IAS/RSX-11
ODT Reference Manual

Order No. DEC-11-0I0DA-B-D

RSX-11M Versions 3.0 and 3.1
RSX-11D Version 6.2
IAS Versions 1.1, 2.0 and 3.0

To order additional copies of this document, contact the Software Distribution Center, Digital Equipment Corporation, Maynard, Massachusetts 01754

digital equipment corporation · maynard, massachusetts

First Printing, December 1975
Revised: May 1977

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by DIGITAL or its affiliated companies.

The postage-prepaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist us in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DIGITAL	DECsystem-10	MASSBUS
DEC	DECtape	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EDUSYSTEM	PHA
UNIBUS	FLIP CHIP	RSTS
COMPUTER LABS	FOCAL	RSX
COMTEX	INDAC	TYPESET-8
DDT	LAB-8	TYPESET-11
DECCOMM	DECSYSTEM-20	TMS-11
ASSIST-11	RTS-8	ITPS-10
VAX	VMS	SBI
DECnet	IAS	

TABLE OF CONTENTS

CONTENTS (Cont.)

FIGURES

TABLE OF CONTENTS

CONTENTS (Cont.)

FIGURES

CONTENTS (Cont.)

TABLES

PREFACE

0.1 MANUAL OBJECTIVES AND READER ASSUMPTIONS

The intent of this manual is to enable its users to understand the debugging techniques provided by ODT. Readers are assumed to be familiar with the information contained in:

1. An appropriate PDP-11 Processor Handbook (i.e., PDP-11/05, /10, /35, /40, /45, or /70)

2. IAS/RSX-11 MACRO-11 Reference Manual

3. An appropriate Task Builder Reference Manual (i.e., for IAS, RSX-11D, or RSX-11M)

It is also important for readers of this manual to have gained an understanding of the terminal device providing the primary operator interface to the PDP-11 processor. For example, on some terminals an up-arrow may be present instead of a circumflex, and a back-arrow instead of an underline character.

In presenting ODT-11, a tutorial format has been adopted that includes explanatory text following actual ODT-11 command sequences. Thus, the flow of material throughout this manual is biased toward the user who is encountering ODT-11 for the first time. Also, those terms and expressions having particular significance in describing the functions and operations of ODT-11 are defined at appropriate points in the manual.

0.2 STRUCTURE OF THE DOCUMENT

Chapter 1 briefly describes the features of ODT-11 and the functions of the three major modules forming the program. Some of the important operational aspects of ODT-11 are described, and the common notation used for describing all ODT-11 command sequences is defined.

Chapter 2 presents the characters and symbols that form the vocabulary of ODT-11/user communications. The significance of these characters and symbols in a functional and operational sense is defined in this chapter.

Chapter 3 describes the composition and function of all the ODT-11 command sequences available to the user for debugging purposes. It is in this chapter that the explanatory text following the command sequence examples has been employed.

Chapter 4 presents the operating procedures for linking and executing ODT-11 with user programs.

Chapter 5 describes ODT-11's response to errors in the keyboard command sequences and lists the error codes resulting from hardware-detected errors during user program execution.

Chapter 6 describes the Trace program, a debugging aid that complements the functions of ODT.

Finally, Appendix A and Appendix B present details of interest in the Processor Status Word and the ODT-11 search algorithms, respectively.

0.3 ASSOCIATED DOCUMENTS

Other manuals closely allied to the purposes of this document are described briefly in the IAS, RSX-11D, or RSX-11M/RSX-11S Documentation Directory. The appropriate Documentation Directory defines the intended readership of each manual in the set for the host operating system, and provides a brief synopsis of each manual's contents.

CHAPTER 1

INTRODUCTION

ODT-11 operating under the Executive of the host system, aids the user in debugging assembled and linked object programs. Through keyboard interaction with ODT, the user can:

- Print the contents of any location in the object program for examination or alteration.
- Run all or any portion of an object program using the ODT breakpoint feature.
- Search the object program for words having a specified bit pattern.
- Search the object program for instructions that reference a specified address.
- Calculate offsets for PC-relative references and branch displacements within the object program.
- Fill a specified block of words or bytes with a designated value.
- List a specified block of words or bytes for examination.

1.1 ODT INTERNAL ORGANIZATION

Internally, ODT is modularized into independent subroutines that provide three major functions:

1. Command decoding
2. Command execution
3. Utility routines

The ODT command decoder routines interpret keyboard commands, check for command errors, save input parameters for use in command execution, and transfer control to the appropriate ODT command execution routines.

The command execution routines take the input parameters saved by the command decoder routines and call the ODT utility routines to execute the specified command. The command execution routines then exit to the object program or return control to the command decoder routines to await further keyboard input.

The utility routines, used by both the command decoder and command execution routines, save and restore the contents of registers and program locations and perform required keyboard input/output operations.

The flow of control and data between the ODT routines and the user object program is illustrated in Figure 1-1.

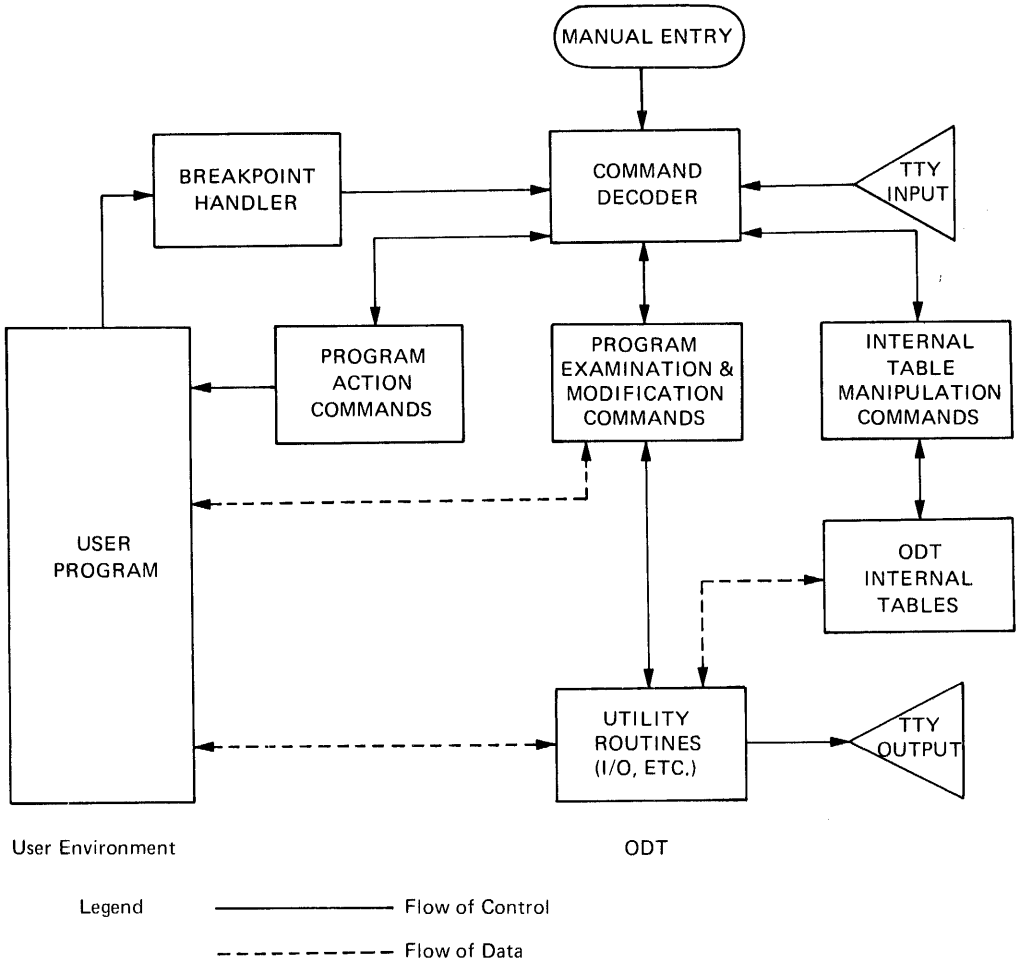

Figure 1-1
ODT Communications and Data Flow

1.2 OPERATIONAL DESCRIPTION

The following paragraphs describe the essential operational aspects of ODT in the user environment.

1.2.1 Linking ODT into the User Program

At task-build time, ODT is linked to the user object program by the Task Builder, thus incorporating ODT into the overall task image. If the task image is overlaid, ODT will be linked into the root segment of the task so that it will always be available for debugging use. The term "task" or "task image," as used throughout this manual, refers to that body of code resulting from Task Builder processing which can be loaded and executed directly. Once incorporated into the task image, ODT's usefulness as a debugging tool stems from its ability to establish selected breakpoints anywhere in the current memory-resident portion of a user program.

ODT runs as part of the user task and does not affect overall system operation. Furthermore, it is executed with the same privileges and at the same priority level as the task to which it is linked. Multiple tasks, each linked to its own copy of ODT, can be debugged at the same time, provided that sufficient memory and a terminal are available for each active task.

1.2.2 User Task Breakpoints

Breakpoints are user-selected locations at which execution is to be halted temporarily to permit interaction with the user task and ODT. Thus, ODT functions effectively as a monitor for the user task during a debugging session.

When the user task is executed under ODT control, the original contents of a breakpoint location are saved by ODT for later restoration. At the same time, ODT places a Breakpoint Trap instruction (octal op-code 000003) in the breakpoint location. Up to eight such breakpoint locations can be established in the user task at any given time. Later, during the active debugging session, program execution proceeds normally until a breakpoint location is reached. The Breakpoint Trap (BPT) instruction is then executed, causing control to be transferred to ODT. ODT then restores the original user instruction to the breakpoint location and awaits any valid command for a wide range of debugging operations.

Breakpoints must be set only in the first word of an instruction, since the BPT instruction must be executed to cause the trap action and the yielding of control to ODT. After the desired debugging operations associated with the current breakpoint have been performed, the user issues an appropriate ODT command to continue execution. ODT then resets all breakpoints to the BPT instruction (including the current breakpoint), and continues task execution.

The assembly listing of the program under examination should be at the terminal for reference during the debugging session. Minor corrections to the program may be made on-line. The program can then be run under ODT control to verify any changes made. Major program modifications, however, are more complex and should be noted on the assembly listing. In either case, all necessary changes should be incorporated into the task image through a subsequent edit, reassembly, and relinking of the program.

1.2.3 Relocation Registers

When MACRO-11 produces a relocatable object module, the base address of each program section in the module is relocatable 000000. The addresses of all program locations, as shown in the assembly listing, are therefore indicated relative to this base address. After the module is linked by the Task Builder to physical memory locations (for an unmapped system) or to virtual memory locations (for a mapped system), many values within the resulting object module will be biased by a constant whose value is the actual absolute base address of the object module after it has been relocated. This constant is called the relocation bias for the object module. Since a task may contain several relocated object modules (each with its own relocation bias), these biases must be continually subtracted from absolute addresses during debugging operations in order to associate relocated code with the assembly listing. ODT provides an automatic relocation facility for calculating the relocation bias of each object module in a relocatable program.

This facility is provided through eight relocation registers, numbered 0 through 7, each of which may be set by the user to the relocation bias of an object module at any given time during debugging operations.

The relocation bias of each program section in the task image is obtained by consulting the memory map produced by the Task Builder. Once set, a relocation register is used by ODT to relate the assembly listing to the relocated code.

For a more detailed description of the linking and relocation process, refer to the Task Builder Reference Manual for the host operating system.

1.3 EXPRESSING ODT COMMANDS AND FUNCTIONS

In debugging operations, many ODT commands and functions are expressed in keyboard sequences involving two or more characters. Such keyboard sequences, having several common elements, appear frequently throughout this manual. For consistency, the notation in Table 1-1 has been adopted to facilitate the expression and understanding of all user keyboard interaction with ODT. This notation is particularly significant in Chapters 2 and 3.

Table 1-1
Common Elements of Keyboard Sequences

Symbol	Meaning
a	Represents an argument that is used to define the address of a task image location.
n	Represents an octal integer in the range 0 through 7.
k	Represents an octal value up to six digits in length, with a maximum value of 177777(8), or an expression which reduces to such a value. If more than six digits are specified, ODT takes only the last six digits, truncated to the low-order 16 bits. The octal value may be preceded by a minus sign, in which case, the 2's complement of the value is taken by ODT.

The following examples illustrate how octal values (k) are interpreted by ODT:

Symbol k (Octal Value Typed)	ODT Interpretation
1	000001
-1	177777 (2's complement)
400	000400
-177730	000050 (2's complement)
1234567	034567 (Truncated to low-order 16 bits)

1.3.1 Forms of Address Expressions

An address expression is always evaluated by ODT as a 16-bit (six octal digit) value. This address expression is represented throughout this manual with the symbol a. An address expression may be typed in any one of three general forms, as described in Table 1-2.

Table 1-2
Forms of Address Expressions

Form of Expression	Format of Expression	Resulting Address Expression (a)
Type 1	k	The value of (a) is simply the value of k.
Type 2	n,k	The value of (a) is the value of k, plus the contents of relocation register n, where n designates any one of ODT's eight relocation registers. In this form, k is a relocatable address. If n is greater than 7, ODT types a question mark (?) character, ignores the current command, types the underline (_) prompting character, and awaits a valid command. ODT recognizes only octal numbers in defining address and other expressions. A decimal number (8 or 9) is illegal, causing a question mark (?) to be printed following the line in error.

Table 1-2 (Cont.)
Forms of Address Expressions

Form of Expression	Format of Expression	Resulting Address Expression (a)
Type 3	C or C,k or n,C or C,C	Whenever C is typed as an element of an ODT command, ODT replaces this character with the contents of the constant register (see $C, Table 3-1) and then evaluates the expression as a Type 2 address (n,k). In other words, the value in the constant register referenced by the C command has the same role as the n or k that it replaces in an address expression. For example, when C is used in place of n, the value in the constant register selects a relocation register for use in printing out task addresses (see section 3.1). In all cases where C is used in place of n, the value of C must be within the range 0 through 7. Whenever C is used in the place of k, the value in the constant register so referenced may be any 16-bit expression value. The commands used in accessing and modifying the contents of the constant register are described in detail in section 3.4 and Table 3-1.

1.3.2 Examples of Address Expressions

In the examples below of the three addressing forms, the following values are assumed:

n = Relocation register 3, containing the value 003400.

$C = Constant register, containing the value 000003.

Form of Expression	Address Expression (a) Keyboard Input	ODT Octal Interpretation
Type 1	5	000005
Type 1	-17	177761
Type 2	3,0	003400
Type 2	3,150	003550
Type 2	3,-1	003377

Form of Expression	Address Expression (a) Keyboard Input	ODT Octal Interpretation
Type 3	C	000003
Type 3	C,0	003400
Type 3	C,10	003410
Type 3	3,C	003403
Type 3	C,C+C	003406

NOTE

For simplicity, most address expression examples in this manual are Type 1; all three types, however, are equally acceptable to ODT.

CHAPTER 2

ODT CHARACTERS AND SYMBOLS

User commands to ODT are composed of the characters and symbols described below in Table 2-1. This table summarizes all the ODT commands in their available forms of use. For the purposes of this chapter, the reader should understand the notation presented in Table 1-1 and the basic concepts in the following paragraphs.

An open location is one whose contents have been printed by ODT for user examination. The value so printed is stored in a special register called the quantity register (see $Q, Table 3-1). The contents of an open location are available for change. A closed location is one whose contents are not immediately available for change.

Typing one of the commands listed below when it is preceded by an address expression opens the addressed location and prints its contents. The format of the printed output is a function of the command so issued. In other words, these are interpretive commands which may be used to print the contents of a specified location in any one or all of several formats (modes). When issued, these commands leave the current location open for further operations.

```
/     (Word mode octal)
\     (Byte mode octal)
"     (Word mode ASCII)
'     (Byte mode ASCII)
%     (Word mode Radix-50)
```

Typing one of the seven commands listed below closes the currently-open location; all but the carriage-return (<CR>) character cause another location to be opened. The location so opened depends on which of the other six commands is typed (see Table 2-1).

```
<CR> (Carriage Return)
<LF> (Line Feed)
^      (or up-arrow)
_    (or back-arrow)
@
>
<
```

In Table 2-1 and throughout this manual, the symbols <CR> and <LF> are used to represent the pressing of the carriage-return and line-feed keys, respectively.

As evident in the tables in this chapter, numerous ODT commands can be entered in any one of several forms. This flexibility stems from the

fact that ODT takes certain operational parameters and values from tables within itself in performing specified commands. These tables are described throughout this manual, particularly in Chapter 3, as "ODT internal registers" or "ODT internal locations." These terms in all cases refer to a block of memory within ODT which is reserved as a temporary storage area for the dynamic debugging variables essential to all ODT operations. These locations, consisting essentially of 17 sets of modifiable registers, are described in detail in section 3.4.

If required parameters for a given operation have already been stored in one or more of these internal locations as the result of a previous operation, a shorter form of a given command may suffice for a current operation, since ODT takes the current value of the relevant internal location(s) in executing a specified command. In the longer command forms, however, required values are an immediate part of the command. The various command forms are summarized briefly in the tables throughout Chapter 2. Chapter 3 treats the command forms in detail in conjunction with the discussions of ODT command sequences and functions.

