TOPS–20
DDT Manual

AA–M273A–TM

May 1985

This manual describes the use of TOPS–20 DDT, the Dynamic Debugging Tool for MACRO–20 programs.

This is a new document.

OPERATING SYSTEM: TOPS–20 V6.1

SOFTWARE: DDT V43

Software and manuals should be ordered by title and order number. In the United States, send orders to the nearest distribution center. Outside the United States, orders should be directed to the nearest DIGITAL Field Sales Office or representative.

Northeast/Mid–Atlantic Region

Digital Equipment Corporation
PO Box CS2008
Nashua, New Hampshire 03061
Telephone:(603)884–6660

Central Region

Digital Equipment Corporation
Accessories and Supplies Center
1050 East Remington Road
Schaumburg, Illinois 60195
Telephone:(312)640–5612

Western Region

Digital Equipment Corporation
Accessories and Supplies Center
632 Caribbean Drive
Sunnyvale, California 94086
Telephone:(408)734–4915

First Printing, May 1985

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.

CONTENTS

CONTENTS

PREFACE

MANUAL OBJECTIVES AND AUDIENCE

This manual explains and illustrates the features of TOPS-20 DDT, the debugger for MACRO-20 programs. Although TOPS-20 DDT can be used to debug the compiled code of programs written in higher-level languages, this manual illustrates only the use of TOPS-20 DDT to debug programs written in MACRO-20.

This manual is both an introduction to the basic functions of TOPS-20 DDT and a reference guide to all TOPS-20 DDT commands and functions.

This manual assumes that the reader is familiar with TOPS-20, has done some programming in MACRO-20, and is familiar with the format of MACRO-20 instructions.

STRUCTURE OF THIS DOCUMENT

This manual has 12 chapters, one appendix, and one glossary.

- Chapter 1 introduces the concept of symbolic debugging and describes the variants of TOPS-20 DDT.
- Chapter 2 describes loading TOPS-20 DDT with your program, discusses basic TOPS-20 DDT commands, and illustrates a sample debugging session.
- Chapter 3 explains the syntax of a DDT command. Chapter 3 also describes expressions to enter data and explains how TOPS-20 DDT evaluates expressions.
- Chapter 4 discusses how to examine and modify a program using TOPS-20 DDT.
- Chapter 5 describes the use of TOPS-20 DDT to control program execution: how to start, stop, and monitor the running of a program.
- Chapter 6 explains how to perform searches of a program's address space using TOPS-20 DDT.
- Chapter 7 discusses the manipulation of program symbols using TOPS-20 DDT.

- Chapter 8 describes how to use the TOPS-20 DDT patching function to insert and test a new series of instructions in your program without reassembling the program.
- Chapter 9 describes the use of FILDDT.
- Chapter 10 describes the use of the privileged DDTs; KDDT and MDDT.
- Chapter 11 describes special-use commands that control physical and virtual addressing. These commands are useful primarily when running EDDT and FILDDT.
- Chapter 12 describes commands that are used when debugging programs that use extended addressing.
- Appendix A explains DDT and FILDDT error messages.
- The glossary defines important TOPS-20 DDT terms.

OTHER DOCUMENTS

Other documents to which the reader should have access are:

- MACRO Assembler Reference Manual
- TOPS-20 LINK Reference Manual
- TOPS-20 Commands Reference Manual
- DECsystem-10/DECSYSTEM-20 Processor Reference Manual
- TOPS-10/TOPS-20 RSX-20F System Reference Manual

CONVENTIONS

The following conventions are used in this manual in the description of DDT commands and concepts.

{}	Curly brackets (braces) indicate that the enclosed item is optional.
. (period)	The address contained in DDT's location counter; also called the current location.
addr	A symbolic location within a program, a symbolic or absolute address in memory, an AC, or ".", the current location.
c	A single ASCII or SIXBIT character.
expr	Any expression that is legal in DDT.
filnam	One or more components of a file specification.

instr	Any instruction in the PDP-10 (KL10) machine instruction set.
location sequence stack	A circular stack of memory locations that is used to store the addresses of certain previously referenced locations.
n	A numeric argument.
page	A page in memory. A page equals 512 words of memory.
symbol	A symbol name of up to 6 RADIX50 characters.
text	Any string of ASCII or SIXBIT characters.
word	Any 36-bit value occupying one word of memory.
<ESC>	Represents pressing the ESCAPE or ALTMODE key once.
<ESC><ESC>	Represents pressing the ESCAPE or ALTMODE key twice.
<CTRL/X>	Represents pressing a key (represented by X) at the same time as you press the key labeled CTRL.
<BKSP>	Represents pressing the BACKSPACE key or <CTRL/H>.
<LF>	Represents pressing the LINE FEED key.
<RET>	Represents pressing the RETURN key.
<TAB>	Represents pressing the TAB key or <CTRL/I>.

