to DBG–11. You can then use DBG–11 to examine the state of your program at the time of each breakpoint to determine if the program is executing properly. Use the following syntax to set a breakpoint in your program:

```
addr[,n];B
```

You can include *n*, where *n* can be 1 through 10_8, to specify which of the 8 breakpoints you want to set at a particular address. If you omit *n*, DBG–11 uses the lowest-numbered breakpoint that is free.

Use the syntax

```
[,n];B
```

to remove a breakpoint from your program. If you include *n*, DBG–11 removes that numbered breakpoint from your program. If you omit *n* and just type ;B, DBG–11 removes all breakpoints.

For example, you can set a breakpoint at relative location 0016 in the sample program, so you can verify that the arguments being passed to the multiply subroutine are correct. You can also set a breakpoint at relative location 0022 to give yourself a chance to check the value returned from the multiply subroutine.

```
DBG>START+16;B
DBG>START+22;B
```

Those examples do not specify any value for *n*, so in each case DBG–11 assigns the lowest-numbered breakpoint that is not already in use.

Alternatively, the commands could be written

```
DBG>1016;B
DBG>1022;B
```

You display a breakpoint table by using the syntax:

```
;D
```

The ;D command displays the breakpoint table. The table shows the breakpoint number (*BPT*), the 16-bit virtual program address at which the breakpoint is set (*Virtual Address*), the PAR value to map that virtual address to physical memory (*PAR Value*), and the number of times the breakpoint can execute (*Execution Count*).

BPT	**Virtual Address**	**PAR Value**	**Execution Count**
1	000000	000000	0
2	000000	000000	0
3	000000	000000	0
4	000000	000000	0
5	000000	000000	0
6	000000	000000	0
7	000000	000000	0
10	000000	000000	0

To delete the breakpoints, type

```
DBG>,1;B
DBG>,2;B
```

or alternatively,

```
DBG>;B
```

That form of the command deletes all breakpoints.

3.7 Executing the Program

Three DBG–11 commands start or continue the execution of the program you are debugging, as shown in Table 3–3.

Table 3–3: Program Execution

Command	Meaning
[*addr*];G	Starts program at specified address, or at current saved PC value if *addr* is omitted. Provided the monitor has not been damaged, DBG–11 exists to the monitor prompt if you specify an address of 0 (0;G).
[*val*];P	Proceeds with user program execution from the current breakpoint location and stops when the next breakpoint is encountered. If you specify *val*, DBG–11 proceeds with program execution from the current breakpoint location and stops at that breakpoint again only after encountering it a total of *val* times. The default value for *val* is 1. Other breakpoints that may be set are not counted or affected; the program stops when it encounters other breakpoints. You can set a different *val* count for each active breakpoint.
[*val*];S	Executes the number of program instructions equal to *val*, then does another breakpoint trap. If you omit *val* or specify a value of zero (0;S), DBG–11 assumes a value of 1 (single-step). The ;S command lets you proceed through your program one instruction at a time.

3.8 Examining Locations

You can examine a program location in seven modes: instruction, symbolic data, numeric data, byte, word mode ASCII, byte mode ASCII, or Radix–50. Each mode uses a separate character operator, listed in table Table 3–4.

Table 3–4: Examination of Program Locations

Operator	Examine Mode	Address Space
Slash (/)	Instruction	I-space
Left bracket ([ )	Symbolic data	Current
Vertical bar (|)	Numeric data	Current
Backslash (\ )	Byte	Current
Double quote (")	Word mode ASCII	Current
Single quote (')	Byte mode ASCII	Current
Percent (%)	Radix–50	Current

These modes are explained in more detail as follows.

3.8.1 Opening and Closing Locations

To examine a location in your program, you must open the location. To open a location in any of the examine modes, type an address expression in response to the DBG> prompt, followed by one of the mode character operators listed above. DBG–11 displays the address, followed by its contents (in the format determined by your choice of examine mode operators and the selected radix), and waits for your response. You can:

- Close the location and return to DBG–11's command prompt by pressing RETURN.
- Change, then close the location and return to DBG–11's command prompt by typing a numeric expression followed by RETURN.
- Close the location and open another location by pressing a linefeed (LF), circumflex (^), or at-sign (@).
- Close the location in the current address space and open the same location in D-space by pressing the pound sign (#).
- Close the location in the current address space and open the same location in I-space by pressing the at-sign (@).
- Change, then close the location and open another location by typing LF, ^, or @.

If you do not specify an address expression when you use one of the examine operators listed above, DBG–11 opens the current or last-opened location. You can find out what the current location is by using the equal sign operator to look at the value of . (period), the current location symbol.

The following series of commands demonstrates the use of LF and ^ to examine the contents of VALUE1 and VALUE1+2:

```
DBG>VALUE1|
(U) VALUE1        |      123     LF
(U) VALUE1+2      |      17      ^
(U) VALUE1        |      123     RET
DBG>
```

Location VALUE1+2 is equivalent to location VALUE2. However, the symbol VALUE2 has not been defined. Therefore, DBG–11 displays the address as an offset from a symbol that is defined, in this case VALUE1.

Although you can type the at-sign command (@) after any location, the command is useful only when the location contains a memory reference. For example, location START+16 calls the routine MULPY. You can use the @ command to examine location MULPY after opening location START+16:

```
DBG>START+16/
(U) START+16       /      CALL    MULPY    @
(U) MULPY          /      CLR     -(SP)    RET
DBG>
```

Since the LF, ^, @, and RET commands were not preceded by any values, the contents of VALUE1, VALUE1+2, START+16, and MULPY remain unchanged. Section 3.9 illustrates the use of commands to change the contents of a location.

The graphic's register display is the easiest way to track contents and changes in the PC, PS, and other machine registers. If your terminal does not support the register display (is not VT100 compatible), you can open the machine registers R0, R1, R2, R3, R4, R5, SP, PC, and PS (the processor status word) by specifying their names in response to DBG–11's prompt. Follow the name with one of the character operators. For example, you can look at the contents of the PS by typing

```
DBG>PS|
PS                 |       140000    RET
DBG>
```

3.8.2 Instruction Mode Examine (/)

Use the slash (/) character to open the current location in instruction mode. For example, if you type / in response to the DBG–11 prompt at the starting address of the sample program, DBG–11 displays the following:

```
DBG>/
(U) START          /       BPT
```

DBG–11 pauses at the end of the line, waiting for input. The location is now open and can be changed. If you press RET, DBG–11 closes the location without changing the contents and displays the DBG–11 command prompt.