Table 2-1
ODT Characters/Symbols

Format	Meaning
+ or space	Arithmetic operator. Sum the preceding argument and the following argument to form the current argument.
-	Arithmetic operator. Subtract the following argument from the preceding argument to form the current argument.
,	Relocation register operator. Use the preceding 1-digit octal value to reference one of ODT's eight relocation registers; the contents of this register and the value of the argument following the comma form the current argument. Thus, in ODT keyboard commands, a comma separates a relocation register specifier from an absolute value, the combination of which is normally used to specify relocatable address values in ODT command sequences.
*	Radix-50 operator. This command is used in forming Radix-50 arguments (see section 3.16.4).
.	Current location operator. Causes the address of the last explicitly-opened location to be used as the current address for ODT operations. This is the address assumed by the left angle bracket (<) command to return to the previous sequence of opened locations (see section 3.2.9). This address is also implied in the use of the slash (/), backslash (\), single quote ('), double quote ("), percent sign (%), and line-feed (<LF>) commands.

Table 2-1 (Cont.)
ODT Characters/Symbols

Format	Meaning
;	Argument identifier. Separates multiple arguments, allowing an address expression or ODT register value to be identified.
k	Represents any 6-digit octal value that is used as an argument in an ODT command. The symbol k also represents any expression which reduces to a 6-digit octal value. Expressions may include special arguments (such as $n, $x, C, or period) alone or in combination with the arithmetic operators (+, -, comma, or *). Expression constructions are terminated by typing a specific ODT command character or a semicolon (;).
n	Represents an octal integer in the range from 0 through 7. Decimal values are illegal in ODT and are flagged with a question mark (?) immediately following the illegal value.
a	Represents an argument whose special attribute is an address of a location. On input, any address value specified is interpreted by ODT as a 6-digit octal value, regardless of its length. Any value exceeding this limit is truncated to the low-order 16 bits. On output, ODT always prints an address value as six octal digits.
<CR> or k<CR>	Close the currently-open location and accept the next command. If <CR> is preceded by k, the value k replaces the contents of the currently-open location before it is closed.
<LF> or k<LF>	Close the currently-open location, open the next sequential location and print its contents. If <LF> is preceded by k, the value k replaces the contents of the currently-open location before it is closed.
^ or k^	Close the currently-open location, open the immediately-preceding location and print its contents. (The up-arrow appears on some keyboards and is used in place of the circumflex.) If ^ is preceded by k, the value k replaces the contents of the currently-open location before it is closed.

Table 2-1 (Cont.)
ODT Characters/Symbols

Format	Meaning
_ or k _	Interpret the contents of the currently-open location as a PC-relative offset and calculate the address of the next location to be opened; close the currently-open location, and open and print the contents of the new location thus evaluated. (The back-arrow appears on some keyboards and is used in place of the underline.) If _ is preceded by k, the value k replaces the contents of the currently-open location before it is closed.
@ or k@	Interpret the contents of the currently-open location as an absolute address, close the currently-open location, and open and print the contents of the absolute location thus evaluated. If @ is preceded by k, the value k replaces the contents of the currently-open location before it is closed.
> or k>	Interpret the low-order byte of the currently-open location as a relative branch offset and calculate the address of the next location to be opened; close the currently-open location and open and print the contents of the relative branch location thus evaluated. If > is preceded by k, the value k replaces the contents of the currently-open location before it is closed. The computation of this address is performed by taking the low-order byte of the currently-open location as a signed value, multiplying this value by 2, increasing the result by 2, and adding this sum to the address of the currently-open location.
< or k<	Close the currently-open location (opened by a _, @, or > command) and reopen the word location most recently opened by a /, <LF>, or ^. If the currently-open location was not opened by a _, @, or >, then < simply closes and reopens the current location. If < is preceded by k, the value k replaces the contents of the currently-open location before it is closed.
$n	Represents the address of one of eight user program general registers, where n is an octal digit identifying R0 through R7 (see section 3.3).

Table 2-1 (Cont.)
ODT Characters/Symbols

Format	Meaning
$x or $nx	Represents the address of one of 17 special ODT internal register sets (see section 3.4), where x is one of the following alphabetic characters, and n is an octal integer identifying a given location within a register set. These addressable register sets exist within ODT in the following order: S Processor Status register (hardware PS), which is saved by ODT when a breakpoint or user program fault occurs W Directive Status Word register for the user's task A Search argument register M Search mask register L Low memory limit register H High memory limit register C Constant register Q Quantity register F Format register X Reentry vector register nB Breakpoint address registers. nG Breakpoint proceed count registers. nI Breakpoint instruction registers. nR Relocation registers. nV SST vector registers. nE SST (synchronous system trap) stack contents registers. nD Device control LUN (logical unit number) registers.
C	Constant register operator. Represents the contents of special register $C (constant register).
Q	Quantity register operator. Represents the contents of special register $Q (quantity register).

Table 2-1 (Cont.)
ODT Characters/Symbols

Format	Meaning
" or a"	Word mode ASCII operator. Interpret and print the contents of the currently-open (or the last previously-opened) location as two ASCII characters, and store this word in the quantity register ($Q). If " is preceded by a, the value a is taken as the address of the location to be interpreted and printed.
' or a'	Byte mode ASCII operator. Interpret and print the contents of the currently-open (or the last previously-opened) location as one ASCII character, and store this byte in the quantity register ($Q). If ' is preceded by a, the value a is taken as the address of the location to be interpreted and printed.
% or a%	Word mode Radix-50 operator. Interpret and print the contents of the currently-open (or the last previously-opened) location as three Radix-50 characters, and store this word in the quantity register ($Q). If % is preceded by a, the value a is taken as the address of the location to be interpreted and printed.
/ or a/	Word mode octal operator. Reprint the contents of the last word location opened, and store this octal word in the quantity register ($Q). If / is preceded by a, the value a is taken as the address of a word location to be opened and printed.
\ or a\	Byte mode octal operator. Reprint the contents of the last byte location opened, and store this octal byte in the quantity register ($Q). If \ is preceded by a, the value a is taken as the address of a byte location to be opened and printed.
k=	Interpret and print expression value k as six octal digits and store this word in the quantity register ($Q).
8 or 9, RUBOUT, or CTRL/U	Cancel the current command and await a new command. The decimal value 8 or 9 is not a legal character and thus, when entered, causes ODT to ignore the current command. The RUBOUT and CTRL/U functions are not operational in RSX-11M, unless the terminal driver supports transparent read/write (a system generation option).
B	Remove all breakpoints from the user task.
nB	Remove the nth breakpoint from the user task.

Table 2-1 (Cont.)
ODT Characters/Symbols

Format	Meaning
a;B	Set the next available sequential breakpoint in the user task at address a.
a;nB	Set breakpoint n in the user task at address a.
K	Using the relocation register whose contents are equal to or closest to (but less than) the address of the currently-open location, compute the physical distance (in bytes) between the address of the currently-open location and the value contained in the selected relocation register; print this offset and store the value in the quantity register ($Q).
nK	Compute the physical distance (in bytes) between the address of the currently-open or the last-opened location and the value contained in relocation register n; print this offset and store the value in the quantity register ($Q).
a;nK	Compute the physical distance (in bytes) between address a and the value contained in relocation register n; print this offset and store the value in the quantity register ($Q).
F or kF	Fill memory locations within the address limits specified by the low memory limit register ($L) and the high memory limit register ($H) with the contents of the search argument register ($A). If F is preceded by k, the value k replaces the current contents of $A before initiating the fill operation.
G or aG	Set BPT instructions in or restore BPT instructions to all breakpoint locations in the task image, restore the Processor Status Word and user program registers, then commence execution at the address specified by the user Program Counter ($7). If G is preceded by a, the value a replaces the current contents of $7 before proceeding as described above.
aO or a;kO	Calculate and print the PC-relative offset and the 8-bit branch displacement from the currently-open location to address a; or calculate and print the PC-relative offset and the 8-bit branch displacement from the specified address a to the specified address k.

Table 2-1 (Cont.)
ODT Characters/Symbols

Format	Meaning
P or kP	Proceed with user program execution from the current breakpoint location and stop when the next breakpoint location is encountered or the end of the program is reached; or proceed with program execution from the current breakpoint location and stop at this breakpoint only after encountering it the number of times specified by integer k.
R	Set all relocation registers to -1, the highest address value, i.e., 177777(8).
nR	Set relocation register n to -1, the highest address value, i.e., 177777(8).
a;R	Set relocation register 0 to address value a.
a;nR	Set relocation register n to address value a.
S or nS	Execute one instruction and print the address of the next instruction to be executed; or execute n instructions and print the address of the next instruction to be executed.
W or kW or m;W or m;kW	Search memory between the address limits specified by the low memory limit register ($L) and the high memory limit register ($H) for words with bit patterns which match those of the search argument specified in the search argument register ($A). Compare each memory word and the search argument for equality under the mask specified in the search mask register ($M). When a match occurs, print the address of the matching location and its contents. If W is preceded by k, the value k replaces the current contents of $A before initiating the search. If W is preceded by m (identified by the semicolon that follows it), the value m replaces the current contents of $M before initiating the search. If W is preceded by both k and m, the current contents of $A and $M are replaced with the respective values so specified before initiating the search. NOTE Testing under a search mask ($M) results in a comparison of the memory word and the search argument only in those bit positions which correspond to the bits set to one (1) in the mask; all other bit positions are ignored in the search comparisons.

Table 2-1 (Cont.)
ODT Characters/Symbols

Format	Meaning
N or kN or m;N or m;kN	Search memory between the address limits specified by the low memory limit register ($L) and the high memory limit register ($H) for words with bit patterns which do not match those of the search argument specified in the search argument register ($A). This search is identical in form and function to the word (W) search described above, except that a test for inequality is performed.
E or kE or m;E or m;kE	Search memory between the address limits specified by the low memory limit register ($L) and the high memory limit register ($H). Examine these locations for references to the effective address specified in the search argument register ($A), as masked by the value specified in the search mask register ($M). (The mask should normally be set to 177777 for the E command.) Such references may be equal to, PC-relative to, or a branch displacement to the location specified in $A. If E is preceded by k, the value k replaces the current contents of $A before initiating the search. If E is preceded by m, the current contents of $M are replaced with the value m before initiating the search. If E is preceded by both k and m, the current contents of $A and $M are replaced with the respective values so specified before initiating the search.
L or kL or a;L or a;kL or n;a;kL	List all word or byte locations in the task between the address limits specified by the low memory limit register ($L) and the high memory limit register, using the listing device specified in the device control LUN register ($lD). If L is preceded by k, the value k replaces the current contents of $H before initiating the list operation. If L is preceded by a, the value a replaces the current contents of $L before initiating the list operation. If L is preceded by both a and k, the values a and k replace the current contents of $L and $H, respectively, before initiating the list operation. If L is also preceded by the value n, this value selects one of the device control LUN registers ($nD) containing the logical unit number of the device to be used in the list operation.

Table 2-1 (Cont.)
ODT Characters/Symbols

Format	Meaning
F or kF	Fill memory locations within the address limits specified by the low memory limit register ($L) and the high memory limit register ($H) with the contents of the search argument register ($A). If F is preceded by k, the value k replaces the current contents of the $A register before initiating the fill operation.
V	Enable ODT's handling of all SST vectors, and write the addresses of ODT's trap entry points into the table used by the SVDB$ Executive directive. (See Table 3-1 for a discussion of the SST vector registers and the $nV/ command.)
X	Exit from ODT and return control to the Executive of the host operating system.

CHAPTER 3

ODT COMMAND SEQUENCES AND FUNCTIONS

When ODT is initiated, its readiness to accept commands is indicated through the underline (_) prompting character (back-arrow on some terminals) at the left margin of the terminal. Most ODT commands can then be issued in response to this character. This chapter describes all the ODT command sequences and specific functions available to the user. Such keyboard interaction takes place using the characters and symbols described in the preceding chapter.

3.1 PRINTING TASK ADDRESSES

Normally, when ODT prints user program addresses (as with the commands <LF>, ^, _, @, <, and >), it attempts to print them in relative form (Type 2, see n;k, Table 1.2). If there is no relocation register containing a value equal to the user task address to be printed, ODT looks for the relocation register whose contents are closest to, but less than, the address. It then represents that address relative to the bias value contained in the register. However, if no relocation register fits this requirement, the user task address is printed in absolute form. Since the relocation registers are initialized to -1 (the highest address value), the user task addresses are initially printed in absolute form. If the contents of any relocation register are subsequently changed, it may then qualify for use in determining task addresses in relative form, depending on the ODT command issued.

For example, assume that relocation registers 1 and 2 contain the bias values 1000 and 1004, respectively, and that all other relocation registers contain much higher values. The following sequence might then occur:

```
_774/012345 <LF>          ;OPENS ABSOLUTE LOCATION 774.
000776 /024145 <LF>       ;OPENS ABSOLUTE LOCATION 776.
1,000000 /106421 <LF>     ;OPENS ABSOLUTE LOCATION 1000.
1,000002 /143164 <LF>     ;OPENS ABSOLUTE LOCATION 1002.
2,000000 /112713 <CR>     ;OPENS ABSOLUTE LOCATION 1004.
_
```

The printout format is controlled by the format register ($F). Normally, this register contains a default value of 0 (see $F, Table 3-1), in which case, ODT prints addresses relatively whenever possible, as noted above. The format register may be opened and changed to a positive, nonzero value, however; in this case, all user task addresses are printed in absolute form.

3.2 COMMANDS FOR OPENING, CHANGING, AND CLOSING LOCATIONS

An open location is one whose contents have been printed by ODT for examination and are thus available for change. A closed location is one whose contents are not immediately available for change.

The contents of an open location may be changed by typing the new value, followed by any ODT command which requires no argument (i.e., <CR>, <LF>, ^, _, @, >, or <). Note that leading zeros can be omitted by the user. Any command typed to open a location when another location is already open, closes the currently-open location before opening the new location.

3.2.1 Close Current Location: <CR> or k<CR>

When the <CR> key is typed while a location is open, that location is simply closed and no new location is opened. When <CR> is preceded by an argument k, that value replaces the current contents of the location before that location is closed. Typing the <CR> key has no effect on ODT when no location is open.

3.2.2 Open Next Sequential Location: <LF> or k<LF>

If the <LF> key is typed while a word location is open, i.e., if word mode is in effect, ODT closes that location and opens the next sequential word location, as shown below:

```
_1000/002340 <LF>          ;THE <LF> KEY IS TYPED AFTER THE
001002 /012740             ;PRINTOUT OF 002340, OPENING THE NEXT
                           ;LOCATION
```

In the example above, typing the <LF> key causes ODT to print the address and the contents of the next location automatically. The value 012740 is thus made available for examination and may be modified by typing a new value before issuing any command which closes the location.

If a byte location is currently open, i.e., if byte mode is in effect, typing the <LF> key opens the next sequential byte location.

Repetitive execution of the <LF> command causes ODT to open successive words or bytes, depending on the mode of the currently-open location.

3.2.3 Open Word Location: / or a/

A word location may be opened using the command form a/, where a is the address of the location to opened, as shown below:

```
_1000/012746               ;OPENS ABSOLUTE LOCATION 1000.
```

After the user types the slash (/), ODT automatically opens the addressed location and prints its 6-digit octal contents, making this value available for examination or change.

If the contents of an open location are not to be changed, the user may issue a <CR> command or any other command which closes an open location, without first typing an argument k. In the case of the <CR> command, ODT then closes the currently-open location, performs a carriage-return and line-feed action, and prints the prompting character (_) to indicate its readiness to accept another command, as shown below:

```
_1000/012746 <CR>          ;CLOSES LOCATION 1000 AND AWAITS
_                          ;NEXT COMMAND.
```

If the user desires to change the contents of an open location, he may do so by entering the new value before issuing a command which closes the location, as shown below:

```
_1000/012746 12345 <CR>    ;MODIFIES LOCATION 1000 AND
_                          ;AWAITS NEXT COMMAND.
```

The slash command can also be used without an address argument to reopen and reprint the contents of the word at the even-numbered location last opened, as indicated in the following example:

```
_1000/012746 12345 <CR>
_/012345                   ;OPENS AND DISPLAYS CONTENTS OF
                           ;PRECEDING WORD LOCATION.
```

This form of the slash command permits verification that a new value was entered correctly in a preceding location.

The slash command may also be used in conjunction with the <LF> command to open and print the contents of successive word locations. After opening a location in word mode, repetitive execution of the <LF> command displays consecutive task locations, as shown below:

```
_1002/000123 <LF>          ;REPETITIVE <LF> COMMAND DISPLAYS
001004 /123456 <LF>        ;CONSECUTIVE WORD LOCATIONS.
001006 /154321 <LF>
001010 /024351
```

In the sequence above, the <LF> command closes the currently-open location before opening the next location. The last <LF> command in a series of such commands leaves the current location open for any desired operation, as reflected above.

If an odd-numbered address is specified in opening a location, the slash command causes the location to be opened in byte mode. In this case, ODT commands then issued operate on byte locations and values, as indicated in the following sequence:

```
_1001/123 321 <CR>         ;LOCATION 1001 OPENED IN BYTE
_/321 <LF>                 ;MODE. SUBSEQUENT COMMANDS OPERATE
_001002 /021 <LF>          ;ON BYTE LOCATIONS.
_001003 /010 <LF>
_001004 /201
```

Word mode can then be restored, if desired, by closing the currently-open byte location and opening another location on an even address boundary, as shown below in the continuation of the preceding sequence:

```
_001004 /201 <CR>          ;<CR> CLOSES CURRENT LOCATION.
_1006/102054               ;NEXT LOCATION OPENED ON WORD
                           ;BOUNDARY, RESTORING WORD MODE.
```

3.2.4 Open Byte Location: \ or a\

As noted in the preceding section, ODT also operates on byte locations and values. The command form a\ is provided in ODT for simplifying the examination and modification of octally represented byte values, including those that fall on odd address boundaries. (On Teletypes, the backslash is typed by holding down the SHIFT key and typing L.) When this command form is used, the address value a, specified prior to the command, may be either odd or even. A byte location may be opened, as shown below:

```
_1001\002                ;LOCATION 1001 OPENED IN BYTE MODE.
```

After the user types the address of the byte location to be opened, followed by the backslash (\), ODT causes the contents of the addressed location to be printed as a 3-digit octal value.

If the contents of the byte location are not to be changed, the <CR> command, or any other command which closes an open location, may be issued without first typing an argument k. In the case of the <CR> command, ODT then closes the currently-open byte location, performs a carriage-return and line-feed, prints the prompting character (_), and awaits another command, as shown below:

```
_1001\002 <CR>           ;CLOSES LOCATION 1001 AND AWAITS
_                        ;NEXT COMMAND.
```

Should the user desire to change the contents of an open byte location, he may do so by entering the new value before issuing a command which closes the location, as reflected below:

```
_1001\002 10 <CR>        ;MODIFIES LOCATION 1001 AND AWAITS
_                        ;NEXT COMMAND.
```

Similar to the slash (/) command, the backslash character may be used without an address argument to reopen and reprint the contents of the byte at the location last opened. This use of the byte command is illustrated in the following sequence:

```
_1001\002 10 <CR>
_\010                    ;OPENS AND DISPLAYS CONTENTS OF THE
                         ;LAST OPENED BYTE LOCATION.
```

Thus, the alteration of a previously-opened byte location can be verified.

The <LF> command is also useful in conjunction with the backslash command, permitting successive byte locations in the task to be examined. After opening a location in byte mode, repetitive typing of the <LF> command displays consecutive byte values, as shown below:

```
_1003\004 <LF>           ;REPETITIVE <LF> COMMAND DISPLAYS
001004 \120 <LF>         ;CONSECUTIVE BYTE LOCATIONS.
001005 \203 <LF>
001006 \310
```

The <LF> command closes the currently-open location before opening the next location; the last such command issued, however, leaves the current location open for any desired operation, as shown above.

If a word location is currently open, typing the backslash command causes the word's low-order byte to be displayed without ODT leaving word mode:

```
_1010/000005 \005          ;DISPLAYS LOW-ORDER BYTE.
```

3.2.5 Open Preceding Location: ^ or k^

If the circumflex (or up-arrow) key is typed when a location is open, ODT closes the currently-open location and opens and prints the contents of the immediately preceding location. (On Teletypes, the circumflex is typed by holding down the SHIFT key, and typing N. The use of the circumflex is reflected in the following sequences:

```
_1000/002340 <CR>          ;LOCATION 1000 IS OPENED AND
                           ;EXAMINED.
_1002/012740 ^             ;LOCATION 1002 IS OPENED AND
                           ;EXAMINED, FOLLOWED BY CIRCUMFLEX
                           ;COMMAND.
001000 /002340             ;PRECEDING LOCATION IS OPENED AND
                           ;PRINTED.

_0,232/005046 <LF>         ;LOCATION 232, RELATIVE TO RELOCATION
                           ;REGISTER 0, IS OPENED AND EXAMINED.
0,000234 /012746 ^         ;NEXT LOCATION IS OPENED AND PRINTED,
                           ;FOLLOWED BY CIRCUMFLEX COMMAND.
0,000232 /005046           ;PRECEDING LOCATION IS OPENED AND
                           ;PRINTED.
```

If a byte location is currently open, issuing the circumflex command opens the preceding byte location and makes its contents available for examination or change, as shown below:

```
_1003\046 <LF>             ;LOCATION 1003 IS OPENED IN BYTE
                           ;MODE.
001004 \003 ^              ;NEXT BYTE LOCATION IS OPENED,
                           ;FOLLOWED BY CIRCUMFLEX COMMAND.
001003 \046                ;PRECEDING BYTE LOCATION IS OPENED
                           ;AND PRINTED.
```

If the command form k^ is used, the expression value k modifies the contents of the currently-open location before that location is closed, as shown in the following sequences:

```
_0,230/005406 <LF>         ;LOCATION 230, RELATIVE TO RELOCATION
                           ;REGISTER 0, IS OPENED.
0,000232 /000626 12345 ^   ;NEXT LOCATION IS OPENED AND
                           ;MODIFIED TO CONTAIN 012345. FOLLOWED
                           ;BY CIRCUMFLEX COMMAND.
0,000230 /005406 <LF>      ;PRECEDING LOCATION IS OPENED AND
                           ;PRINTED.
0,000232 /012345           ;CONTENTS OF MODIFIED LOCATION ARE
                           ;VERIFIED.
```

If a location is not currently open, typing the circumflex command opens and prints the contents of the last previously-opened word (or byte) location, as shown in the following sequence:

```
_0,236/000100 <CR>         ;RELOCATABLE ADDRESS 236 IS OPENED
                           ;AND CLOSED.
_^                         ;CIRCUMFLEX OPENS AND PRINTS LAST
0,000236 /000100           ;OPENED LOCATION.
```

3.2.6 Open PC-Relative Location: _ or k _

If the underline (or back-arrow) key is typed when a location is currently open, the contents of that location are added to its address+2 (PC value), yielding the address of the location to be opened. (On Teletypes, the underline is typed by holding down the SHIFT key, and typing O.) This computation is effectively a PC-relative reference. After this calculation, the current location is closed, the new location is opened, and its contents are printed, as shown in the following sequences:

```
_1000/000040 _          ;UNDERLINE OPENS PC-RELATIVE
001042 /052470          ;LOCATION AND PRINTS ITS CONTENTS.
```

If the currently-open location contains an odd value when the underline command is issued, the referenced location is not on a word boundary, and so is opened as a byte, as shown in the following sequences:

```
_0,232/012345 _         ;PC-RELATIVE ADDRESS IS CALCULATED,
0,012601 /041           ;CALCULATED ADDRESS IS THAT OF A
                        ;BYTE.

_0,422/000001 _         ;SAME AS ABOVE.
0,000425 /246
```

When the command form k_ is used, the expression value k modifies the contents of the currently-open location, and this new value is then used in the calculation of the PC-relative address of the location to be opened and printed, as shown in the following sequences:

```
0,232/012345 123456 _   ;LOCATION 232, RELATIVE TO
                        ;RELOCATION REGISTER 0, IS
                        ;OPENED AND MODIFIED TO
                        ;CONTAIN 123456, FOLLOWED BY
                        ;UNDERLINE COMMAND.
0,123712 /020301        ;PC-RELATIVE LOCATION IS
                        ;OPENED AND PRINTED.
```

3.2.7 Open Absolute Location: @ or k@

The @ sign typed when there is a currently-open location takes the contents of that location as the address of the next location to be opened. (On Teletypes, the @ sign is typed by holding down the SHIFT key, and typing P.) The currently-open location is then closed, and the new location is opened. The following sequences reflect the use of this command:

```
_1006/001024 @          ;USES CONTENTS OF CURRENT
                        ;LOCATION AS ADDRESS OF NEXT LOCATION
                        ;TO OPEN.
001024 /000500          ;LOCATION 1024 IS OPENED AND ITS
                        ;CONTENTS PRINTED.

_100;R                  ;SETS RELOCATION REGISTER 0 TO 100(8)

_0,232/000456 @         ;SAME ACTION AS ABOVE, IN
                        ;RELOCATABLE FORMAT.
0,000356 /005046        ;LOCATION 456 OPENED AND
                        ;CONTENTS PRINTED.
```

If the command form k@ is employed, the expression value k modifies the contents of the currently-open location, and this new value is then taken as the address of the next location to be opened, as shown in the following sequences:

```
_1006/001024 2100@          ;LOCATION 1006 IS MODIFIED TO
002100 /177774              ;CONTAIN 002100.  THIS VALUE IS
                            ;THEN USED TO OPEN NEXT LOCATION.

_370;R                      ;SET RELOCATION REGISTER 0 TO
                            ;BIAS VALUE OF 370(8) FOR MODULE.
_0,600/012345 12746 @       ;CONTENTS OF RELOCATABLE ADDRESS 600
                            ;ARE MODIFIED TO CONTAIN 012746. THIS
                            ;VALUE IS THEN USED TO CALCULATE
                            ;ADDRESS OF NEXT LOCATION TO BE
                            ;OPENED.
0,012356 /027117            ;EVALUATED ADDRESS IS OPENED AND ITS
                            ;CONTENTS PRINTED.
```

In the example above, note that the relocatable address of the next location opened (0,012356) is represented relative to the bias value 370(8) contained in relocation register 0. The accuracy of this calculation is verified by adding the value 370(8) to the value 012356(8), yielding the sum 012746(8).

3.2.8 Open Relative Branch Offset Location: > or k>

When the right-angle bracket (>) command is issued for an open location, ODT takes the low-order byte of this location to calculate a relative branch offset in determining the address of the next location to be opened. The current location is closed when this command is executed.

The relative branch offset, i.e., the address of the next location to be opened, is calculated as follows:

1. Take the low-order byte of the currently-open location as a signed value.
2. Multiply this value by 2.
3. Add the result of Step 2 to the address of the currently-open location+2 (PC value).

The examples below show the use of the relative branch offset command:

```
_1032/000407 >              ;TAKES THE LOW-ORDER BYTE OF THE
001052 /001456              ;CURRENT LOCATION AS RELATIVE
                            ;BRANCH OFFSET TO OPEN NEXT
                            ;LOCATION.

_0,66/005046 >              ;SAME OPERATION AS ABOVE, EXCEPT
0,000204 /000601            ;RELOCATABLE ADDRESS VALUES ARE
                            ;USED.
```

If the command form k> is used, the expression value k modifies the contents of the currently-open location, and the low-order byte of this new value is then used in the calculation of the relative branch offset location, as shown in the following sequences:

```
_1032/000407 301>          ;LOCATION 1032 IS MODIFIED TO
000636 /000010             ;CONTAIN 000301. LOW-ORDER
                           ;BYTE OF THIS NEW VALUE IS
                           ;THEN USED IN DETERMINING RELATIVE
                           ;BRANCH LOCATION.

_0,232/000456 134561 >     ;RELOCATABLE LOCATION 232 IS
0,000576 /002340           ;MODIFIED TO CONTAIN 134561. LOW-
                           ;ORDER BYTE OF THIS NEW VALUE IS
                           ;THEN USED IN CALCULATING RELATIVE
                           ;BRANCH LOCATION.
```

Note in the first example above illustrating the k> command form, that the byte value 301 is interpreted by ODT as a negative value (the high-order bit in this byte is 1). Therefore, a negative branch offset results, causing location 636 (a lower physical address) to be opened and its contents printed.

3.2.9 Return to Interrupted Sequence: < or k<

The left-angle bracket command (<) can be used immediately after the issuance of any of the following address calculation commands:

1. Open PC-relative location (_) (see section 3.2.6).
2. Open absolute location (@) (see section 3.2.7).
3. Open relative branch offset location (>) (see section 3.2.8).

The user can issue any of these three commands in any order after explicitly opening a word location with a /, <LF>, or ^. The /, <LF>, and ^ commands are explicit in that they open a specified location, the word following a currently-open location, or the word preceding a currently-open location, respectively. They do not depend on the contents of the open location as the address calculation commands mentioned above do.

The left angle bracket command (<) causes ODT to close the currently-open location, and reopen the word location most recently opened by a /, <LF>, or ^. If the currently-open location was not opened by a _, @, or > command, then < merely closes and reopens the current location itself.

The effect of the < command is reflected in the following sequences:

```
_10000;R                   ;SETS RELOCATION REGISTER 0 TO 10000
_0,1030/000174 <LF>        ;OPENS RELOCATABLE 1030; <LF> OPENS
                           ;NEXT WORD
0,001032 /000200 @         ;@ OPENS LOCATION 200 IN ABSOLUTE
                           ;FORMAT BECAUSE NO RELOCATION
                           ;REGISTER'S CONTENTS ARE LESS THAN
                           ;OR EQUAL TO 200
000200 /007020 @           ;@ OPENS 7020 IN ABSOLUTE FORMAT (NO
                           ;RELOC. REG. CONTENTS < OR = TO 7020)
007020 /000000 <           ;< REOPENS RELOCATABLE 1032
0,001032 /000200
```

```
_1036/021346 ^          ;OPENS 1036; ^ OPENS PRECEDING WORD
1034 /101036 _          ;OPENS PC RELATIVE LOCATION.
102074 /000000 @        ;OPENS ABSOLUTE LOCATION.
000000 /000000 >        ;OPENS RELATIVE BRANCH OFFSET
                         ;LOCATION.
000002 /000102 <        ;RETURNS TO LAST EXPLICITLY-OPENED
                         ;LOCATION.
001034 /101036          ;OPENS AND PRINTS CONTENTS OF
                         ;LOCATION 1034.
```

The contents of any location opened by one of the three address calculation commands may be altered, if desired, before issuing a command which closes that location. This option is illustrated in the following sequences:

```
_1064/000276 @
000276 /000340 336_
000636 /000000 302>
000444 /026474 474@
000474 /015325 <
001064 /000276
```

3.3 ACCESSING USER PROGRAM GENERAL REGISTERS: $n

ODT has a set of fixed locations which are used to store the current values of the user program's general registers when a breakpoint occurs. Thus, the current state of the user program is preserved so that task execution can be resumed normally when control is returned to the user through the execution of the G (Go) or P (Proceed) commands (see section 3.6). These registers, numbered 0 through 7, can be examined with a command of the following form:

```
_$n/
```

where n represents an octal integer representing the desired register. When the slash is typed, the contents of the specified register are automatically printed by ODT, making this value available for examination or change. The user can change the contents of a general register by issuing a command of the form $n/a <CR>, where n represents the octal register specifier, and a represents the new value to be entered. The following examples show how the user program's general registers are opened and modified:

```
_$0/000033 <CR>          ;REGISTER 0 IS EXAMINED AND CLOSED.
_$4/000474 464 <CR>      ;REGISTER 4 IS OPENED, MODIFIED TO
_                        ;CONTAIN 000464, AND CLOSED.
```

Any register modification just completed, as shown in the $4 line above, can be verified by typing a slash in response to ODT's prompting character. Thus, the continuation sequence is:

```
_/000464                 ;PRINTS THE CONTENTS OF THE
                         ;PREVIOUSLY-OPEN LOCATION.
```

Note that the <LF>, ^, _, or @ command may be used in connection with a user program general register when that register is open.

3.4 ACCESSING SPECIAL ODT INTERNAL REGISTERS: $x or $nx

ODT contains a number of fixed locations which are used as registers for temporary storage of values essential to debugging operations. In addition, these registers provide a mechanism through which the current state of the user program is preserved when a breakpoint occurs, saving and restoring such values as the Processor Status Word and the user program stack pointer during debugging operations. These internal registers, which are accessible to the user in the same manner as any location within the task image, are described in detail in Table 3-1.

The command form $x/ is used to access an ODT internal register, where x represents the alphabetic register identifier. The processor status register, for example, can be accessed with the following command:

```
_$S/000011                ;THE COMMAND $S/ OPENS THE STATUS
                          ;REGISTER AND PRINTS ITS CONTENTS.
```

In response to the $S/ command, ODT prints the 16-bit Processor Status Word in 6-digit octal format. If desired, any new value can be entered into the register, followed by a command which closes the register.

The command form $nx/ is used to access an internal register set consisting of several separate locations, where x represents the alphabetic register identifier (as above), and n represents an octal integer referencing a particular location within the register set. Relocation register 7, for example, can be accessed with the following command:

```
_$7R/000040               ;THE COMMAND $7R/ OPENS RELOCATION
                          ;REGISTER 7 AND PRINTS ITS CONTENTS.
```

The contents of this register may also be modified, if desired, by entering the new value and issuing a command which closes the location.

All the ODT internal registers described in Table 3-1 can be accessed and modified in similar fashion.