You can also specify a particular address to open in instruction mode. Consider the following sequence of commands:

```
DBG>1002/
(U) START+2         /       MOV     #RESULT,R5       RET
DBG>MULPY/
(U) MULPY           /       CLR     -(SP)
```

The first command specifies the address to open as an absolute octal value. The second command uses the user-defined symbol MULPY to specify the address to open.

3.8.3 Symbolic Data Mode Examine ([)

The left bracket ([) character opens a location in symbolic data mode. Use this mode to examine the contents of a location when the contents are defined as a symbol. For example, to examine the contents of location VALUE1, where the contents are equal to 123_8 and the symbol ABC is defined as 123_8 in the DBG–11 user symbol table, the command and DBG–11 response looks like this:

```
DBG>VALUE1[
(U) VALUE1         [       ABC
```

If you use this mode to examine a location and you have not defined a symbol that represents the contents, the display is determined by the symbol offset range. If the value at the location is within range, the display shows an offset from the nearest symbol. If you have not defined any symbol within range, the output is numeric, using the current output radix. You can set the range by using the internal registers _ADROFF and _DATOFF. The following example shows a value that is out of range:

```
DBG>VALUE1+2[
(U) VALUE1+2       [       17
```

Even though the contents of VALUE1+2 is defined as the symbol DEF in the sample program, the symbol DEF was not placed in DBG–11's user symbol table. Therefore, because of the offset range, DBG–11 displays the contents of VALUE1+2 as 17, rather than symbolically.

Location VALUE1+2 is, of course, equivalent to location VALUE2. However, the symbol VALUE2 has not been defined. Therefore, you must specify the address as an offset from a symbol that is defined, or as an absolute octal address. (Alternatively, you can define the symbol VALUE2 using the : command.)

3.8.4 Numeric Data Mode Examine (|)

If you want to know what the numeric data values are, rather than their symbolic equivalents, use the vertical bar (|) data mode examine command. This command displays the contents of a location as a numeric value using the current default output radix.

The previous section shows how to examine location VALUE1 in symbolic data mode. That mode displays the contents of VALUE1 as the symbol ABC. If you use the | command instead of the [command to examine location VALUE1, the display is in numeric rather than symbolic form:

```
DBG>VALUE1|
(U) VALUE1         |      123
```

Since the sample program does a decimal multiply and produces a decimal result, you probably want to know what VALUE1 and VALUE2 are in decimal. The following commands set the output radix to decimal, examine locations VALUE1 and VALUE2 (displayed as VALUE1+2, since the symbol VALUE2 is not defined in DBG–11's user symbol table), and return the output radix to octal:

```
DBG>_DEC;R
DBG>VALUE1|
(U) VALUE1         |      83        LF
(U) VALUE1+2       |      15        RET
DBG>_OCT;R
```

3.8.5 Byte Mode Examine (\)

The backslash (\) command lets you examine data one byte at a time, rather than one word at a time. The ASCII string *multiplied by*, in the sample program, can illustrate the use of the \ command:

```
DBG>START+342\
(U) VALUE1+6      \      40       LF
(U) VALUE1+7      \      155      LF
(U) VALUE1+10     \      165      LF
(U) VALUE1+11     \      154      LF
(U) VALUE1+12     \      164      LF
(U) VALUE1+13     \      151      LF
(U) VALUE1+14     \      160      RET
DBG>
```

You can look up those byte values in a table of ASCII characters and determine that they are the ASCII characters *multip*. More simply, you can use the ASCII mode examine commands " and ′ described in Section 3.8.6.

3.8.6 ASCII Mode Examine (′ or ")

Although the \ command lets you examine data one byte at a time, if the bytes are ASCII characters, you probably want to see them as ASCII characters, not as numeric quantities. Use the double quote (") or single quote (′) command to display locations as ASCII words or bytes, respectively.

The remainder of the ASCII string *multiplied by*, examined in byte mode in Section 3.8.5, is easily identified when examined with the ′ command:

```
DBG>VALUE1+15′
(U) VALUE1+15     ′      l       LF
(U) VALUE1+16     ′      i       LF
(U) VALUE1+17     ′      e       LF
(U) VALUE1+20     ′      d       LF
(U) VALUE1+21     ′              LF
(U) VALUE1+22     ′      b       LF
(U) VALUE1+23     ′      y       LF
(U) VALUE1+24     ′              LF
(U) VALUE1+25     ′      ^@      RET
DBG>
```

The terminating byte of 200_8 displays as ^@.

3.8.7 Radix–50 Mode Examine (%)

MACRO–11 programs can also contain Radix–50 character strings. Use DBG–11's % command to display words as Radix–50 equivalents. Although the sample program does not contain any defined Radix–50 character strings, the following example illustrates the use of the % command:

```
DBG>START+6%
(U) START+6        %       D0P
```

The location START+6 contains 16700_8, the first word of the instruction MOV VALUE1,R0, but the % command interprets the contents of the location as the

Radix–50 string D0P. In other words, the % command assumes the value 16700 results from the definition .RAD50 /D0P/.

3.9 Changing Locations

Whenever you use any of the examine commands described in Section 3.8 to examine a location in your program, the location you examine is open and available for change. You can enter MACRO–11 instructions, numeric values as numbers or defined symbols, ASCII characters, or Radix–50 strings.

3.9.1 Entering MACRO–11 Instructions

You can type in symbolic MACRO–11 instructions to DBG–11 as new contents for open locations. DBG–11 recognizes most permanent MACRO–11 instruction mnemonics, except those defined for the CIS and FPP instruction sets.

DBG–11 is not an assembler, and its capabilities are limited. If you enter a two- or three-word instruction as a replacement for a single-word instruction or vice-versa, DBG–11 does the replacement and writes over or leaves unchanged the words that are not part of the instruction. You must count words yourself and make sure that everything is aligned properly.

For example, you can replace the initial BPT instruction in the sample program with another one-word instruction, perhaps CLR R0:

```
DBG>START/
(U) START        /      BPT      CLR R0 RET
DBG>START/
(U) START        /      CLR      R0        RET
DBG>
```

However, If you try to replace it with a two-word instruction, say, CLR ANSWER, the results are less desirable, as illustrated below.

```
DBG>START+340,ANSWER:
DBG>START/
(U) START        /      BPT     CLR ANSWER RET
DBG>START/
(U) START        /      CLR     ANSWER   LF
(U) START+4      /      BNE     466      LF
(U) START+6      /      MOV     VALUE1,R0        RET
DBG>
```

The leftover portion of the instruction MOV #RESULT,R5 gets interpreted as BNE 466, which is meaningless in the context of the program.

If you have tried this example on your own computer, be sure you put the original instructions of the program back before you try to run it.

3.9.2 Entering Numeric Values

You can type in numeric values using the digits 0 through 9 and an optional decimal point. If the number you type contains an 8, 9, or decimal point, DBG–11 interprets the number as decimal. Otherwise, DBG–11 interprets the number as octal. Use the _$ prefix operator to enter numbers in hexadecimal. You cannot enter a number in binary.

Locations VALUE1 and VALUE2 in the sample program contain the two numbers that the program multiplies together. Consider the following series of commands:

```
DBG>VALUE1/
(U) VALUE1       /       JMP     (R3)+    RET
DBG>|
(U) VALUE1      |       123     85 RET
DBG>|
(U) VALUE1      |       125      RET
DBG>
```

Typing a slash as the first command opens the location VALUE1 in instruction mode. Since VALUE1 contains a data value, not a PDP–11 instruction, the resulting display, JMP (R3)+, has no meaning. The second command line corrects that mistake by using a vertical bar (|) to open the current (last opened) location in data mode, which displays a value of 123_8. Typing 85 followed by a line terminator changes the contents of VALUE1 to 85_{10}. DBG–11 interprets the number as decimal because it contains an 8. Typing another vertical bar again displays the contents of VALUE1, this time as 125_8.

You can also enter the name of any user-defined symbol as the new contents of a location. The following commands define the symbol ABC, then use it to change the contents of location VALUE1.

```
DBG>123,ABC:
DBG>VALUE1|
(U) VALUE1       |       125     ABC RET
DBG>|
(U) VALUE1       |       123      RET
DBG>[
(U) VALUE1       [       ABC      RET
DBG>
```

The vertical bar opens location VALUE1 and displays the contents as 125_8, set in the previous example. Entering the symbol ABC followed by a RETURN changes the contents of VALUE1 to the value of symbol ABC. Typing another vertical bar reopens the current location and displays the new contents of 123_8, the value of symbol ABC. If you use [, the symbolic data display command, DBG–11 displays the contents of VALUE1 symbolically and confirms that 123_8 is equal to ABC.

3.9.3 Entering ASCII, Hexadecimal, and Radix–50 Strings

You can enter data in ASCII, hexadecimal, or Radix–50 format by preceding the data with a special prefix operator. The prefix operator tells DBG–11 to interpret what follows as a particular type of character string.

You can specify a word or byte address for a destination. If you specify a byte address and the data you enter is too large to fit in one byte, DBG–11 truncates without reporting an error.

ASCII

Use the _′ or _" prefix operator to enter one or two ASCII characters at a time. The ASCII equivalent of the character that follows the _′ prefix, or the two characters that follow the _" prefix, are stored one character per byte at the address you specify.

The sample program contains the ASCII string *multiplied by*. You can change *by* to uppercase, for example, with either of the following DBG–11 command lines:

```
DBG>VALUE1+22"
(U) VALUE1+22     "       by       _"BY RET
DBG>
```

or

```
DBG>VALUE1+22′
(U) VALUE1+22     ′       b        _′B LF
(U) VALUE1+23     ′       y        _′Y RET
DBG>
```

In the first case, the _" prefix operator requires that you type two characters, B and Y. The _′ operator, on the other hand, lets you replace a single character (byte) at a time.

Hexadecimal

The _$ prefix operator lets you enter up to four hexadecimal digits. DBG–11 stores the binary equivalent at the address you specify. You can use this operator to change the contents of location VALUE1 to a value corresponding to the hexadecimal string 3F:

```
DBG>VALUE1|
(U) VALUE1        |       123      _$3F
DBG>|
(U) VALUE1        |       77        RET
DBG>
```

The data mode examine operator opens location VALUE1 and displays its octal contents, 123. The prefix operator _$ tells DBG–11 to interpret the next two characters, 3F, as two hexadecimal digits and to store them in the currently open location. Reopening location VALUE1 shows that the contents of VALUE1 are now 77_8, which is equivalent to $3F_{16}$. You can also display the contents of VALUE1 as a hexadecimal value by changing DBG–11's output radix to hexadecimal with the _HEX;R command.

Radix–50

Use the _% prefix operator to enter up to three Radix–50 characters. If you want to include a leading or embedded space as part of the Radix–50 character string, replace the space with an underscore (_) character when you type in the string. For example, typing the characters _%_AC stores the Radix–50 string <space>AC. If you type less than three characters, DBG–11 assumes trailing spaces.

The sample program uses no Radix–50 strings, but, for illustration, the following example stores the Radix–50 string LST in location START+340, then uses the | and % commands to examine the location and verify the new contents:

```
DBG>START+340|
(U) VALUE1+4     |       0       _%LST RET
DBG>|
(U) VALUE1+4     |       47014   %LST    RET
DBG>
```

3.10 Special Symbols . and Q

When debugging, you frequently need to refer to the current location, that is, where you are in the program right now. DBG–11 defines the special symbol period (.) to be equal to the address of the current location. The current location is the location that is currently open, if any, or the last location that was open. You can define the value of . yourself, using the : symbol definition command, but its value will change as soon as you open a new location.

In general, if you do not specify an address in a DBG–11 command that requires an address argument, DBG–11 defaults to the current location.

DBG–11 defines the symbol Q to be equal to the last symbolic or numeric value that was displayed on the terminal. In general, that value will be the last symbol in an instruction, or the numeric equivalent of ASCII or Radix–50 characters. If you open a location in instruction mode but the instruction refers only to registers (for example, MOV R1,R0), Q is set to the opened address. You can define the value of Q yourself, using the : symbol definition command, but its value will change as soon as you examine a new numeric value.