Note in Table 3-1, that the value a, k, or n may appear in connection with the generic command forms used to open the ODT internal registers (e.g., $C/a, $F/n, $A/k, etc.). These symbols represent new values that may be entered into the register if the current value displayed upon opening the register is not desired. Also, these symbols may be used in connection with other ODT commands (described in Table 3-1 and elsewhere throughout this manual) which automatically enter parameters into specific internal registers without overtly opening the locations to which these parameters apply. The symbols a, k, and n are described in the context of the operation being performed and, in all cases, represent the specific parameters or arguments defined by the user to serve current debugging purposes.

Table 3-1
Internal Register Access/Modification Commands

Command	Function
$S	Processor status register. Contains the Processor Status Word after the execution of the last user program instruction prior to the occurrence of the breakpoint. Although this register is accessible to the user through the $S/ command, it is set by the Executive of the host operating system and normally should not be changed by the user. This register provides the mechanism through which the Processor Status Word may be examined during a debugging session. For a detailed description of the Processor Status Word, refer to Appendix A.
$W	Directive Status Word register. Contains the Directive Status Word (DSW) of the user's task. This word contains information on either the success, or specific cause of rejection of the most recently executed directive. The DSW is saved across breakpoints, and is available for examination and modification through the $W/ command. (This status word is discussed more fully in the Executive Reference Manual of the host system.)
$C	Constant register. Set by the user to any 16-bit value representing an address (a) or an expression value (k) through the $C/a <CR> or the $C/k <CR> commands, respectively. Both command forms are identical in function and are shown to represent the possible uses of the values so entered. For example, any value entered in the constant register may be used by typing C as an argument in an ODT command. The possible uses of this value are illustrated as a Type 3 address in Table 1-2. The constant register is described in further detail in section 3.16.2.
$F	Format register. Set by the user to an octal value (n) through the $F/n <CR> command. When set to zero (the default value), all user task addresses are printed by ODT in relative form when appropriate (as described in section 3.1). All other values of n cause user task addresses to be printed in absolute form.
$M	Search mask register. Set by the user to a word or byte search mask value through the $M/m <CR> command. A mask value may also be set in those commands which initiate search operations (see sections 3.8.1 through 3.8.3). This register is initialized by ODT to minus one (-1), 177777(8). Thus, unless otherwise modified, all bit positions in the search argument (see $A below) and the memory word/byte will be compared in a search operation.

Table 3-1 (Cont.)
Internal Register Access/Modification Commands

Command	Function
$A	Search argument register. Set by the user to a word or byte search argument (k) through the $A/k <CR> or the $A\k <CR> commands, respectively. A search argument may also be set in those commands which initiate search operations (see sections 3.8.1 through 3.8.3).
$L	Low memory limit register. Set by the user to an address value (a) through the $L/a <CR> command, establishing the lower memory limit for all ODT search, list, and fill operations which reference this register. This register is initialized by ODT to zero (0). Either absolute or relocatable address values may be entered into this register.
$H	High memory limit register. Set by the user to an address value (a) through the $H/a <CR> command, establishing the upper memory limit for all ODT search, list, and fill operations which reference this register. This register is also initialized by ODT to zero (0). As with the $L register, either absolute or relocatable address values may also be entered into this register.
$Q	Quantity register. Set automatically by ODT to the last value printed on the console. This register is described in further detail in section 3.16.3.
$X	Reentry vector register. This register is normally set to one (1) by the user through the $X/n <CR> command when an initial debugging pass has been completed, thus allowing the user program to be executed directly without again reentering ODT. If set to one (1), the task then starts at its normal entry-point address or at the address specified by the user in general register 7 (see section 3.3), rather than at ODT's starting address. The use of this register is described in further detail in section 3.17.
$nR	Relocation register n. One of eight (n) register locations which may be set by the user to a value (a) through the a;nR command or the $nR/a <CR> command. A value set in a specified register location represents the relocation bias of a given relocatable object module of interest during the debugging session. Once set, the contents of a given location enable ODT to print user task addresses relative to a base address. Both positive and negative offsets (biases) can be calculated by ODT using these register locations (see section 3.11). Also, when the user opens a given location in a relocatable module, the value in the associated relocation register enables ODT to calculate the relocated address of the user task location (see section 3.12). Thus, relocatable code in the assembly listing can easily be associated with the addresses of relocated code during the debugging session. This register is initialized by ODT to minus one (-1), 177777(8).

Table 3-1 (Cont.)
Internal Register Access/Modification Commands

Command	Function
$nB	Breakpoint address register n. One of eight (n) locations which may be set by the user to an address value (a) through the a;B or a;nB commands (see section 3.5) or the $nB/a <CR> command. These user-specified addresses identify breakpoint addresses in the user task whose contents are to be swapped with BPT instructions in an associated breakpoint instruction register (see $nI below). This swapping process occurs upon execution of the G command when the debugging session is initiated or upon execution of the P command when task execution is resumed from a breakpoint location (see section 3.6). The breakpoint address registers are described in further detail in section 3.5.
$nD	Device control LUN register n. One of three (n) locations which may be set by the user to a value (k) through the $nD/k <CR> command, where the values defined for n and k have the following significance: Value n — Value k 0 - User terminal device logical unit number (see Note below). The value of k in this location ($0D) is normally 000007(8). 1 - Console listing device logical unit number (see Note below). The value of k in this location ($1D) is normally 000010(8). 2 - QIO event flag number - The value of k in this location ($2D) is normally a default value of 000034(8). NOTE The user terminal device LUN (TI:) and the console list device LUN (CL:) are assigned by the Task Builder, which examines the UNITS= Keyword option (having a default value of 6). The LUN n+1 is assigned to the user terminal device, and the LUN n+2 is assigned to the console list device, where n is the default value 6 or the value used as the argument to the UNITS= Keyword option. Thus, $0D normally contains 000007(8), and $1D normally contains 000010(8).
$nI	Breakpoint instruction register n. One of eight (n) locations which may be set by ODT to contain BPT instructions. These BPT instructions are swapped with user program instructions at the breakpoint locations

Table 3-1 (Cont.)
Internal Register Access/Modification Commands

Command	Function
$nI (Cont.)	defined through the breakpoint address register (see $nB above). This swapping process occurs upon execution of the G command when the debugging session is initiated or upon execution of the G or P command when task execution is resumed from a breakpoint location (see section 3.6). This register is initialized by ODT to BPT instructions, i.e., op code 000003(8).
$nG	Breakpoint proceed count register n. One of eight (n) locations which may be set by the user to a proceed count value (k) through the kP command or the $nG/k <CR> command. The proceed count value set in each of these locations is associated with a given breakpoint address, as defined by the user in the breakpoint address register (see $nB above). It is sometimes useful, for example, to set a breakpoint in a loop. After the breakpoint occurs, the user may type the kP command (see section 3.6) to resume execution. The program then executes through the loop k number of times before again recognizing the breakpoint. Each time the breakpoint location is encountered, the proceed count value in the associated register location is decremented. When the count reaches zero (0), the breakpoint is again recognized, suspending user task execution and transferring control to ODT for any desired debugging operations. This register is initialized by ODT to one (1).
$nV	SST vector register n. One of eight (n) locations that contain entry-point addresses of ODT routines for handling synchronous system traps. These traps occur during user program execution as a result of certain hardware-detected errors and programming conditions. The value n refers to a given SST vector address location, as listed below. Each of these locations contains a pointer to an ODT error-handling routine which evaluates the SST error condition and prints out an appropriate console error message (see section 5.2). If a user program and ODT both have SST vectors enabled for a condition that then occurs, ODT receives the trap. As released, ODT has seven vectors enabled. Only vector number 6 (TRAP instruction executed) is disabled. ODT's handling of vector 6 can be enabled by the user through the V command. The V command enables ODT's handling of all SST vectors, and writes the addresses of ODT's trap entry points into the table used by the SVDB$ Executive directive (see the Executive Reference Manual of the host system).
$nV	Value n — SST Vector Register 0 - Odd address reference in word instruction. (Also, on some PDP-11 processors (e.g., PDP-11/45), the execution of an illegal instruction is trapped here rather than through SST vector 4.)

Table 3-1 (Cont.)
Internal Register Access/Modification Commands

Command	Function
$nV (Cont.)	Value n — SST Vector Register 1 - Memory protect violation (segment fault). 2 - T-bit trap or BPT instruction executed. 3 - IOT instruction executed. 4 - Reserved or illegal instruction executed. 5 - Non-IAS/RSX-11 EMT instruction executed. 6 - TRAP instruction executed. 7 - PDP-11/40 floating point exception error. These vector locations are accessible to the user through the $nV/ command in a manner similar to any other ODT internal register, where the value n selects one of the eight locations listed above. Normally, these ODT locations are not manipulated by the user. However, the user has the option of handling some or all of the SST traps (except the BPT instruction) that may occur during program execution. In this case, the user may set the corresponding SST vector location in ODT to zero (0), thereby causing the user program to trap to an SST processing routine within itself, i.e., the trap will reference the user SST vector address directly without invoking ODT control. Such an option obviously assumes that the user program contains appropriate routines for handling SST error conditions.
$nE	SST stack contents register n. One of three (n) locations (where n is equal to 0, 1, or 2) into which the top three items on the user program stack are placed when a synchronous system trap occurs (see $nV above). These stack items have different values, depending on the type of trap taken. (Consult the Executive Reference Manual of the host operating system for a discussion of synchronous system traps.) These locations, containing user task information of interest following an SST interrupt, can be examined through the $nE/ command, where n selects one of the three register locations, as noted above.

3.5 TASK BREAKPOINT COMMANDS: a;B, a;nB, B, or nB

Breakpoints must be set in the first word of an instruction. When set through one of the commands described below, ODT places the address of each breakpoint location in an associated breakpoint address register (see $nB, Table 3-1).

When a G or P command is issued to initiate or resume task execution (see section 3.6), ODT swaps the user instructions in the specified breakpoint locations with BPT instructions in the breakpoint instruction registers (see $nI, Table 3-1). Later, as a breakpoint location is encountered during task execution, the BPT instruction in that location causes control to be transferred to ODT at the address contained in SST vector register 2 (see $nV, Table 3-1).

The BPT instruction, in connection with the PDP-11 hardware facilities, thus serves as a simple and efficient mechanism for calling a debugging aid. As the final consequence of this software-generated trap, ODT suspends task execution and restores the original user instruction to the breakpoint location. Since ODT retains control, the user can then perform any desired debugging operations from the current breakpoint location.

It is important to note that the original user instruction is always restored to the current breakpoint location when the breakpoint trap occurs, ensuring that all user task instructions will be executed during the course of a debugging session if the program is allowed to proceed to completion.

It should also be noted that debugging overlaid tasks presents special considerations in setting and maintaining breakpoints. Since breakpoints established for the current segment do not remain valid for a subsequent segment, all breakpoints should be removed from the current segment before a new segment is loaded. Otherwise, task instructions saved from breakpoint locations in the current segment will later be swapped into a new segment, thus implanting invalid instructions and corrupting the program. It is recommended, therefore, that tasks be non-overlaid for debugging.

Up to eight breakpoints, numbered 0 through 7, can be set at any given time. The command which accomplishes this action takes the form:

```
a;B
```

where a represents the address of the breakpoint location. Repetitive execution of this command can be used to establish all eight (0 through 7) breakpoint locations, since each of the addresses so defined is entered sequentially into the breakpoint address registers (see $nB, Table 3-1).

Specific breakpoints can be set or changed through the following command:

```
a;nB
```

where a represents the address of the desired breakpoint location, and n represents one of eight (0 through 7) such specific breakpoints. The examples below illustrate how breakpoints are set and changed:

```
_B                      ;CLEAR ALL BREAKPOINTS.
_1020;B                 ;SET BREAKPOINT 0.
_1030;B                 ;SET BREAKPOINT 1.
_1040;B                 ;SET BREAKPOINT 2.
_1032;1B                ;RESET BREAKPOINT 1.
_
```

The B command typed alone removes all breakpoints in the user task, as shown in the initial command of the preceding sequence. The command form nB removes only the specified breakpoint, as shown in the last command below, where n represents any one of the eight (0 through 7) breakpoints currently in effect. The following sequence shows how breakpoints are set, changed, and removed:

```
_1020;0B                ;SET BREAKPOINT 0 AT LOCATION 1020.
_1030;1B                ;SET BREAKPOINT 1 AT LOCATION 1030.
_1064;2B                ;SET BREAKPOINT 2 AT LOCATION 1064.
_1220;3B                ;SET BREAKPOINT 3 AT LOCATION 1220.
_1324;4B                ;SET BREAKPOINT 4 AT LOCATION 1324.
_1032;1B                ;CHANGE BREAKPOINT 1 TO LOCATION
                        ;1032.
_3B                     ;REMOVE BREAKPOINT 3.
```

The command form $nB/ references the address of the nth breakpoint, as stored in the nth breakpoint address register (see $nB, Table 3-1). Assuming that the previous command sequence is still in effect, the user may reference breakpoint 0 through the following command.

```
_$0B/001020             ;OPENS BREAKPOINT REGISTER 0 SET IN
                        ;PREVIOUS SEQUENCE.
```

The command $nB/ thus opens breakpoint address register n, causing its contents to be printed. Continuing with the current command sequence, the user may examine the contents of successive breakpoint address registers by repetitively typing the <LF> key, as shown below:

```
_$0B/001020 <LF>        ;OPENS BREAKPOINT 0.
$1B /001032 <LF>        ;BREAKPOINT 1, OPENED BY LINE FEED.
$2B /001064 <LF>        ;BREAKPOINT 2, OPENED BY LINE FEED.
$3B /000364 <CR>        ;BREAKPOINT 3, OPENED BY LINE FEED.
_
```

All eight breakpoint address registers can be examined in this manner.

3.6 PROGRAM EXECUTION COMMANDS: G or aG and P or kP

Two general command forms are available for running the user task: the G (Go) command and the P (Proceed) command. An alternate form of each command is also available which takes an argument, as described in the following paragraphs. The G command exists primarily to begin program execution at the user task's transfer address, and the P command is used to resume program execution at the next logical instruction after a breakpoint has occurred.

When the G command is executed, the BPT instructions in the breakpoint instruction registers (see $nI, Table 3-1) are swapped with the user instructions in the task image locations defined in the breakpoint address registers (see $nB, Table 3-1). Task execution then begins at the program's entry-point address, i.e., the address contained in the user task's program counter (PC).

If an address argument a is specified with the G command, the swapping of BPT instructions and user task instructions at breakpoint locations occurs as described above, but program execution begins at the specified task address. For example, the command:

```
_1000G
```

initiates execution at task location 1000. Note that any address argument used with the G command must specify an even address, i.e., a word location boundary. The program runs until a breakpoint is encountered or until the end of the program is reached. (A program that is in an infinite loop must be aborted and then restarted.)

When a breakpoint is encountered, the contents of the user task general registers are stored in ODT locations $0 through $7 (see section 3.3). In addition, task execution is suspended, the user program instructions are restored to all breakpoint locations, and ODT prints a console message indicating the occurrence of a breakpoint. This message takes the following form:

```
nB:a
```

where n represents the breakpoint number, and a represents the address of the breakpoint location. The prompting character (_) then appears on the following line to indicate ODT's readiness to accept any valid command, as shown in the sequences below:

```
_1010;3B                ;BREAKPOINT 3 IS SET AT LOCATION
                        ;1010.
_1000G                  ;EXECUTION IS STARTED AT LOCATION
                        ;1000.
3B:001010               ;EXECUTION STOPS AT BREAKPOINT 3, AND
_                       ;THE ADDRESS OF THE BREAKPOINT
                        ;LOCATION IS PRINTED.
```

To continue program execution from a breakpoint location, the user types either the G or P command. Thus, the G or P command can be used without an argument, if program execution is to be resumed after a breakpoint occurs. If program execution is interrupted by the occurrence of an error of the type BE, IO, EM, TR, or FP (see section 5.2), use of the G or P command causes execution to resume at the word location following the error location, rather than at the error location itself.

The P command is illegal if a breakpoint has not yet occurred. If this command is issued before a breakpoint location has been encountered, ODT responds with a question mark (?) on the line in error and prompts with the underline character (or back-arrow) on the following line, indicating that the user must issue the G (Go) command to begin or resume program execution.

If the task has not yet been run, the G command starts execution at the program's entry-point (transfer) address; otherwise, the G command causes program execution to resume immediately following the last logical instruction executed. In this case, the G command has the same effect as the P command when resuming execution from a breakpoint location.

When the G or P command is executed, the user general registers (see section 3.3) are restored to their original (pre-breakpoint) values, the BPT instructions are swapped with the user instructions referenced by the breakpoint address registers, and control is returned to the user program.

When a breakpoint is set within a loop, it may be desirable to allow the program to execute through the loop a specified number of times before recognizing the breakpoint. This can be done through an alternate form of the P command which takes an argument k, where k is an octal integer specifying the number of times the breakpoint is to be encountered before program execution is suspended. If the P command is issued without an argument, execution continues only to the next breakpoint (or to the end of the program).