The following examples show the value of Q after DBG–11 displays various types of expressions:

```
DBG>START+2/
(U) START+2     /       MOV     #RESULT,R5
```

Q = 1374, the value of the symbol RESULT.

```
DBG>START+16/
(U) START+16    /       CALL     MULPY
```

Q = 1102, the value of the symbol MULPY.

```
DBG>PUTNUM/
(U) PUTNUM      /        MOV     R0,-(SP)
```

Q = 1124, the value of the symbol PUTNUM.

```
DBG>START+2%
(U) START+2     %        CSM
```

Q = 12705, the Radix–50 equivalent of *CSM*.

```
DBG>START+2[
(U) START+2     [        12705
```

Q = 12705, the octal contents of START+2.

```
DBG>VALUE1+22"
(U) VALUE1+22   "       by
```

Q = 74542, the ASCII equivalent of *by*.

3.11 Putting It All Together

Previous sections have shown examples of individual DBG–11 commands. This section shows a simple DBG–11 session from start to finish, which may give a more meaningful context for some of the commands. It is based on the sample program in Appendix A.

The sample program multiplies two numbers and stores the result as an ASCII string without giving any visible output. To find out if the program runs properly, you can watch the program execute with DBG:

```
.LOAD SD RET
```

```
DBG V01.00 - RT-11  ( SOFT  SD:  GRH )
```

```
.R DBGDEM RET
```

```
BE:(U) 1000
DBG>
```

DBG–11 stops at the BPT instruction at the beginning of the program, prints its command prompt, and waits for a command.

Set the operating parameters for DBG–11; for example, enable video mode rubouts:

```
DBG>_RUB;T
```

Define user symbols to which you may want to refer when debugging the program:

```
DBG>1000,START:
DBG>START+102,MULPY:
DBG>START+124,PUTNUM:
DBG>START+334,VALUE1:
DBG>START+374,RESULT:
DBG>
```

Set breakpoints at places in the program where you want to check location values or program status:

```
DBG>START+16;B
DBG>START+22;B
```

Now, continue executing the program.

```
DBG>;P
BPT1>(U) START+16      /       CALL    MULPY   RET
DBG>
```

Display the breakpoint table:

```
DBG>;D
```

BPT	Virtual Address	PAR Value	Execution Count
1	001016	000000	1
2	001022	000000	1
3	000000	000000	0
4	000000	000000	0
5	000000	000000	0
6	000000	000000	0
7	000000	000000	0
10	000000	000000	0

DBG–11 stops at the first breakpoint, displays the instruction to be executed, and prints the DBG–11 command prompt. The program is about to call the integer multiply subroutine. Check the values of the parameters being passed in R0 and R1:

```
DBG>R0|
R0      |       123     LF
R1      |       17      ;P
```

If you are using the graphics register display (Section 3.3.5), you can check the contents of R0 and R1 there, without having to type any DBG–11 commands.

The ;P command continues execution until DBG–11 encounters the next breakpoint, which is set after the program returns from the multiply subroutine with the answer in R0. You can check the result of the multiply by looking at the contents of R0. The program stores the final result in decimal, not octal form; to examine R0 as a decimal value, change the output radix mode to decimal:

```
BPT2>(U) START+22      /       MOV     R0,ANSWER   RET
DBG>R0|
R0      |        2335    _DEC;R
DBG>R0|
R0      |        1245    _OCT;R
DBG>;S
```

After DBG–11 displays the answer in decimal, the _OCT;R command changes the default output radix back to octal. A ;S command single-steps to the next instruction and breaks again:

```
BPT0>(U) START+26       MOV     VALUE1,R0       ;S
BPT0>(U) START+32       CALL    PUTNUM  RET
DBG>
```

Define a breakpoint at the HALT instruction in the program, and proceed:

```
DBG>START+100;B
DBG>;P
BPT3>(U) START+100      HALT
```

Display the breakpoint table and note the third breakpoint:

```
DBG>;D
```

	Virtual	**PAR**	**Execution**
BPT	**Address**	**Value**	**Count**
1	001016	000000	1
2	001022	000000	1
3	001100	000000	1
4	000000	000000	0
5	000000	000000	0
6	000000	000000	0
7	000000	000000	0
10	000000	000000	0

At this point, the sample program has finished executing and stored the result of its computation as an ASCII string beginning at the symbol RESULT. Use the word-mode ASCII examine operator to check the answer:

```
BPT3>(U) START+100      HALT    RESULT"83 LF
(U) RESULT+2    "        m      LF
(U) RESULT+4    "       ul      LF
(U) RESULT+6    "       ti      LF
(U) RESULT+10   "       pl      LF
(U) RESULT+12   "       ie      LF
(U) RESULT+14   "       d       LF
(U) RESULT+16   "       BY      LF
(U) RESULT+20   "        1      LF
(U) RESULT+22   "       5       LF
(U) RESULT+24   "       eq      LF
(U) RESULT+26   "       ua      LF
(U) RESULT+30   "       ls      LF
(U) RESULT+32   "        1      LF
(U) RESULT+34   "       24      LF
(U) RESULT+36   "       5^@     RET
DBG>
```

The answer is correct. Type another ;P command to execute the HALT instruction:

```
DBG>;P
```

```
T10:(U) MULPY
```

This example was run using an RT–11 XM system with the sample program running in User mode, so the HALT instruction causes a trap to 10 (on some processors, a HALT in User mode may cause a trap to 4). DBG–11 intercepts the trap and reports it as shown.

The HALT instruction may also behave differently on processors that do not support memory mapping. If you run this example on an unmapped system, the HALT instruction may not cause a trap and the processor may do a hard halt at the HALT instruction.

Chapter 4

The DBGSYM Utility Program

You can define up to 20 user symbols by using DBG–11's colon (:) symbol definition command described in Section 2.2. However, that method of defining symbols can become tedious and the 20-symbol limit may prove restrictive if you are debugging a large program.

Use the utility program DBGSYM.SAV to create a DBG–11 symbol table from a symbol table (.STB) file.[1] You do not need to type the symbol definitions, and there is no limit except memory space on the number of symbols you can define. DBGSYM takes .STB file symbol definitions, adds prefix and suffix text, and creates a pseudodevice handler ST.SYS or STX.SYS. When you load the SD handler, DBG–11 looks for the presence of ST and, if found, uses the symbol definitions that DBG–11 finds there.

You can use the *symbol*;K command to delete symbols defined in ST. If you delete a symbol defined in ST, you can use the colon command to define another symbol in its place, in addition to the 20 other symbols you can define using the colon command. For example, if you define 30 symbols in ST and then delete them all with the ;K command, you can then define a total of 50 symbols using the colon command.

To define symbols with DBGSYM, perform the following steps:

1. Define as global all the symbols in your program that you want to use with DBG–11 (only global symbols go into an .STB file).
2. Create an .STB file by specifying a third output file name in your CSI command line when you link the program that you plan to debug, or use the /SYMBOL:*file* option in your DCL command line.
3. Run DBGSYM and specify the .STB file as input. DBG–11 uses the default output file name ST.SYS if you are running under the unmapped monitors, STX.SYS if you are running under the mapped monitors. You can specify another output file name if you want, but you must name the file STx.SYS before you can use it with DBG–11. If you try to load ST under a different name, you will get the error message *?ST–F–handler must be named ST*.
4. Load SD, then install and load ST on your system volume. DBGSYM makes the SYSGEN configuration bits of the ST handler it creates compatible with the currently running monitor. If you copy the ST handler to a different system or boot a different monitor, you may need to issue the command SET ST SYSGEN before installing ST.

[1] Refer to the *PDP–11 MACRO–11 Language Reference Manual* for more information about .STB files.

5. Run the program you want to debug, as described in Chapter 3. DBG–11 will look for the ST device handler and use any symbol definitions that it finds.
6. When you finish debugging a program, you can unload the ST handler you have been using, and then create, install, and load a new version of the ST handler containing symbols for another program you want to debug.

Chapter 5

Advanced DBG–11 Techniques

This chapter discusses some ways of using DBG–11 you may find useful in special situations.

5.1 Debugging a Device Handler

You must use the hardware I/O versions of SD (SDH.SYS or SDHX.SYS) when you debug a device handler. You can use either of two techniques to coordinate SD with the device handler:

- Issue the RT–11 monitor SET command, SET SD BREAK, after you load the SD handler and the handler you want to debug. When you issue SET SD BREAK, control transfers to DBG–11. You can put breakpoints in the handler you want to debug, examine locations, or perform other DBG–11 operations without putting a BPT instruction in your handler code. After setting breakpoints, type ;P to return to the RT–11 monitor, and run a test program to call the handler. When the test program calls the handler and encounters a DBG–11 breakpoint, control transfers to DBG–11.

  The SET SD BREAK command may be useful in other circumstances as well, but its functionality is somewhat limited. The program code you want to debug must be loaded when you issue the SET SD BREAK command, and you can only issue the SET command from the RT–11 monitor level.

- You can use the same technique to initially enter a device handler as you use when debugging a background program. Put a BPT instruction at the start of the handler you are debugging. Load the handler, load SD, and then run a test program to call your handler. When the test program calls the handler and executes the BPT instruction, control will transfer to DBG–11. You can then set additional breakpoints, examine locations, and perform any other DBG–11 operations.

5.2 Separate Address Spaces and Multiple Processor Modes

This section applies to only mapped monitors, because with unmapped monitors, all memory is within one address space and processor mode. All mapped monitors support Kernal and User mode. Only fully-mapped (ZB and ZM) monitors support separated I-D address space and Supervisor mode.

5.2.1 Linking a Separate Address Space Program for Debugging

The DBG–11 symbol table stores 16-bit location values and does not distinguish between Instruction and Data PSECTs. If your program is linked such that Instruction and Data PSECTs occupy the same virtual addresses in separate address spaces, symbols defined in one address space become mixed with those of the other. That makes debugging, using symbols, very confusing.

A useful technique is to link your program (for debugging purposes only) such that the Data PSECTs are located above the Instruction PSECTs. The linker contains a number of options that let you control PSECT placement; use the /BOUNDARY:value:DAS option to control the base of the Data PSECTs. In the example program, the base of the Data PSECT is linked at boundary value 1024. (The boundary values must be a power of 2.)

After assembling DBGDEM.MAC, link the object module by issuing the following command. The command specifies an extended save image (/IDSPAC) with the Data PSECT (DATA) located at boundary value 1024_{10}:

```
.LINK/MAP:DBGDEM/IDSPAC DBGDEM/BOUNDARY:1024.:DAS RET
Data boundary section?  DATA RET
.
```

This produces the following link map:

```
RT-11 LINK  V05.25      Load Map        Tuesday 18-Dec-90 17:18  Page 1
DBGID .SAV      Title:  DBGDEM  Ident:

Section  Addr   Size    Global  Value   Global  Value   Global  Value

 . ABS.  000000 001000 = 256.   words  (RW,D,GBL,ABS,OVR)
 INST    001000 001000 = 256.   words  (RW,I,LCL,REL,CON)
                        START   001000  MULPY   001102  PUTNUM  001124
 DATA    001000 000324 = 106.   words  (RW,D,LCL,REL,CON)
                        VALUE1  001144  VALUE2  001146  ANSWER  001150
                        X       001152  EQ      001172  RESULT  001204

Transfer address = 001000, High limit = 001776 = 511.   words
```

Without raising the DATA PSECT boundary, symbols START and VALUE1 would have the same location value, 1000, and various Instruction and Data symbols would be interspersed. Using this method, all symbol locations in both Instruction and Data address spaces can be defined. Note again that this procedure is probably useful only for debugging. When linking your debugged program, either let the linker determine location values or specify values for your own purposes.

When you work through the examples in this manual, use the location values shown above (or those you obtain from your own link map) as symbol locations. Assuming that, the general discussion will then agree with what you see displayed on your screen.

5.2.2 Breakpoints

The DBG–11 breakpoint table stores breakpoints as 22-bit values (physical memory location). Therefore, DBG–11 keeps track of breakpoints set under all processor modes. You can, however, only issue breakpoints in the Instruction address space.