The breakpoint proceed count is associated only with that breakpoint which has most recently occurred, i.e., a different proceed count is associated with each breakpoint, determining the number of times each breakpoint is to be encountered before program execution is suspended, as shown in the example below:

```
3B:001010                ;EXECUTION IS HALTED AT BREAKPOINT 3.
_1250;5B                 ;BREAKPOINT 5 IS SET AT LOCATION
                         ;1250.
_7P                      ;EXECUTION IS CONTINUED, LOOPING
                         ;THROUGH BREAKPOINT 3 SIX TIMES,
                         ;HALTING ON THE 7TH OCCURRENCE OF THE
                         ;BREAKPOINT.
3B:001010                ;EXECUTION IS HALTED AT BREAKPOINT 3,
_                        ;ODT PRINTS BREAKPOINT MESSAGE AND
                         ;AWAITS A COMMAND.
```

The breakpoint proceed counts can be inspected by typing a command in the form:

```
_$nG/
```

where n represents the octal identifier for the breakpoint proceed count register (see $nG, Table 3-1). After the slash (/) is typed, ODT prints the contents of the specified register. The user may type the <CR> key to close the location (leaving the count unchanged), or a new count may be entered through the command $nG/k <CR>, where k represents the new count to be entered. Still another alternative is to type the <LF> key repetitively to examine the values in subsequent (or all) breakpoint proceed count registers. The following sequence shows how the proceed counts are examined and changed:

```
_$0G/000001 15 <LF>      ;PROCEED COUNT FOR BREAKPOINT 0 IS
                         ;EXAMINED, MODIFIED TO 15, FOLLOWED
                         ;BY LINE-FEED COMMAND.
$1G /000001 <LF>         ;PROCEED COUNT FOR BREAKPOINT 1 IS
                         ;EXAMINED, FOLLOWED BY LINE FEED
                         ;COMMAND.
$2G /000001 <LF>         ;PROCEED COUNT FOR BREAKPOINT 2 IS
_3G /000005 <CR>         ;EXAMINED, FOLLOWED BY RETURN
_                        ;COMMAND.
```

3.7 SINGLE-INSTRUCTION MODE COMMANDS: S or nS

A command has been provided in ODT to allow the user to step through the execution of the program one instruction at a time, if desired. An alternate form of this command takes an argument, allowing the user to specify the number of instructions to be executed before task execution is again suspended.

When the single-instruction mode is in effect, breakpoints are not present in the user task. Rather, task execution is suspended as a result of setting the T-bit in the Processor Status Word (see Appendix A) as the user instruction is executed. Thus, when executing the S command without an argument, each user instruction encountered is trapped to suspend execution. If the S command is being used with an argument, however, the trap occurs, but ODT does not suspend task execution until the specified instruction count has been completed, as described in the following paragraphs.

The command for the single-instruction mode takes the form nS, where n represents an octal integer specifying the number of user task instructions to be executed before control is returned to ODT. If n is omitted, an argument of 1 is assumed. When the instruction count (n) is completed, ODT suspends task execution and prints a message of the form 8B:a, where a represents the address of the next instruction to be executed. The following sequence illustrates the use of this command:

```
_7S                     ;SETS INSTRUCTION COUNT, ESTABLISHES
                        ;SINGLE-INSTRUCTION MODE, AND
                        ;INITIATES TASK EXECUTION.

8B:001000               ;INDICATES THAT INSTRUCTION COUNT HAS
                        ;BEEN COMPLETED, TYPES OUT ADDRESS OF
                        ;NEXT INSTRUCTION TO BE EXECUTED.
```

If ODT is currently representing task addresses relative to a relocation register, note (for reasons stated in section 3.1) that the terminal message for single instruction mode takes the form 8B:n,a. The value n represents the octal register specifier indicating the relocation register whose contents are closest in value to the address of the last instruction executed, and the value a represents the 6-digit octal address which must be added to the contents of relocation register n (i.e., the relocation bias of the module in question) to determine the actual relocated address of the location being displayed. The following command sequence illustrates this principle:

```
_101200;1R              ;SETS RELOCATION REGISTER 1
                        ;TO THE VALUE 101200.
_1,1052;B               ;SETS BREAKPOINT 0 RELATIVE
                        ;TO CURRENT VALUE OF
                        ;RELOCATION REGISTER 1.
_G                      ;SETS BREAKPOINT IN TASK AND
                        ;INITIATES TASK EXECUTION.
0B:1,001052             ;BREAKPOINT 0 OCCURS.
_S                      ;INITIATES SINGLE-INSTRUCTION MODE.
8B:1,001056             ;ADDRESSES OF NEXT INSTRUCTION
_S                      ;TO BE EXECUTED ARE REPRESENTED
8B:1,001062             ;AS VALUES WHICH MUST BE BIASED BY
                        ;CONTENTS OF RELOCATION REGISTER 1.
```

In the example above, to determine the relocated address of the next instruction to be executed, the user must add the values 1056 and 1062, respectively, to the relocation bias 101200 in relocation register 1, thus yielding relocated address values of 102256(8) and 102262(8).

3.8 SEARCH OPERATIONS

All or any specified portion of memory within the task's partition can be searched for word or byte locations which contain specific bit patterns. A second type of search can also be initiated which examines memory locations for words which reference a specified location in the user task. The following sections describe these search operations. (See also Appendix B, Search Algorithms.)

3.8.1 Word/Byte Search Commands: W, kW, m;W, or m;kW

Before initiating a word search, several preconditions must be established: (1) the search limits must be defined; (2) the search mask must be established; and (3) the search argument itself must be specified.

The search limits are defined through address values entered into the low memory limit regsiter ($L) and the high memory limit register ($H), as noted in Table 3-1. If, after opening the low memory limit register with the $L/ command, the current 6-digit octal value being displayed is not desired, the user may enter any new value appropriate to the intended search operation. The desired address value in the high memory limit register may be established in like fashion after first opening the register with the $H/ command.

As reflected in the command forms below, either absolute or relocatable task addresses can be set in the $L and $H registers:

```
$L/000000 1000          ;SETS $L TO ABSOLUTE ADDRESS.
$L/000000 1,1000        ;SETS $L TO RELOCATABLE ADDRESS.

$H/000000 2000          ;SETS $H TO ABSOLUTE ADDRESS.
$H/000000 0,2000        ;SETS $H TO RELOCATABLE ADDRESS.
```

When relocatable address values are specified in the search limit registers, the apparent values in $L and $H are effectively augmented by the value of the relocation bias for the object module.

The search mask is specified in the search mask register ($M). The command which accomplishes this action is described under $M in Table 3-1. Bits set to 1 in the mask define corresponding bit positions in the search argument (see below) and the memory words (or bytes) which will be compared during the search operation; bits not set to 1 in the mask cause the corresponding bit positions in the search argument and the memory words (or bytes) being searched to be ignored in all compare operations.

The search argument is specified in the search argument register ($A). The command which accomplishes this action is also described in Table 3-1 under $A. Note that either a word or byte search argument value can be specified.

The discrete actions described above establish the necessary preconditions for initiating search operations. These actions are reflected in the first four lines of the ODT command sequence in the example. At this point, the user need only type the W command in order to initiate the search, as shown in the fifth line of the example.

If a desired mask already exists, however, as a residual parameter from a previous search operation or as the result of an overt action in preparing for a new search operation, the user may initiate the search through a command of the following form:

```
kW
```

where k represents the desired search argument. In using this command form, note that the value preceding the W command is taken by ODT as the search argument, not the search mask.

On the other hand, if a desired search argument already exists as a result of the actions noted above, the user may initiate the search operation through a command of the following form:

```
m;W
```

where m represents the desired search mask.

A more convenient method of initiating search operations, however, is to specify the search mask and the search argument as part of a multi-element ODT command. Assuming that the search limits have already been specified in the $L and $H registers (as described above), the user may simply type a command in the following form:

```
m;kW
```

where m represents the search mask ($M), and k represents the search argument ($A). Typing W then initiates the search operation without further intervention. When a match occurs, i.e., when the corresponding bit positions in the search argument and the memory word being compared agree under the specified mask, ODT prints the address of the matching location and its contents.

The search operation is conducted in either word or byte mode, depending on the mode of the last open command.

In the search process, an exclusive OR (XOR) is performed with the word (or byte) currently being examined and the search argument. The result of this comparison is then ANDed with the specified search mask. If the result is zero, a match occurs. ODT then types the address and the contents of the matching location, as shown throughout the example below.

The following command sequences illustrate both word and byte search operations:

```
_$M/177777 177400 <CR>    ;SET MASK TO TEST HIGH-ORDER BYTE.
_$L/000000 1000 <CR>      ;SET LOW LIMIT SEARCH ADDRESS.
_$H/000000 1400 <CR>      ;SET HIGH LIMIT SEARCH ADDRESS.
_$A/000000 600 <CR>       ;SET WORD SEARCH ARGUMENT TO 600.
_W                        ;INITIATE WORD SEARCH OPERATION.
001010 /000770            ;PRINT ADDRESS AND MATCHING WORD.
001034 /000404            ;PRINT ADDRESS AND MATCHING WORD.
_377;W                    ;CHANGE MASK TO TEST LOW-ORDER
                          ;BYTE AND INITIATE SEARCH.
001020 /000200            ;PRINT ADDRESS AND MATCHING WORD.
_213W                     ;CHANGE SEARCH ARGUMENT TO 213 AND
                          ;INITIATE SEARCH.
001032 /000213            ;PRINT ADDRESS AND MATCHING WORD.
_$A\213 200 <CR>          ;SET BYTE SEARCH ARGUMENT TO 200.
_5W                       ;CHANGE BYTE SEARCH ARGUMENT TO 5 AND
                          ;INITIATE SEARCH.
```

If the user specifies a mask having zeros throughout, all memory locations within the search limits are printed by ODT.

The word/byte search algorithm is described in further detail in Appendix B.

3.8.2 Not This Word/Byte Search Commands: N, kN, m;N, or m;kN

This search works exactly the same as the word search described above, except that words (or bytes) which do not match are printed. Thus, a test for inequality is performed on all memory words/bytes in the specified search range.

3.8.3 Effective Address Search Commands: E, kE, m;E, or m;kE

ODT also searches for memory locations containing instructions which, when executed, effectively result in a reference to a specified task address. After first defining the search limits in the $L and $H registers, as described in section 3.8.1 above, the effective address search may be initiated by typing the following command:

```
m;kE
```

where m represents the search mask, and k represents the search argument. The values m and k are entered automatically in the search mask register ($M) and the search argument register ($A), respectively, when the m;kE command is issued.

As is the case with word/byte search operations described in sections 3.8.1 and 3.8.2, the command forms used to initiate an effective address search depend on which of the required ODT internal register values currently exist. For example, if the required register values are specified in discrete steps, as shown in the first four lines of the command sequences below, the effective address search can be initiated by typing only the E command. If the desired search mask value exists, however, as the result of prior action, the command form:

```
kE
```

suffices to initiate search operations, where k represents the search argument. If, on the other hand, the desired search argument exists as a result of prior action, the command form:

```
m;E
```

may be used to initiate the search, where m represents the search mask.

In an effective address search, the following types of words are printed by ODT:

1. Words which contain an absolute address (i.e., the search argument itself);
2. Words which contain a relative address offset reference to the specified search argument address; and
3. Words which contain a relative branch reference to the specified search argument address.

Note that since references to k, an effective address, are being searched for, normal usage of this command requires that the mask register be set to 177777; otherwise, the effective address will be modified.

The command sequences and ODT responses in an effective address search are illustrated in the example below:

```
_$M/000000 177777 <CR>    ;SET MASK TO COMPARE ALL BITS.
_$L/000000 400 <CR>       ;SET LOW LIMIT SEARCH ADDRESS.
_$H/000000 100400 <CR>    ;SET HIGH LIMIT SEARCH ADDRESS.
_$A/000000 1034 <CR>      ;SET EFFECTIVE ADDRESS SEARCH
                          ;ARGUMENT TO 1034.
_E                        ;INITIATE EFFECTIVE ADDRESS SEARCH
                          ;OPERATION.
001016 /001006            ;PRINT RELATIVE BRANCH LOCATION AND
                          ;CONTENTS.
001054 /002767            ;PRINT RELATIVE BRANCH LOCATION AND
                          ;CONTENTS.
_1020E                    ;INITIATE A NEW SEARCH FOR REFERENCES
                          ;TO LOCATION 1020.
001022 /177774            ;PRINT RELATIVE ADDRESS OFFSET
                          ;LOCATION AND CONTENTS.
001030 /001020            ;PRINT LOCATION CONTAINING ABSOLUTE
                          ;ADDRESS 1020.
```

Particular attention should be given to the reported references to the effective address, because a word may have the specified bit pattern of an effective address without actually being referenced in the program. ODT reports all occurrences of a possible effective address reference.

3.9 FILL COMMANDS: F or kF

The search argument register (see $A, Table 3-1) can be used in conjunction with the F command to set a block of memory to a specified value. While the most commonly-used value is zero, other possibilities are +1, -1, ASCII space, etc. Before a block of memory can be initialized to a given value, the limits of the memory area to be filled must be defined through the low memory limit register ($L) and the high memory limit register ($H). Consult table 3-1 for a description of the commands which store address values in these registers.

The initialization value may be stored in the search argument register (see $A, Table 3-1) as a discrete step. It is more convenient, however, to specify this value in the initialization command itself, which takes the following form:

```
kF
```

This command automatically stores the initialization value k in the search argument register ($A) and initiates the fill operation. ODT then stores this value into successive memory words or bytes, starting at the address specified in the low memory limit register ($L) and ending with the address specified in the high memory limit register ($H).

The initialization command fills the specified memory range with words if the last open command was performed in word mode; correspondingly, the specified memory range is filled with byte values if the last open command was performed in byte mode.

For the examples below, assume that the listed relocation registers contain the following values:

Relocation register 1 = 1000

Relocation register 2 = 2000

Relocation register 3 = 3000

The command sequences below might then occur. The first fill operation sets word locations 1000 through 1776 to zeros (0), while the second operation sets byte locations 2000 through 2777 to ASCII spaces.

```
_$L/000000 1,0 <CR>        ;SET LOW MEMORY LIMIT TO 1000.
_$H/000000 2,-2 <CR>       ;SET UPPER MEMORY LIMIT TO 1776.
_$A/123456 0 <CR>          ;SET SEARCH ARGUMENT REGISTER TO 0.
_F                         ;FILL SPECIFIED MEMORY BLOCK WITH
                           ;ZEROS.

_$L/001000 2,0 <CR>        ;CHANGE LOW MEMORY LIMIT TO 2000.
_$H/001776 3,-1 <CR>       ;CHANGE UPPER MEMORY LIMIT TO 2777.
_$A\000 40 <CR>            ;OPEN SEARCH ARGUMENT REGISTER IN
                           ;BYTE MODE AND CHANGE ITS CONTENTS TO
                           ;OCTAL 40 (ASCII SPACE).
_F                         ;FILL BYTES IN SPECIFIED MEMORY BLOCK
                           ;WITH SPACES.
```

In a fill operation, the memory limits must be defined through discrete actions which deposit the desired address values in the low memory limit register ($L) and the high memory limit register ($H). The only argument that can be specified as an element of the fill command is the fill value itself. When so specified, this fill value establishes the initial contents of the search argument register ($A) or modifies its current contents before initiating the fill operation. If the fill value is not specified when the F command is issued, ODT takes the current contents of $A in performing the fill operation.

3.10 OFFSET CALCULATION COMMANDS: aO or a;kO

Relative addressing and branching involve the use of an offset, i.e., the number of words or bytes forward or backward from the currently-open location to the effective address. During a debugging session, it may be desirable to change a relative address or branch reference by replacing an existing instruction offset with another value. ODT calculates and prints instruction offsets in response to the commands described below.

The aO command causes ODT to calculate and print the PC-relative offset and the branch displacement from the currently-open location to a specified address. Thus, the aO command is equivalent in function to the command .;kO, where the dot (.) represents the currently-open location (see section 3.16.1), and k represents the specified address.

The following sequences illustrate the use of the aO command:

```
_16126/001402 161340 _000004 >000002 <CR>
_1034/103421 10460 _000010 >000004 <CR>
_
```

In using the aO command form, it is assumed that a location is already open, as shown in the example above. Thus, the user need only specify the desired address to be used in calculating the offsets from the current location. After typing the O character, ODT calculates the offsets and prints the results on the same line. The PC-relative offset is flagged with the underline (_) or back-arrow character, and the branch displacement is flagged with the right angle-bracket (>) character.

The a;kO command causes ODT to calculate and print the PC-relative offset and the branch displacement from one specified address to another. In this command form, the symbol a represents the first address, and k represents the second address.

The following sequences illustrate the use of this command form:

```
_16126;16134O _000004 >000002 <CR>
_1034/103421 1022;1034O _000010 >000004 <CR>
_1022;1034O _000010 >000004 <CR>
_
```

In the first line of the examples above, the first address (16126) is specified, followed by a semicolon (;) and the second address (16134). After typing the O character, ODT calculates the offsets and prints them in the same manner as in the aO command above. The remaining examples follow this same pattern.