When the current location is at a breakpoint, the ;D command displays the breakpoint table. The table shows the breakpoint number (*BPT*), the 16-bit virtual program address at which the breakpoint is set (*Virtual Address*), the PAR value to map that virtual address to physical memory (*PAR Value*), and the number of times the breakpoint can execute (*Execution Count*). The following is an example breakpoint table in the fully-mapped environment:

BPT	Virtual Address	PAR Value	Execution Count
1	001016	004450	1
2	001022	004450	1
3	000000	000000	0
4	000000	000000	0
5	000000	000000	0
6	000000	000000	0
7	000000	000000	0
10	000000	000000	0

5.2.3 Multiple Processor Modes

In mapped systems, the processor mapping mode (Kernal, Supervisor, or User) determines whether DBG–11 references the same memory as the program or device handler being debugged and the rest of the system. Typically, a foreground or background job is mapped in User mode, the monitor and device handlers are mapped in Kernel mode, and elements such as libraries can be mapped in Supervisor mode.

By default, DBG–11 tracks and uses the current address mode in the processor status word (PSW); the same as the program being debugged. When you debug a background or foreground job under the mapped monitors, DBG–11 can reference whatever is mapped in User or Supervisor mode. If user mapping does not correspond to kernel mapping, the monitor and handlers are invisible to you. Likewise, if you use the DBG–11 procedures described in Section 5.1 to debug a device handler, DBG–11 can access whatever is mapped in Kernel mode, the mapping mode of the handler, but any background or foreground job mapped in User or Supervisor mode is invisible to DBG–11.

Occasionally, you may want to change the contents of the PSW current mode field to access memory outside the current mapping. You can temporarily change the processor addressing mode (the current mode bits in the PSW) by issuing the [*qual*];M command, as described in Section 2.1. For example, this technique provides a simple way to look at the I/O page (mapped in Kernel mode) while debugging a program in User mode. Issue the _K;M command to change the current mapping mode from User to Kernel mode. DBG–11 automatically returns to the initial mapping mode before executing any Proceed (;P) command. You can explicitly return to initial mapping by issuing the ;M command with no qualifier.

If your processor (or your program) does not support Supervisor mode, you can stop the processor mode toggle operator (~) from calling Supervisor mode by issuing the command _NOS;T from within DBG–11. Otherwise, the toggle goes through Supervisor mode each time you toggle between Kernel and User modes.

You can also use DBG–11 to modify the KT11 page address registers (PARs) if you want to examine memory that is not currently mapped in Kernel or User mode. For example, if you are debugging a device handler that does its own extended memory mapping, you may find that you need to use DBG–11 to modify the contents of the PARs.

CAUTION

If you change the contents of a PAR, be sure to restore the contents to the original value, or be sure the new value is what your program expects, before attempting to execute the program.

Appendix A
DBGDEM.MAC Sample Program

The examples in this manual were created using an RT–11 XM single-mapped monitor (any RT–11 system with a VT100-compatible terminal should work as well) and the example program shown below. An explanation of how to assemble and link this program for separated I-D space and a fully-mapped monitor is located in Chapter 5.

If you have difficulty following any command explanation, assemble and link the example program, run it with DBG–11, and try typing the example of the command exactly as given. That may help you understand what the command does.

This program does no I/O itself, and should assemble and run on any PDP–11 that supports the MACRO–11 assembler.

```
DBGDEM.MAC      MACRO V05.05  Tuesday 18-Dec-90 13:39  Page 1

      1                 .TITLE  DBGDEM.MAC
      2                 .ENABL  LC
      3                 .NLIST  BEX
      4
      5                 ; Demonstration program for DBG-11 manual.
      6                 ; Multiplies VALUE1 by VALUE2, puts
      7                 ; result as ASCII string in ANSWER.
      8
      9 000000          .PSECT  INST,I
     10
     11         000123  ABC =   83.     ;The values
     12         000017  DEF =   15.     ; to multiply
     13
     14 000000  000003  START:: BPT
     15 000002  012705          MOV     #RESULT,R5      ;Put result here
     16 000006  016700          MOV     VALUE1,R0       ;Set up R0, R1
     17 000012  016701          MOV     VALUE2,R1
     18 000016  004767          CALL    MULPY           ; R0 <-- R0*R1
     19 000022  010067          MOV     R0,ANSWER       ;Save answer
     20 000026  016700          MOV     VALUE1,R0       ;Store first value
     21 000032  004767          CALL    PUTNUM          ; as ASCII string
     22 000036  012700          MOV     #X,R0           ; multiplied by
     23 000042  004767          CALL    PUTSTR
     24 000046  016700          MOV     VALUE2,R0       ;Store 2nd value
     25 000052  004767          CALL    PUTNUM          ; as ASCII string
     26 000056  012700          MOV     #EQ,R0          ; "equals...
     27 000062  004767          CALL    PUTSTR
     28 000066  016700          MOV     ANSWER,R0       ;Store answer
     29 000072  004767          CALL    PUTNUM          ; as ASCII string
     30 000076  105025          CLRB    (R5)+           ;End with 0 byte
     31 000100  000000          HALT
     32
     33                 ;Single-precision unsigned multiply,
     34                 ; single-precision result.
     35                 ;Multiply R0 by R1, put result in R0.
```

```

000102  005046  MULPY::  CLR     -(SP)           ;Put answer on stack
000104  000402          BR      3$
000106  060016  1$:     ADD     R0,(SP)
000110  006300  2$:     ASL     R0
000112  006201  3$:     ASR     R1
000114  103774          BCS     1$
000116  001374          BNE     2$
000120  012600          MOV     (SP)+,R0        ;Get answer from stack
000122  000207          RETURN                  ;Return to caller

                ; Store contents of R0 as decimal number
                ; as ASCII string with no leading zeros.
                ; R5 --> place to store string.

000124  010046  PUTNUM:: MOV     R0,-(SP)
000126  005000          CLR     R0
000130  005200  1$:     INC     R0
000132  162716          SUB     #10.,(SP)
000136  002374          BGE     1$
000140  062716          ADD     #60+10.,(SP)
000144  005300          DEC     R0
000146  001402          BEQ     2$
000150  004767          CALL    PUTNUM
000154  112625  2$:     MOVB    (SP)+,(R5)+
000156  000207          RETURN

                ; Copy ASCII string pointed to by R0 to locations
                ; pointed to by R5.  End on 0 or 200 byte, don’t
                ; store the 0 or 200 byte.

                        .ENABL  LSB
000160  112025  10$:    MOVB    (R0)+,(R5)+
000162  105710  PUTSTR::TSTB    (R0)
000164  003375          BGT     10$
000166  000207          RETURN
                        .DSABL  LSB

000000          .PSECT  DATA,D
000000                  .BLKW    50.             ;Put some space here
000144  000123  VALUE1:: .WORD   ABC             ;Arbitrary values
000146  000017  VALUE2:: .WORD   DEF             ; to multiply
000150  000000  ANSWER:: .WORD   0               ;Save result here
000152     040  X::      .ASCII  / multiplied by /<200>
000172     040  EQ::     .ASCII  / equals /<200>
                        .EVEN
000204          RESULT:: .BLKB   80.             ;Final string here
                        .EVEN
        000000’         .END    START
```

Appendix B

DBG–11 and DBGSYM Error Messages

?DBGSYM–E–No input .STB file specified

Explanation: You did not specify an input file in the DBGSYM command line.

User Action: Review the documentation of DBGSYM in Chapter 4 to be sure you understand the command line format required. Retype the command line and include an input .STB file.

?DBGSYM-E-No symbols in .STB file

Explanation: There were no global symbol definitions in the input .STB file that you specified. Therefore, nothing can be put in the output .SYS file.

User Action: Only global symbols get placed in .STB files. Be sure any symbols you want included in the ST.SYS or STX.SYS file are defined as global symbols in your program.

?DBGSYM–F–Error writing .SYS file

Explanation: An output error occurred while writing the output ST[x].SYS file.

User Action: Be sure you have enough space for the file on the output volume.

?DBGSYM–F–Output device full

Explanation: There is not enough space on the output volume to hold the symbol table pseudodevice handler created by DBGSYM.

User Action: Delete some files on the output volume, squeeze the volume to consolidate free space, or specify a different output device.

?DBGSYM–F–Protected file already exists DEV:FILNAM.TYP

Explanation: A protected file of the same name as the DBGSYM output file specification already exists on the output volume. DBGSYM uses a default output filename of ST.SYS if you are using an unmapped monitor, STX.SYS if you are using a mapped monitor.

User Action: Unprotect the existing protected file if you want to replace that file, or explicitly specify an output filename in the DBGSYM command line that does not conflict with existing files on your output volume.

?DBGSYM–I–Default output filename is DEV:FILNAM.TYP

Explanation: DBGSYM uses a default output filename based on the type of monitor you are using. This message tells you what DBGSYM used. The possible defaults are ST.SYS for unmapped monitors, or STX.SYS for mapped monitors.

User Action: The message is informational; no action is required.

?DBGSYM–I–Number of symbols written was <val>

Explanation: This message tells you how many symbol definitions DBGSYM put in its output file.

User Action: The message is informational; no action is required.

?DBGSYM–U–Error reading proto blocks

Explanation: This error should not occur.

User Action: If you get this error, send an SPR to Digital. Include a copy of the program you are debugging, together with a detailed explanation of how to reproduce the situation that causes this error to occur.

?DBGSYM–U–Unable to locate proto blocks

Explanation: This error should not occur.

User Action: If you get this error, send an SPR to Digital. Include a copy of the program you are debugging, together with a detailed explanation of how to reproduce the situation that causes this error to occur.

?DBGSYM–U–Wrong version of RT–11

Explanation: You tried to use a version of DBGSYM supplied with one version of the RT–11 monitor with an earlier or later version of the RT–11 monitor.

User Action: Use DBGSYM and the rest of the pieces of the DBG–11 debugging package only with the version of RT–11 with which they were shipped.

?DBGSYM–W–File created; Protected file already exists DEV:FILNAM.TYP

Explanation: The output file created by DBGSYM on the output device has the same name as a protected file that already exists. This situation can occur if a foreground or system job creates a file of the same name as the file created by DBGSYM in the background.

User Action: List the device directory and find the location of both files relative to each other. The version created by DBGSYM will not be protected. Use the RENAME command to change the first occurrence of the filename in the directory to a different name.

?DBGSYM–W–Remove and reinstall ST handler

Explanation: Because the RT–11 monitor automatically locates and installs handlers only at bootstrap time, you must remove and install the ST handler before your monitor can use it. This warning message reminds you to do so.

User Action: Use the monitor commands REMOVE ST and INSTALL ST to ensure that your monitor uses the latest version of the ST handler.

?DBG–U–Internal error, aborting

Explanation: This error should not occur.

User Action: If you get this error, send an SPR to Digital. Include a copy of the program you are debugging, together with a detailed explanation of how to reproduce the situation that causes this error to occur.

?DBG–U–Stack overflow

Explanation: This error should not occur.

User Action: If you get this error, send an SPR to Digital. Include a copy of the program you are debugging, together with a detailed explanation of how to reproduce the situation that causes this error to occur.

?DBG–W–Bad expression

Explanation: The address or numeric expression you typed contains a syntax error.

User Action: Examine the expression, be sure it follows the rules for expressions given in Chapter 1, correct any errors, and enter the expression again.

?DBG–W–Can't proceed—<PC value>

Explanation: You issued a ;P or ;S command, but the last entry to DBG–11 occurred because of a fatal trap to 4 or trap to 10. DBG–11 displays the value of the PC where the trap occurred.

User Action: Check your program to determine the cause of the trap to 4 or trap to 10, and correct the error. You can restart the program using the addr;G command.

?DBG–W–Command parameters invalid

Explanation: The parameters you typed are invalid for the command you are attempting to execute.

User Action: Check the syntax of the command in Chapter 2. Be sure you understand what each parameter is supposed to be, and enter the command again with valid parameters. You also get this error if you attempt to delete the special symbols . or Q.

?DBG–W–Instruction syntax error

Explanation: You typed a MACRO–11 instruction incorrectly.

User Action: Correct the syntax of the instruction and enter it again.

?DBG–W–Invalid command

Explanation: You typed a command that DBG–11 does not recognize.

User Action: Type only valid DBG–11 commands. Chapter 2 lists all valid DBG–11 commands.

?DBG–W–Invalid internal register

Explanation: You tried to examine a location that lies outside the DBG–11 internal register table by using the circumflex (^) or LF command.

User Action: Examine DBG–11 internal registers individually by name, or do not exceed the register table limits when moving through the table with ^ and LF commands.