Note in the command form a;kO that it need not be issued in connection with an open location. Since both address values are explicitly specified, the address of the currently-open location has no implied effect in the calculation of the offset values. The second and third examples above illustrate this principle.

The command form a;kO is also useful in calculating negative offset values, as shown below.

```
_1022;1034O _000010 >000004 <CR>
_1034;1022O _177764 >177772 <CR>
_
```

The first example calculates a positive offset value, while the second sequence calculates a negative value. It is often desirable to know the PC-relative offset and the branch displacement values from a higher memory address to a lower memory address, since many instructions in the normal flow of program logic result in a transfer of control in the negative direction.

In either command form (aO or a;kO), note that the location for which offsets have been calculated remains open for further operations. For example, if the user wants to change the offset value in the low-order byte of the instruction word, he may do so as shown in the following sequence:

```
_1034/103421 1132O _000074 >036 1034\021 36 <CR>
_/103436
```

Note that location 1034 is first opened in word mode. If, after calculating the offsets, the user desires to change the value of the low-order byte, byte mode must first be established for that location; the command 1034\, as shown in the example above, is essential to this purpose. Unless byte mode is established for the current location,

any value then entered is interpreted as a word value. The net result in that case is the obliteration of the instruction op-code in the high-order byte.

If the user wishes to verify the alteration of the offset value, he may do so by typing the slash (/) command on the succeeding line, as shown above.

3.11 RELOCATION REGISTER COMMANDS: a;nR, a;R, nR, or R

The function of the relocation registers is described in section 1.2.3 and in Table 3-1. At the beginning of a debugging session, these registers are preset to the relocation biases of the relocatable modules of interest during the debugging session.

Relocation registers are initialized to -1 (octal 177777, the highest possible memory address), so that unwanted registers do not enter into the selection process when ODT searches for the most appropriate relocation register for its address calculations. When relocation registers are set to -1, all task image addresses reference either absolute physical memory locations (for non-mapped systems) or virtual memory locations (for mapped systems).

A relocation register is set by typing the desired bias value, followed by a semicolon and the specification of one of the eight relocation registers, as shown below:

```
a;nR
```

The symbol a represents an address expression, and n represents an integer from 0 through 7.

The following command form may also be used:

```
a;R
```

In this case, relocation register 0 is assumed to be specified, since the register specifier has been omitted. In contrast to the command form a;B for the breakpoint address registers described in section 3.5, however, the repetitive execution of the command form a;R does not enter values serially into the relocation registers. Therefore, the command form a;nR must be used to enter a bias value into a specific relocation register other than register 0.

To set all relocation registers to -1, the following command is typed:

```
_R                      ;SETS ALL RELOCATION REGISTERS
_                       ;TO -1, 177777(8).
```

To set a specified relocation register to -1, a command in the form nR is used, where n represents an octal register specifier, as shown below:

```
_3R                     ;SETS ONLY RELOCATION REGISTER 3 TO
_                       ;-1, 177777(8).
```

To set a specified relocation register to a desired value, a command in the form a;nR is used, where a represents the desired value, and n represents the octal register specifier, as shown below:

```
_1000;5R                ;SETS RELOCATION REGISTER 5 TO 1000.
_5,100;5R               ;EFFECTIVELY ADDS 100 TO THE CONTENTS
                        ;OF RELOCATION REGISTER 5.
_
```

Position-independent code may be loaded into address space other than that to which it was linked. When a program is loaded into address space below that at which it was linked, the appropriate relocation bias is the 2's complement of the apparent downward displacement. One method for easily evaluating this bias and storing it in the relocation register is illustrated below.

Assume, for example, that the program was linked to location 5000 and then moved downward to location 1000. The following command sequence would then be used:

```
_1000;1R                ;SETS RELOCATION REGISTER 1 TO 1000.
_1,-5000;1R             ;CHANGES RELOCATION BIAS TO ACTUAL
                        ;DOWNWARD DISPLACEMENT.
_
```

The last command above stores the 2's complement of 4000 in relocation register 1, as desired.

An alternate method of establishing the downward displacement might be the following command sequence:

```
_$0R/177777 1000-5000 <CR> ;OPENS RELOCATION REGISTER 0 AND
                           ;SETS RELOCATION BIAS TO ACTUAL
                           ;DOWNWARD DISPLACEMENT.
_/174000                   ;SLASH COMMAND PRINTS RELOCATION
                           ;BIAS IN PREVIOUSLY-OPENED LOCATION.
```

ODT maintains a table of relocation register locations, beginning with $0R. These locations may be opened through a command of the following form:

```
_$nR/
```

The symbol n represents an octal digit specifying which one of the eight locations is to be opened. Such locations may be opened and modified in the same manner as any other register location, as shown in the following sequence:

```
_$3R/001000 <LF>        ;RELOCATION REGISTER 3 IS OPENED.
                        ;THE <LF> COMMAND OPENS RELOCATION
$4R /002000 <LF>        ;REGISTER 4.  A SECOND <LF> COMMAND
                        ;OPENS RELOCATION REGISTER 5.  THE
$5R /004000 <CR>        ;<CR> COMMAND ENDS SEQUENCE.
_$3R/001000 1040 <CR>   ;OPENS RELOCATION REGISTER 3 AND
                        ;CHANGES ITS CONTENTS TO 1040.
_6000;6R                ;SETS RELOCATION REGISTER 6 TO 6000
                        ;AND ENDS SEQUENCE.
_
```

3.12 RELOCATION CALCULATOR COMMANDS: a;nK, nK, or K

When a location has been opened, it is often desirable to associate the relocated address of that location with its relocatable value.

To calculate the relocatable address of a given location relative to a particular relocation bias, the user types a command in the following form:

```
a;nK
```

The symbol a represents the relocated address value from which the relocatable address is to be calculated, and n represents the relocation register specifier (0 through 7).

The K command is effective in conjunction with opened word and byte locations. For example, if the command elements a and ; are not specified, the currently-open location is assumed to be operative (i.e., .; is assumed). Thus, the command 3K is equivalent in function to the command .;3K (see section 3.16.1).

If the relocation register specifier n is omitted, the relocation register whose contents are equal to, or closest to (but less than) the currently-open location is automatically selected by ODT for use in calculating the relocatable address. In the example below, relocation register 2, which contains 2000, meets this requirement:

```
_2500;K  2,000500          ;CALCULATES RELOCATABLE ADDRESS.
```

Thus, ODT's response to the K command consists of the octal identifier (2) of the relocation register used in the calculation, followed by the relocatable address value (000500) of the specified relocated address (2500).

3.13 LISTING COMMANDS: L, kL, a;L, a;kL, or n;a;kL

A number of command forms are available for listing a block of memory locations within the user task's partition. The particular command form used in initiating a listing operation depends on whether the required ODT register values exist as the result of prior action or whether they must be specified overtly as an argument in the listing command itself. In either case, the following ODT registers must contain the user-specified values required for the intended listing operation:

1. The beginning address of the memory range to be printed must be deposited in the low memory limit register (see $L, Table 3-1).

2. The ending address of the memory range to be printed must be deposited in the high memory limit register (see $H, Table 3-1).

3. The logical unit number of the listing device must be deposited in the device control LUN register (see $nD, Table 3-1). The appropriate values for the device control LUN registers are normally established by the Task Builder. Therefore, the user need not be concerned with any explicit ODT operations in establishing or altering these values. The

default values for these registers are described in detail in Table 3-1. Normally, device control LUN register 0 ($0D) contains the logical unit number of the user terminal device (TI:), while device control LUN register 1 ($1D) contains the logical unit number of the console listing device (CL:).

Assuming that the necessary register values described above exist as the result of prior action, the user need only type the L key to initiate a listing operation:

```
_L                          ;PRINTS MEMORY LOCATIONS WITHIN
                            ;SPECIFIED ADDRESS LIMITS USING
                            ;CONSOLE LISTING DEVICE (CL:).
```

If the desired address value presently exists in the low memory limit register ($L) and the user wishes either to establish the required value for the high memory limit register ($H) or to modify its current contents, the following command form is used:

```
_kL                         ;TAKES ADDRESS VALUE "k" AS ENDING
                            ;LOCATION AND INITIATES LISTING
                            ;OPERATION.
```

Conversely, if the desired address value presently exists in the high memory limit register ($H) and the user wishes either to establish the required value for the low memory limit register ($L) or to modify its current contents, the following command form is used:

```
_a;L                        ;TAKES ADDRESS VALUE "a" AS
                            ;BEGINNING LOCATION AND INITIATES
                            ;LISTING OPERATION.
```

If neither of the required address values presently exist in $L and $H, the following command form is used:

```
_a;kL                       ;TAKES ADDRESS VALUES "a" AND "k"
                            ;AS THE BEGINNING AND ENDING
                            ;ADDRESSES, RESPECTIVELY, AND
                            ;INITIATES THE LISTING OPERATION.
```

Finally, a fourth command form is used if none of the required values presently exist. In this case, all the discrete values required for a listing operation must be specified as arguments in the listing command itself, as shown below:

```
_n;a;kL                     ;ALL LISTING CONTROL ARGUMENTS
                            ;ARE SPECIFIED IN SINGLE
                            ;LISTING COMMAND.
```

The command form above uses the logical unit number contained in the specified device control LUN register n ($nD) to print all memory locations within the specified address limits a and k. Any value for n other than zero (0) or one (1) is treated by ODT as though the console listing device ($1D) was specified, i.e., the default value for n is one (1).

The address values a and k specified as arguments in the listing command may be either absolute or relocatable in form. It is advisable, however, when debugging relocatable program segments, to use relocatable address expressions in defining the memory limits in $L and $H for listing operations. By so doing, the task addresses

printed out by ODT during the listing of the specified memory block can be associated directly with the relocatable addresses in the assembly listing. Any address values so specified, whether absolute or relocatable, cause corresponding values to be entered into the memory limit registers when the L command is executed. The use of relocatable address values, however, assumes that an appropriate relocation bias for the object module in question has been established in one of the eight available relocation registers. For example, if an object module has a relocation bias of 370(8), the following command sequence might be used in initiating a listing operation:

```
_0,370;R <CR>          ;SETS RELOCATION REGISTER 0 TO
                       ;RELOCATION BIAS FOR DEBUGGING THE
                       ;MODULE.
_0,1020;0,1040L        ;INITIATES THE LISTING OF THE MEMORY
                       ;BLOCK BETWEEN RELOCATABLE TASK
                       ;ADDRESSES 1020 AND 1040.
```

The beginning address in the resulting listing will then be represented in relocatable form, i.e., 0,001020, as will the ending address.

In listing a block of memory, it is important to note that the listing format is governed by the mode of the previous open command. In other words, the interpretation of each memory location and the format of the listing output are determined by the last output mode used by ODT. In this connection, two general output listing modes are available, as described below:

1. Word mode octal. Established through opening a word location through the slash (/) command (see section 3.2.3) or any other command which opens a location and causes its entire 6-digit octal value to be printed.

2. Byte mode octal. Established through opening a byte location through the backslash (\) command (see section 3.2.3) or any other command which opens a byte location and causes its 3-digit octal value to be printed.

A sequence of operations which results in the printout of both word and byte locations is shown in Figure 3-1.

Three other listing options are available to the user through establishing alternate modes, as described below:

1. Word mode ASCII. Established through interpreting the contents of a location using the double-quote (") character before issuing the listing command. For example, the representative expression 0,1020", as shown in Figure 3-1, establishes word mode ASCII before the listing operation is initiated.

2. Byte mode ASCII. Established through interpreting the contents of a location using the single-quote (') character before issuing the listing command. As shown in Figure 3-1, the representative expression 0,1020' establishes byte mode ASCII before the listing command is issued.

3. Word mode Radix-50. Established through interpreting the contents of a location using the percent sign (%) before issuing the listing command. This listing option is invoked through the representative expression 0,1020%, as shown in Figure 3-1.

Note that the address expressions referenced in Items 1 through 3 above are intended to be illustrative only. Any means of establishing the desired output mode before issuing the listing command is permissible.

In all cases of ODT listing output, the first line starts with the beginning octal address of the memory block being printed, followed by the contents of eight consecutive word or byte locations. Subsequent lines consist of the beginning octal address of the next eight consecutive locations, etc.

```
>PRE
ODT:...PRE
.
_110400;R
_$L/000000 0,04522
_$H/000000 0,04622
_L
0,004522 /020105 000000 000000 020000 047101 020104 054524 042520
0,004542 /041440 051101 042522 037124 046451 052517 052116 044440
0,004562 /050116 052125 053040 046117 046525 020105 000000 000000
0,004602 /020000 047101 020104 054524 042520 041440 051101 042522
0,004622 /037124
_0,04522\105 L
0,004522 \105 040 000 000 000 000 000 040
0,004532 \101 116 104 040 124 131 120 105
0,004542 \040 103 101 122 122 105 124 076
0,004552 \051 115 117 125 116 124 040 111
0,004562 \116 120 125 124 040 126 117 114
0,004572 \125 115 105 040 000 000 000 000
0,004602 \000 040 101 116 104 040 124 131
0,004612 \120 105 040 103 101 122 122 105
0,004622 \124
_0,04522'E L
0,004522 'E
0,004532 'A N D   T Y P E
0,004542 '  C A R R E T >
0,004552 '> M O U N T   I
0,004562 'N P U T   V O L
0,004572 'U M E
0,004602 '   A N D   T Y
0,004612 'P E   C A R R E
0,004622 'T
_0,04522"E  L
0,004522 "E      AN D  TY PE
0,004542 " C AR RE T> >M OU NT  I
0,004562 "NP UT  V OL UM E
0,004602 "  AN D  TY PE  C AR RE
0,004622 "T>
_0,04522%EFU L
0,004522 %EFU          ED2 LT3 EFT NK. KCX
0,004542 %J/X MFQ KCZ I86 LM3 MY9 MSV K.
0,004562 %L38 MS/ M1H LHO LN7 EFU
0,004602 %ED2 LT3 EFT NK. KCX J/X MFQ KCZ
0,004622 %I86
-
```

Figure 3-1 ODT Listing Modes and Formats

3.14 REPRINTING OPEN LOCATIONS

It is often desirable to print the contents of an open location in a mode other than that in which it was opened or to print its contents in other than 6-digit octal format. The commands described below, all of which cause the word or byte value to be stored in the quantity register ($Q) when printed, are available for this purpose.

An important operational characteristic of ODT should be noted in connection with the use of the interpretive commands described below and the <LF> command. As pointed out in section 3.13, ODT has five distinct output modes:

1. Word mode octal (/)
2. Byte mode octal (\)
3. Word mode ASCII (")
4. Byte mode ASCII (')
5. Word mode Radix-50 (%)

When a location is opened in any one of these output modes, ODT "remembers" (saves) the mode of the location just opened/interpreted. Although the user may then issue any other interpretive command(s) on the same line, when the <LF> command is entered to close that location, ODT opens the next sequential location in the mode of the previous location. In other words, after a location is opened or interpreted, the output mode thus established, prevails for all subsequent <LF> commands. The following command sequences illustrate this principle:

```
_0,234\346 'F <LF>          ;RELOCATABLE LOCATION 234 OPENED IN
                            ;BYTE MODE OCTAL AND INTERPRETED IN
                            ;BYTE MODE ASCII, FOLLOWED BY <LF>
                            ;COMMAND.
000235 \025 <LF>            ;NEXT SEQUENTIAL LOCATION OPENED
                            ;IN BYTE MODE.
000236 \100 <CR>            ;SAME AS ABOVE.


_1000"AB 'A <LF>            ;CONTENTS OF LOCATION 1000 INTERPRETED
                            ;IN WORD MODE ASCII AND BYTE MODE
                            ;ASCII, FOLLOWED BY <LF> COMMAND.
001002 "CD <LF>             ;NEXT SEQUENTIAL LOCATION INTERPRETED
                            ;IN WORD MODE ASCII.
001004 "EF <CR>             ;SAME AS ABOVE.


_0,232/034567 'W "W9 %IG1 <LF> ;RELOCATABLE LOCATION 236
                               ;OPENED IN WORD MODE OCTAL AND
                               ;INTERPRETED IN BYTE MODE ASCII, WORD
                               ;MODE ASCII, AND WORD MODE RADIX-50,
                               ;FOLLOWED BY <LF> COMMAND.
0,000234 /000624 <LF>          ;NEXT SEQUENTIAL LOCATION OPENED IN
                               ;WORD MODE OCTAL.
0,000236 /000100 <CR>          ;SAME AS ABOVE.
```

Although the examples above are general in nature and do not illustrate the use of all the output modes, the principle so demonstrated applies to all the interpretive commands described in sections 3.14.1 through 3.14.4. For convenience, these examples are presented in this section, rather than being repeated in context with the discussions below.