?DBG–W–Invalid memory reference

Explanation: You typed a memory address that DBG–11 cannot access. DBG–11 can access only memory that is currently mapped by the program you are debugging. This error applies only to XM systems.

User Action: Be sure you typed the address correctly. If you did, check to see why the address is not currently mapped by the program you are debugging. If you are attempting to access an address outside the program's address space, you will need to take special action, as described in Chapter 5.

?DBG–W–Line too long

Explanation: You typed a line longer than 80 characters.

User Action: DBG–11 input lines must be 80 characters or less. Break the line you are typing into parts, and enter the parts individually.

?DBG–W–Maximum nesting depth exceeded

Explanation: You entered an expression with too many levels of precedence using the angle bracket (<>) operators. DBG–11 allows a maximum nesting depth of six.

User Action: Reformat the expression so that it requires six or fewer pairs of brackets, or break the expression into parts and enter the parts individually.

?DBG–W–Maximum parameter count exceeded

Explanation: You typed more parameters than expected for a DBG–11 command.

User Action: Check the command syntax in Chapter 2, and enter the command correctly.

?DBG–W–No location open

Explanation: You attempted to store a value when DBG–11 was at its command prompt.

User Action: Be sure you have opened a location using one of the operators, such as /, [, or | , before you try to store a value.

?DBG–W–No room

Explanation: You tried to define more than 20 user symbols using the colon (:) operator. DBG–11 accepts a maximum of 20 user symbol definitions.

User Action: Use the ;K command to delete an existing symbol definition, then use the : operator to define a new symbol.

?DBG–W–Odd address

Explanation: You specified an odd address for an operation that must start at an even (word) address.

User Action: Retype the command using an even address, or use a command that allows odd byte addressing.

?DBG–W–Offset truncation error

Explanation: You typed a PDP–11 instruction, such as a branch or SOB, with an offset that exceeds the maximum of $+177_8$ or -200_8 words from the current location. For example, you typed BNE .+6000.

User Action: Be sure you do not exceed the offset limits when entering branch or SOB instructions.

?DBG–W–Undefined symbol—<symbol>

Explanation: You used a symbol that DBG–11 does not recognize. DBG–11 displays the symbol at the end of the message.

User Action: Be sure you typed the symbol correctly. If the symbol is undefined, define the symbol with the colon (:) symbol definition command before you attempt to use it as part of a DBG–11 command.

?SD–F–Handler must be named SD

Explanation: You tried to load the SD handler under a different name.

User Action: You can load the SD handler only under the name SD. Copy or rename the file to SD.SYS for unmapped systems, or SDX.SYS for mapped systems, and repeat the LOAD command.

?SD–F–Handler must be named ST

Explanation: You tried to load the ST handler under a different name.

User Action: You can load the ST handler only under the name ST. Copy or rename the file to ST.SYS for unmapped systems, or STX.SYS for mapped systems, and repeat the LOAD command.

?SD–W–BPT not found in code

Explanation: DBG–11 found at least one breakpoint/instruction pair in its breakpoint table that does not correspond to the program contents at that location.

The correspondence between a defined breakpoint and the corresponding instruction in a program is maintained in the DBG–11 breakpoint table as long as SD remains loaded in memory. If a program containing breakpoints is run through DBG–11 and another program is then run (without unloading SD), that correspondence is broken.

User Action: Clear all breakpoints and thereby reestablish correspondence by issuing the ;B command. You can then set whatever breakpoints you require to debug the program.

Appendix C
DBG–11 Command Summary

Command	Meaning	Section Reference
′	Byte-mode ASCII examine	2.5
\	Byte mode examine	2.5
|	Data mode examine	2.5
@	Indirect examine in I-space	2.6
#	Indirect examine in D-space	2.6
‘	Toggle between address spaces	2.6
~	Toggle between processor modes	2.6
/	Instruction mode examine	2.5
^	Previous location examine	2.6
%	Radix–50 examine	2.5
[	Symbolic data mode examine	2.5
"	Word-mode ASCII examine	2.5
_′	Byte-mode ASCII data entry prefix	1.3
_$	Hexadecimal data entry prefix	1.3
_%	Radix–50 data entry prefix	1.3
_"	Word-mode ASCII data entry prefix	1.3
+	Addition operator, unary plus	1.5
_/	Division operator	1.5
!	Inclusive OR Boolean operator	1.5
&	Logical AND Boolean operator	1.5
*	Multiplication operator	1.5
–	Subtraction operator, unary minus	1.5
< >	Brackets for expression nesting	1.5
=	Display value	2.7
:	Symbol definition	2.2
.	Value of current location counter	3.10

Command	Meaning		Section Reference
Command	**Meaning**		**Section Reference**
[*addr*][,*n*];B	Set and remove breakpoints		2.3
;D	Display breakpoint table		2.3
[*addr*];G	Begin program execution		2.4
[*qual*];M	Change processor mode and Address Space display; *qual* can be:		2.1
	D	Data address space display	
	I	Instruction space	
	K	Temporary Kernel mode	
	U	Temporary User mode	
	S	Temporary Supervisor mode	
		(No *qual*); restore Instruction space and initial processor mode	
[*qual*];R	Set output radix; *qual* can be:		2.1
	_BIN (binary)	_DEC (decimal)	
	_HEX (hexadecimal)	_OCT (octal)	
[*qual*];T	Set DBG–11 features; *qual* can be:		2.1
	_ESC or _NOESC	Support ESCAPE key as single-step operator	
	_K or _NOK	Kernel mode support for ~ operator	
	_S or _NOS	Supervisor mode support for ~ operator	
	_U or _NOU	User mode support for ~ operator	
	_PC or _PS	PC or PS displayed	
	_ADR or _NOADR	Symbolic address display	
	_REG or _NOREG	Graphics register display	
	_RUB or _NORUB	Scope/hardcopy rubout	
	_SYM or _NOSYM	Symbolic data display	
[*qual*];V	Set trap handling; *qual* can be:		2.1
	_T4	Pass traps to 4	
	_T10	Pass traps to 10	
symbol;K	Delete user symbol definition		2.2
[*val*];P	Proceed from breakpoint		2.3
[*val*];S	Single-step from breakpoint		2.3