3.14.1 Print Octal Byte Value: \

Typing the backslash (\) command when a word location is currently open causes ODT to interpret and print the low-order byte of the word as three octal digits, as shown below:

```
_0,20/044520 \120          ;PRINTS LOW-ORDER BYTE IN
                           ;CURRENTLY-OPEN WORD LOCATION.
```

Typing the backslash command when a byte location is currently open causes ODT to reprint the contents of the byte location; as shown below:

```
_0,23\021 \021             ;REPRINTS VALUE OF BYTE LOCATION.
```

Typing the backslash command when a location is not open causes ODT to print the byte value of the last-opened location, as shown below:

```
_0,234/000247 123456 <CR>
_\056                      ;PRINTS LOW-ORDER BYTE OF
                           ;PREVIOUSLY-OPENED WORD LOCATION.

_0,237\041 <CR>
_\041                      ;REPRINTS VALUE OF PREVIOUSLY-
                           ;OPENED BYTE LOCATION.
```

3.14.2 Print Byte Mode ASCII Character: ' or a'

Typing the single-quote character (') when a word location is currently open causes ODT to interpret and print the low-order byte of the location as one ASCII character, as shown below:

```
_0,232/034567 'W           ;PRINTS THE CONTENTS OF THE OPEN WORD
                           ;LOCATION AS ONE ASCII CHARACTER.
```

Typing the single-quote character when a byte location is open causes ODT to interpret and print the byte value as one ASCII character, as shown in the examples below:

```
_0,232\167 'W
_0,1020\323 'S
_0,233\071 '9
```

When the single-quote character is preceded by an address expression, ODT uses the expression as an argument in determining the location to be interpreted and printed, as shown below:

```
_0,232'W                   ;INTERPRETS REFERENCED LOCATION AS
                           ;ONE ASCII CHARACTER.
```

If no location is currently open when the single-quote command is issued, the previously-opened location is interpreted and printed, as shown in the following sequence:

```
_0,232/034567 <CR>
_'W                          ;INTERPRETS PREVIOUSLY-OPENED WORD
                             ;LOCATION AS ONE ASCII CHARACTER.
```

3.14.3 Print Word Mode ASCII Characters: " or a"

Typing the double-quote character (") when a word location is currently open causes ODT to interpret and print the contents of the location as two ASCII characters, as indicated below:

```
_0,232/034567 "W9            ;PRINTS THE CONTENTS OF THE OPEN WORD
                             ;LOCATION AS TWO ASCII CHARACTERS.
```

If the double-quote character is preceded by an address expression, ODT takes the expression as an argument in determining the location to be interpreted and printed, as shown below:

```
_0,232"W9                    ;INTERPRETS REFERENCED LOCATION AS
                             ;TWO ASCII CHARACTERS.
```

If no location is currently-open when the double-quote command is issued, the previously-opened location is interpreted and printed, as reflected below:

```
_0,232/034567 <CR>
_"W9                         ;INTERPRETS PREVIOUSLY-OPENED WORD
                             ;LOCATION AS TWO ASCII CHARACTERS.
```

Note that the double-quote command is effective only when issued in connection with task locations which fall on even (word-boundary) addresses, as shown below:

```
_0,232/034567 W9 <CR>        ;INTERPRETS ENTIRE WORD VALUE.
_0,232\167 "W9 <CR>          ;ALSO INTERPRETS ENTIRE WORD.
_
```

In contrast, however, issuing the double-quote command in connection with an odd address (byte) location, although legal, merely causes the 3-digit octal byte value to be reprinted, as reflected in the following sequence:

```
_0,233\071 "071 <LF>         ;REPRINTS ODD BYTE VALUE.
0,000234 \346 "F <LF>        ;INTERPRETS WORD LOCATION NORMALLY.
0,000235 \025 "025           ;REPRINTS ODD BYTE VALUE.
```

Note, in the second line above, that the word location relocatable 234 is interpreted normally in its entirety, even though the location is currently open in byte mode. This example underscores the usefulness of the double-quote character only in connection with word-boundary locations.

3.14.4 Print Word Mode Radix-50 Characters: % or a%

Typing the percent sign (%) when a word location is open causes ODT to interpret and print the contents of the location as three Radix-50 characters, as shown below:

```
_0,232/034567 %IG1          ;PRINTS THE CONTENTS OF THE OPEN WORD
                            ;LOCATION AS THREE RADIX-50
                            ;CHARACTERS.
```

If the percent sign is preceded by an address expression (a), ODT evaluates the expression to determine the location to be interpreted and printed, as shown below:

```
_0,232%IG1                  ;INTERPRETS REFERENCED LOCATION AS
                            ;THREE RADIX-50 CHARACTERS.
```

As with the preceding single-quote and double-quote interpretive commands, typing the percent sign when no location is currently open causes ODT to print the contents of the previously-opened location in Radix-50 form, as shown below:

```
_0,232/034567 <CR>
_%IG1                       ;INTERPRETS PREVIOUSLY-OPENED
                            ;LOCATION AS THREE RADIX-50
                            ;CHARACTERS.
```

Also, the percent sign is effective only when issued in connection with an even address location, even though a word-boundary location may be currently open in byte mode, as shown below:

```
_0,242/001542 % UZ <CR>     ;INTERPRETS ENTIRE WORD VALUE.
_0,242\142 % UZ <CR>        ;ALSO INTERPRETS ENTIRE WORD.
_
```

Using the percent sign in connection with an odd address value has the same effect as that described above for the double-quote character, as indicated below:

```
_0,241\025 %025 <LF>        ;REPRINTS ODD BYTE VALUE.
0,000242 \142 % UZ <LF>     ;INTERPRETS WORD LOCATION NORMALLY.
0,000243 \003 %003          ;REPRINTS ODD BYTE VALUE.
```

3.15 INTERPRETING EXPRESSION VALUES: k=

The equal sign (=) enables the user to interpret expression values, address values, and a variety of other expression forms involving arithmetic operations. The use of this command implies that it is preceded by the entry of one or more characters constituting a legal ODT expression.

This command cannot be used to interpret the currently-open location or the last previously-opened location. A character sequence must be entered overtly prior to issuing the = command: otherwise, ODT prints out a string of six octal zeros (000000).

As shown in the examples below, this command causes any expression value which precedes it to be converted to a 6-digit octal value. In addition, the word so printed is stored in the quantity register (see

$Q, Table 3-1). A wide range of operations is possible using this command, as reflected in the following sequences:

```
_0,370;R
_0,0=000370
_2,16*$0D+6=003364
_0,16=000406
_370+16=000406
_0,16+16+2=000426
_370+16+16+2=000426
_16+370=000406
_16-370=177426
_-370+16=177426
_-370-16=177372
_-177777+16+16=000035
_+1+16+16=000035
_177777+16+16=000033
_-1+16+16=000033
_232323=032323
```

Note in the sequence above that any expression value that references relocation register 0 causes ODT to take the current contents of that register as an argument in evaluating the expression. Those expressions beginning with 0 cause the value 370(8) to be used as the effective value of that element, as established through the initial command of the sequence above. Any of the relocation registers can be used in this manner.

Both positive and negative values may be specified as arguments in an expression, as reflected throughout the examples above. ODT performs the necessary arithmetic calculations and prints out the result as a 6-digit octal value.

Note also that any expression value preceding the equal sign is truncated to 16 bits before being evaluated and printed in 6-digit octal form, as shown in the last expression in the above sequence.

3.16 USING SPECIAL ARGUMENTS IN ODT COMMANDS

The special arguments described below may be used in place of the elements a and k in ODT commands.

3.16.1 Current Location Indicator: .

When used in a command sequence, the dot (.) represents the address of the currently-open location, as shown below:

```
_1000/000000 .=001000        ;DOT (.) REFERENCES ADDRESS OF
                             ;LAST OPENED LOCATION.
```

3.16.2 Constant Register Indicator: C

The user may store any 16-bit expression value in the constant register (see $C, Table 3-1). To open the register and print its contents, the user issues the $C/ command. Any new value desired can

then be entered directly, followed by a <CR> command. The value so contained in the constant register may then be used in any subsequent ODT operations by typing the letter C as an argument in a command.

The example below shows how the constant register is accessed and modified:

```
_$C/000000 123 <CR>     ;CONSTANT REGISTER OPENED, MODIFIED
                        ;TO CONTAIN 123, FOLLOWED BY RETURN
                        ;COMMAND.
_/000123                ;CONSTANT REGISTER OPENED, CURRENT
                        ;CONTENTS ARE PRINTED.
```

3.16.3 Quantity Register Indicator: Q

Each time ODT prints a value on the console, the value is stored in the quantity register (see $Q, Table 3-1). The value so stored may then be used in any subsequent ODT operations by typing Q as an element of a command. This facility is useful in modifying open locations. For example, if location 1342 contains a value which is too small for current debugging purposes, the value may be modified, as shown in the first line of the command sequence below:

```
_1342/173214 Q+10 <CR>  ;ADD THE VALUE 10 TO THE CURRENT
                        ;CONTENTS OF LOCATION 1342.
_/173224                ;SLASH OPENS PREVIOUS LOCATION,
                        ;PRINTS ITS CONTENTS, AND STORES NEW
                        ;VALUE IN Q REGISTER.
```

Therefore, when Q is employed as an argument in an expression in this manner, the contents of the currently-open location are modified to contain the octal equivalent of the expression. Note in the last line of the sequence above that issuing the slash command verifies the modification of the previously-open location.

As a second example, assume that the contents of user general register 3 point to a routine that has been relocated through relocation register 5. The following command sequence might then occur:

```
_$3/013624 Q;5R         ;SETS RELOCATION REGISTER 5 TO 013624
                        ;(THE CURRENT VALUE OF Q REGISTER).
_5,20=013644            :EVALUATES THE RELOCATABLE ADDRESS
                        ;EXPRESSION, PRINTS THE VALUE OF THE
                        ;EXPRESSION, AND STORES THIS WORD IN
                        ;THE OUANTITY REGISTER.
```

After the relocatable address expression in the example above is evaluated, the quantity register contains the value 013644(8), while relocation register 5 retains the value 013624(8) set in the preceding command.

3.16.4 Radix-50 Operator: *

The asterisk (*) is an arithmetic operator that is used primarily in the calculation of Radix-50 arguments. Calculating such arguments is necessary if the user wants to enter any of the Radix-50 characters into a word location. The asterisk (*) allows the user to derive the 6-digit octal equivalent of the desired Radix-50 character(s) so that the proper value can be entered into the appropriate location. Table 3-2 lists all the legal Radix-50 characters, together with an equivalent numeric value that is used in conjunction with the asterisk to calculate a 6-digit octal representation of any combination of up to three Radix-50 characters.

By consulting Table 3-2, the user can calculate any desired 1- to 3-character Radix-50 sequence for entry into a word location. For example, if it is desirable to enter the Radix-50 characters ABC into a given location, the 6-digit octal representation of these characters is calculated as follows:

```
_1*2*3=003223
```

Note from this sequence that the numeric values 1, 2, and 3 are taken from Table 3-2 as the arguments to be used in the calculation. For this purpose, these numeric values represent the Radix-50 characters A, B, and C. Hence, the 6-digit result of this calculation is the octal value that must be entered into the desired location to correctly represent the intended Radix-50 sequence.

Table 3-2 may be used in this manner to calculate any corresponding 6-digit octal value of any desired combination of three Radix-50 characters, including the special characters space, $, and dot (.).

The following sequences illustrate how the Radix-50 operator (*) may be used:

```
_1052/174777 %999 1*2*3=003223 3223 <CR>
_%ABC

_1054/003151 %AAA 1*3*5=003275 3275 <CR>
_%ACE
```

In the examples above, note that the contents of the open location are first interpreted by typing the percent sign (%). This command, although entirely optional, permits the user to ascertain the current contents of the location in Radix-50 form (see section 3.14.4). At this point, if the user elects to enter some other Radix-50 character sequence, he may do so by calculating its value as shown above and then entering this value before closing the location. Also optional is the second line of each sequence which interprets the new Radix-50 character(s) in the changed location to verify their accuracy.

As a further example, the user may wish to enter the Radix-50 characters $TJ into location 1054. After opening this location, Table 3-2 is consulted for the corresponding numeric values to be used in the calculation. These values are determined to be 33, 24, and 12. The sequence of ODT operations then proceeds as follows:

```
_1054/124157 33*24*12=125752 125752 <CR>
_%$TJ
```

Note that spaces (blanks) are valid characters in deriving Radix-50 character sequences. A space is represented in the calculation as 0, as shown below:

```
_1*0=000050
_0*1=000001
_2*0*3=006203
```

Table 3-2
Legal Radix-50 Characters and Numeric Equivalents

Radix-50 Character	Numeric Equivalent	Radix-50 Character	Numeric Equivalent
Space	0	T	24
A	1	U	25
B	2	V	26
C	3	W	27
D	4	X	30
E	5	Y	31
F	6	Z	32
G	7	$	33
H	10	.	34
I	11	Unused	35
J	12	0	36
K	13	1	37
L	14	2	40
M	15	3	41
N	16	4	42
O	17	5	43
P	20	6	44
Q	21	7	45
R	22	8	46
S	23	9	47

3.17 REENTRY VECTOR REGISTER: $X

When a task is requested frequently for execution, as is often the case in debugging operations, it may be desirable to fix an installed task in memory. Once fixed in memory, the task does not relinquish its memory space, and a fresh copy of the task is not loaded from disk each time the task is requested for execution.

If the user is performing multiple executions of the task during a debugging session, the reentry vector register is useful in connection with such a fixed task. This register may be set to a value which causes the task to be reentered directly at its own starting address, rather than at ODT's starting address.

Initially, when ODT is linked to the user task, the reentry vector register has a value of minus one (-1). The first time the task is executed, this register is incremented to zero (0). The next time the task is requested for execution, ODT inhibits the printout of the task's name following the "ODT:" console output. The absence of the task name in this console response indicates that the task is still fixed in memory. At this point, the user may set the reentry vector

register to a value of one (1), or any positive nonzero value, causing the task to be reentered at the address contained in $7 (the program counter) when the G command is again issued to begin task execution.

The significance of this option is that any task locations modified (patched) during the initial debugging pass are retained. Thus, the task, as initially executed, remains in core, and the user need not start all over again each time the task is executed.

When this option is exercised, task execution is initiated without reentering ODT. In employing this option, however, the user must be aware of any execution-dependent variables in the task. Switch values or counters, for example, may be altered during the course of initial task execution and may not be valid for a subsequent execution. Such possibilities must be kept in mind when performing multiple executions of a fixed task.

CHAPTER 4

OPERATING PROCEDURES

ODT is supplied to the user as a relocatable object module which must be linked by the Task Builder to the user program. The output of the Task Builder is thus an executable task image comprising both the user object modules and ODT. After loading, this task image file is the focus of all ODT debugging operations.

The term Task Builder, as used in this manual refers to the particular Task Builder of the system being used, i.e., that of RSX-11M, RSX-11D, or IAS. The user is referred to the appropriate Task Builder Reference Manual for a more detailed discussion of its functions.

4.1 FILE TYPE OR EXTENSION VALUES

Any number of input files can be specified for the Task Builder. If the input file specification does not contain an explicit declaration of the file type (RSX) or extension (IAS), the Task Builder assumes the default value .OBJ for the input file.

The Task Builder produces three types of output files:

TSK - Executable Task Image File

MAP - Memory Allocation Map File

STB - Task Image Symbol Table File

4.2 OUTPUT FILE SWITCH OR QUALIFIER OPTIONS

The output file switch (RSX) or qualifier (IAS) options that are applicable to the ODT user, relate to the executable task image file (TSK). The options of specific interest are:

/DA (RSX) or /DEBUG (IAS)	This option indicates that a debugging aid (ODT) is to be linked with the user task, causing the object module ODT.OBJ, which resides in the system area (SY:[1,1]), to be linked automatically into the task image.

When the /DA switch under RSX, or the /DEBUG qualifier of the LINK command under IAS is specified, registers 0 through 4 become initialized with the following values. Register 0 contains the task's entry-point address. User registers 1 and 2 contain the run-time name

of the task in Radix-50 form; register 1 contains the first three characters of the task name, and register 2 contains the last three characters. Registers 3 and 4 contain the version number of the user's task if an .IDENT assembler directive is present in the task, or the version number of ODT if no .IDENT directive is present. When ODT gains control at task initiation, user general registers 0 through 7 are saved in ODT's registers $0 through $7, and ODT's stack is set up. While ODT is in control, the ODT stack is used, and ODT registers $0 through $7 contain the current values of the corresponding user program registers. When ODT returns control to the user program, the task's starting address is restored to the program counter, and the task's stack and other general registers are likewise restored.

For tasks that were built without a debugging aid, control is transferred directly to the user program's entry-point (transfer) address upon task initiation.

4.3 LINKING AND INITIATING ODT

4.3.1 RSX-11 Systems

As an example of linking and initiating ODT with the user program, assume that an object module named TEST.OBJ is to be linked with ODT. The following sequence of keyboard commands and responses would then occur:

```
>TKB

TKB>TEST/DA=TEST

TKB>/

TKB>ENTER OPTIONS:

TKB>TASK=TEST

TKB>/

>INSTALL TEST

>RUN TEST

ODT:TEST

_
```

The first line of the command sequence above requests the execution of the Task Builder. The second line contains the Task Builder prompting sequence TKB>, indicating its readiness to accept input; the entry of the input/output file specifications then follows on the same line. This command string indicates that ODT is to be linked with the user program TEST.OBJ in forming the executable task image output file TEST.TSK.

In the third line, the user types a slash (/) in response to the Task Builder prompting sequence to indicate that additional input is to be entered. The Task Builder then responds with the fourth line,

continuing with the TKB> prompting sequence in the fifth line to indicate its readiness to accept additional input; the user then establishes the task name with the entry TASK=TEST to complete the line. The slash (/) entered in line six in response to the Task Builder prompting sequence terminates this part of the input.

The seventh line of the sequence installs the task in the system, and makes it available for execution. This process consists of building an entry for the task in the system task directory, setting up various pointers, and assigning devices to logical unit numbers. Line eight requests the execution of the task, causing the system to respond with the message ODT:TEST, indicating that ODT has been given control. Finally, the prompting character (_) on the last line indicates ODT's readiness to accept user input to initiate the debugging session.

4.3.2 IAS System

The following example shows the linking of ODT with a user object module (TEST.OBJ) under IAS, and the initiation of the task image (TEST.TSK).

```
PDS> LINK/DEBUG TEST      ;LINK ODT WITH TEST.OBJ
PDS> RUN TEST             ;EXECUTE TEST.TSK
ODT:TEST

_
```

4.4 OTHER DEBUGGING AIDS

Other debugging aids can be specified by the user in place of ODT as follows:

RSX:

```
TKB>TEST=ERRTST/DA,TEST  ;/DA ON INPUT, NAMES USER DEBUG AID
```

IAS:

```
PDS> LINK/DEBUG:ERRTST TEST  ;ERRTST IS USER DEBUGGING AID
```

These forms of /DA and /DEBUG have the following effects:

1. The transfer address in the alternate debugging aid overrides the task transfer address.

2. On initial task load, the following registers have the indicated values:

   R0 - Transfer address of task
   R1 - Task name specified at task-build time in Radix-50 format (word #1)
   R2 - Task name (word #2)
   R3 - Version number of user's task (if .IDENT assembler
   & directive is present in the task)
   R4 or
   Version number of the first input file encountered that contains an .IDENT directive

   If the run-time task name is desired, rather than the task-build task name, the user can issue the GET TASK PARAMETERS Executive directive, which returns the run-time name.

4.5 RETURNING CONTROL TO THE HOST SYSTEM

If ODT is awaiting keyboard input, typing an X causes execution of the system Task Exit directive, thereby terminating task execution and returning control to the host operating system.

CHAPTER 5

ERROR DETECTION

5.1 COMMAND INPUT ERRORS

ODT checks the legality of an address when commanded to open a location for examination or modification. If an error is detected, ODT responds by printing the question mark (?) character, followed by the underline (_) prompting character on the next line. For example, if the command

```
177774/ ?
_
```

references nonexistent memory or an address outside the task's partition, the request is ignored, and the appropriate typeouts occur.

In addition, a command such as

```
$20/ ?
_
```

which specifies an invalid (nonexistent) register, causes ODT to flag the line with a question mark (?), ignore the request, and print the prompting character.

In general, typing an illegal character or command causes ODT to ignore the input, print the question mark error indicator

```
  ?
_
```

and wait for a valid command.

To cause ODT to ignore a command just entered, any illegal character (such as the decimal value 8 or 9) may be typed, causing the command to be treated as an error (ignored).

ODT suspends task execution whenever a breakpoint location is encountered (i.e., the user program traps to ODT's breakpoint processing routine). If the breakpoint routine is entered and no known breakpoint has caused the trap action, ODT prints a message in the form

```
BE:001542
_
```

and waits for another command. In the example above, the message BE:001542 denotes a bad entry from location 001542 (see BE, Table 5-1). This type of error message may be caused by an illegal BPT instruction in the user task, setting the T-bit in the Processor Status Word ($S), or a branch to a location within ODT.

Although octal op code 000003 (the BPT instruction) is a valid instruction in a user program, its use is discouraged, since this instruction will result in an unwanted breakpoint when the program is run under ODT control.

5.2 TASK IMAGE ERROR CODES

In addition to command input errors, ODT alerts the user to certain hardware-detected errors that occur during task execution. Such errors, which are attributable to problems in the task itself, result in a trap to one of the error-handling routines pointed to by the SST vector registers in ODT (see $nV, Table 3-1). These registers, each containing an entry-point address to an associated error-handling routine, constitute the mechanism through which ODT handles synchronous system traps. Depending on the particular type of error condition detected, ODT activates the appropriate error-handling routine, which then evaluates the condition and prints out an error code in the form

```
cc:k
```

where cc represents one of the 2-character alphabetic error codes listed in Table 5-1, and k represents the 6-digit octal address following the location in error, unless the code is BE. If the code is BE, k represents the address of the location in error. The alphabetic error code and the address value are always separated by a colon (:).

At this point, the user can examine the error location, register values, and other key locations in the task image. If the cause of the error can be determined, appropriate modifications can then be made in the user task and noted on the assembly listing; if the error cannot be isolated, it is advisable to load a fresh copy of the entire task and initiate a new debugging session.

Table 5-1
ODT Error Codes

Code	Meaning
MP*	Memory protect violation or illegal memory reference.
OD	Odd address reference on word instruction. (Also, on some PDP-11 processors (e.g., PDP-11/45), the execution of an illegal instruction is reported by this code rather than by an IL.)
IO	IOT instruction executed.
IL	Reserved or illegal instruction executed.
EM	Non-IAS/RSX-11 EMT instruction executed.
TR	TRAP instruction executed.
TE	T-bit exception. T-bit was set, but setting was not

* Occurs only in RSX-11M systems equipped with memory management hardware (i.e., a mapped system).

Table 5-1 (Cont.)
ODT Error Codes

Code	Meaning
	caused by a breakpoint, single-step mode, a Proceed command, or a BPT instruction. Probably caused by improperly maintained task stack resulting in a word (that happens to have its bit 4 set) being used as the Processor Status Word (PS).
FP	Floating-point instruction error.
BE	Breakpoint instruction executed at unexpected location.

CHAPTER 6

TRACE DEBUGGING AID

6.1 INTRODUCTION

The Trace program is a debugging aid for use in the development and checkout of user-written tasks. Trace produces a listing that contains the register contents at the time each instruction in the user task is executed. Execution of an entire task can be traced, or up to four user-specified areas can be traced. Selective tracing allows the user to suppress the tracing of fully debugged subroutines and system routines to achieve output faster.

As a debugging aid, Trace complements the capabilities of ODT. Trace is not an interactive program. It is specified for a task during task building and is best suited for the debugging of relatively simple tasks. Trace is effective when a quickly-obtained listing showing the program's progress can be used to determine the cause of an error.

Two lines of register contents are printed on pseudo-device CL for each instruction executed in the user's task. The first line contains the contents of the following registers:

1. Current relative PC
2. Current PC
3. Next PC
4. PS
5. Directive Status word

The relative PC is computed by subtracting a user-specified bias from the actual PC. In this way, the trace output can be more easily compared with a module listing.

The second line contains the following register contents:

1. Contents of R0 through R5 and SP,
2. Contents of the top of the stack.

See Figure 6-1 for an example of Trace output.

Trace is included as an OBJ file named TRACE.OBJ under UFD [1,1] on the system disk.

```
000366  000566  000572  174020  000001
    000566  131574  051025  140750  000000  000000  000176  000444

000372  000572  000574  174020  000007
    000566  131574  051025  140750  000000  000000  000200  000000

000374  000574  000576  174020  000007
    000566  131574  051025  140750  000000  000000  000200  000000

000376  000576  001626  174020  000007
    000566  131574  051025  140750  000000  000000  000176  000602

001426  001626  000602  174020  000007
    000566  131574  051025  140750  000000  000000  000200  000000

000402  000602  000604  174020  000007
    000566  131574  051025  140750  000000  000000  000200  000000

000404  000604  001220  174020  000007
    000566  131574  051025  140750  000000  000000  000176  000610
```

Figure 6-1 Sample Trace Output

6.2 OPERATIONAL INFORMATION

The Trace module must be built into the task. Task Builder operation is described in the IAS, RSX-11D, and RSX-11M Task Builder Reference Manuals. Procedures specific to Trace are described below.

1. Assemble the program in the normal fashion.

2. Include the request for Trace in the command string to the Task Builder when building the task. Trace is requested as described in section 4.4, but here, specifying its disk area:

   IAS:

   ```
   PDS> LINK/DEBUG:[1,1]TRACE PROG1
   ```

   RSX:

   ```
   TKB>PROG1.TSK=PROG1.OBJ,[1,1]TRACE/DA
   ```

3. If desired, use the GBLPAT Task Builder option to specify the relative PC bias and one or more ranges to be traced. The GBLPAT option has the following formats for specifying a bias and ranges, respectively.

   ```
   GBLPAT=segname:.BIAS:bias value

   GBLPAT=segname:.RANGE:range/low:range/high: ... :range4low
          :range4high
   ```

segname = name of the task's root segment.

bias value = octal value to be subtracted from the actual PC to compute the relative PC. If a bias is not specified, the initial stack pointer is used.

range/low, range4low, = the low address, relative to the bias value, of any of up to four ranges. The address is expressed as an octal value.

range/high, range4high, = the high address, relative to the bias value, of any of up to four ranges. The address is expressed as an octal value.

NOTE

Both the low and high addresses of a range must be specified.

Example 1: The following GBLPAT options are used to restrict the tracing of PROG1 to a single module located at address 1364 (octal).

```
GBLPAT=PROG1:.BIAS:1364
GBLPAT=PROG1:.RANGE:0:214
```

The low and high range limits specified above are relative to the specified bias value.

Example 2: Task PROG2 consists of a single module (PROG2) and system subroutines linked above PROG2. The following GBLPAT excludes these subroutines from the trace.

```
GBLPAT=PROG2:0:1342
```

In this example, trace ranges are set relative to the default bias (initial SP). The high limit is the length of the module PROG2.

4. When task building is completed, run the task. Trace output is printed on CL.

APPENDIX A

PROCESSOR STATUS WORD

The Processor Status Word (PS), stored at hardware location 777776, contains information on the current status of the processor. The information contained in this location includes the current and previous operational modes of the processor (mapped system only), the current processor priority, an indicator which, when set, causes a trap upon completion of the current instruction, and condition codes describing the results of the last instruction executed. The format of the Processor Status Word is shown in Figure A-1 below.

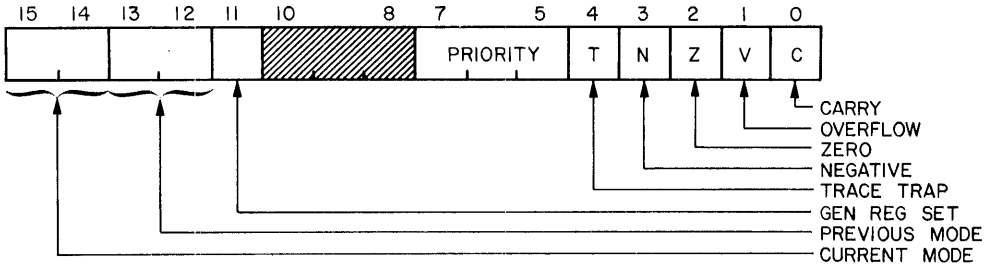

Figure A-1 Format of Processor Status Word

A.1 MODES (MEMORY MANAGEMENT OPTION)

Bits 15 and 14 of the Processor Status Word indicate the current processor mode, i.e., either User mode (11) or kernel mode (00). Bits 13 and 12 indicate the previous mode, i.e, the mode the machine was in (User or Kernel) prior to the last interrupt or trap.

User and Kernel modes afford a fully-protected environment for a multiprogramming system by providing the system with two distinct sets of processor stacks and memory segmentation registers for memory mapping. In user mode, a program is inhibited from executing a "HALT" instruction, and the processor will trap through location 4 (location 10 on PDP-11/40) if an attempt is made to execute this instruction. Furthermore, the processor will ignore the RESET instruction (and the SPL instruction on the PDP-11/45). In Kernel mode, the processor will execute all instructions.

A program operating in Kernel mode can map a user's program anywhere in memory and thus explicitly protect key areas (including the device

registers and the Processor Status Word) from the user operating environment.

A.2 PROCESSOR PRIORITY

The current priority of the central processor is maintained in bits 7 through 5 of the Processor Status Word. The central processor operates at any one of eight levels of priority (0 through 7). When the central processor is operating at level 7 (the highest priority), an external device cannot interrupt it with a request for service. The central processor must be operating at a lower priority than the external device's request in order for the interrupt to take effect. The eight processor levels provide an effective interrupt mask. On the PDP-11/45, these bits can be altered dynamically through the use of the SPL (Set Priority Level) instruction. This instruction is legal only in Kernel mode.

A.3 TRAP (T-BIT)

The trap bit (bit 4) can be set or cleared under program control. When set, a processor trap will occur through location 14 upon completion of the current user instruction, and a new Processor Status Word will be loaded. The trap (T) bit is especially useful in debugging programs, since it provides an efficient means for stepping through the task one instruction at a time. ODT uses the T-bit to execute instructions in the single-instruction mode (see section 3.7).

A.4 CONDITION CODES

The condition codes N, Z, V, and C (bits 3 through 0, respectively) contain information indicating the result of the last central processor operation. These bits are set as follows:

N=1, if the result was negative.
Z=1, if the result was zero.
V=1, if the operation resulted in an arithmetic overflow.
C=1, if the operation resulted in a carry from the most significant bit.

A.5 TRAP PROCESSING

Both interrupts and trap instructions automatically cause the previous Processor Status Word and program counter to be saved on the stack and replaced by the new values stored in the associated interrupt vectors. The interrupt vectors contain a new program counter value and new Processor Status Word. The user can cause the central processor to switch modes automatically (context switching), or disable the trap bit whenever a trap or interrupt occurs.

APPENDIX B

SEARCH ALGORITHMS

As described in section 3.8, ODT allows the user to search for specific bit patterns in the task or to identify words in the task which reference a specific location. The algorithms for these two types of search operations are described below.

B.1 WORD/BYTE SEARCHES (W or N)

The word search compares selected bits in a range of memory words with a user-specified search argument. The bits to be compared in the memory words and the search argument are defined through the search mask register (see section 3.8.1). If all the selected bits in any given memory word and the search argument are equal, a match has occurred, and ODT prints the "unmasked" task word.

The algorithm for the word search operation follows:

1. Fetch the memory word at the current address.

2. Perform the logical operation XOR (exclusive OR) on the memory word and the specified search argument.

3. Perform the logical operation AND on the result from Step 2 and the specified mask.

4. If a W search is being performed and the result of Step 3 is zero, or if an N search is being performed and the result of Step 3 is nonzero, print the address of the unmasked word and its contents.

5. Add two (2) to the current address, fetch the word in the next location, and go to Step 2. If, after adding 2 to the current address, the resulting value is greater than that contained in the high search limit register (see $H, Table 3-1), print the underline (_) prompting character and return control to the ODT command decoder routine to await the next user command.

B.2 EFFECTIVE ADDRESS SEARCH (E)

In the effective address search, ODT treats every task word within the specified search range as a value which has a possible direct relationship to the search argument, i.e., as a word which addresses a

specified location. Each address to be compared is first masked by the contents of the search mask register (see $M, Table 3-1) before comparison. Therefore, the contents of the mask register should normally be set to 177777 for the E command.

In the algorithm for the effective address search described below, the following symbology is used:

(X) = The contents of location X.

K = The effective address search argument.

1. Fetch the word at location X (the current location).

2. If (X) = K, i.e., if location X contains a value which is an absolute address reference to the search agrument (i.e., contains the search argument itself), print the contents of location X and go to Step 5.

3. If (X)+X+2 = K, i.e., if the contents of location X, as indexed by the contents of the program counter (PC), are equal to the search argument, print the contents of location X and go to Step 5.

4. If low-order (X)x2+X+2 = K, i.e., if the contents of the low-order byte at location X reference a location whose address value is the same as the search argument, print the contents of location X, and go to Step 5.

5. Add 2 to the current address. If the resulting value is greater than that contained in the high search limit register (see $H, Table 3-1), print the underline (_) prompting character and return control to the ODT command decoder routine to await the next user command; otherwise, go to Step 1.

INDEX

IAS/RSX-11
ODT Reference
Manual
DEC-11-OIODA-B-D

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. Problems with software should be reported on a Software Performance Report (SPR) form. If you require a written reply and are eligible to receive one under SPR service, submit your comments on an SPR form.

Did you find errors in this manual? If so, specify by page.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

Is there sufficient documentation on associated system programs required for use of the software described in this manual? If not, what material is missing and where should it be placed?

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Non-programmer interested in computer concepts and capabilities

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code _____
or
Country

Please cut along this line.

Do Not Tear - Fold Here and Tape

digital

No Postage Necessary if Mailed in the United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

RT/C SOFTWARE PUBLICATIONS TW/A14
DIGITAL EQUIPMENT CORPORATION
1925 ANDOVER STREET
TEWKSBURY, MASSACHUSETTS 01876

Do Not Tear - Fold Here

Cut Along Dotted Line