Numbers are in octal radix unless otherwise specified.

Examples of interaction between the user and DDT show user input in lowercase and DDT output in uppercase.

The symbols <BKSP>, <CTRL/x>, <ESC>, <LF>, <RET>, and <TAB> always represent user input.

NOTE

The descriptions of many DDT commands list the actions and effects of those commands. The actions and effects may not occur in precisely the order specified, but this has no effect on the results.

CHAPTER 1

INTRODUCTION TO DDT

DDT is a utility program for debugging MACRO-20 programs. This manual describes how to use DDT.

1.1 SYMBOLIC DEBUGGING

It is sometimes difficult to understand precisely the operation of a program by reading the source code. DDT is a tool for interactively examining the operation of a MACRO-20 program while it is running. DDT is useful for finding programming errors (bugs) in programs that do not run correctly. You can also use DDT to analyze the flow of control in a program that is to be revised or rewritten.

With DDT, you can interrupt the execution of your program at locations (breakpoints) you choose, and then examine and modify the program's address space as required. You can execute instructions one-by-one to check whether the effect of each instruction is what is intended. You can then set other breakpoints in your program before continuing execution.

When you refer to program locations and values, DDT allows you to use the symbols that are defined in the program rather than absolute values and addresses. This makes it much easier to refer to the source listing and to find specific locations in memory.

After modifying the program's instructions or data, you can exit DDT and save (with the EXEC's SAVE command) the changed version of the program for further testing.

1.2 TOPS-20 VARIANTS OF DDT

There are several variants of DDT, each useful under specific circumstances or for specific tasks.

The variants of TOPS-20 DDT are:

- EDDT
- FILDDT
- KDDT
- MDDT
- RDDT
- SDDT
- UDDT
- XDDT

EDDT is used to debug programs that run in executive mode (such as BOOT), and is described in Chapter 10.

FILDDT is used to examine and patch disk files and structures. You can also use FILDDT to examine the running monitor. FILDDT is described in Chapter 9.

KDDT is used to debug and patch monitor .EXE files and the running monitor, and is described in Chapter 10.

MDDT is used to debug and patch the running monitor, and is described in Chapter 10.

RDDT is a relocatable variant of DDT that can be used to debug programs in user mode. If your program is in memory (and has been loaded with RDDT as below), you invoke RDDT by typing in (at TOPS-20 command level):

```
START
```

You load RDDT with your program by running LINK as follows:

```
@LINK
*MYPROG,SYS:RDDT.REL/GO
```

where MYPROG is the name of your program. Loading RDDT.REL with your program does not prevent you from using other LINK features. You must load RDDT.REL last, or its start address is lost. RDDT.REL is useful in situations where you do not wish to have DDT loaded at its default location.

This example shows only the minimal steps required to load the relocatable DDT with your program. See the LINK Reference Manual for further information about using LINK.

SDDT is a "stub" that places XDDT in its own section, with system symbols defined as in MONSYM and MACSYM. SDDT is the DDT variant invoked when, at TOPS-20 command level, you type in

```
SDDT
```

SDDT exists so that typing in SDDT will invoke DDT version 43 in the same manner as previous versions.

UDDT is a "stub" that resides in your user program's section if the program has a TOPS-10-style entry vector and the program entry vector is in section zero. This is done for compatibility with programs that use locations 770000, 770001 and 770002. If you load a program in section zero and the program has a TOPS-10-style entry vector, when you use the DDT command, the EXEC loads the UDDT stub into your program's section at address 770000. UDDT then loads XDDT into the highest-numbered free (nonexistent) section (if XDDT is not already loaded), and starts XDDT.

XDDT is the DDT variant normally used to debug user programs. If you load your program in a nonzero section or the program does not have a TOPS-10-style entry vector, the DDT command causes the EXEC to load XDDT directly into the highest-numbered free section. XDDT is also invoked by the SDDT and UDDT stubs. If you type in XDDT while at TOPS-20 command level, the EXEC loads XDDT into section zero, with system symbols defined.

CHAPTER 2

GETTING STARTED WITH DDT

2.1 INTRODUCTION

This chapter describes how to load DDT with your program and shows how to perform basic DDT functions. It then illustrates a sample session debugging a simple MACRO-20 program using basic DDT functions. You can debug programs using only the DDT commands described in this chapter. Once you are familiar with these commands, you may wish to learn how to use the commands and functions described in the rest of the manual to perform more sophisticated debugging.

The commands used in this chapter are described only in sufficient detail for the debugging task being performed; all commands are thoroughly described in Chapters 3 through 12 of this document.

The best way to learn is by doing. You will learn the commands and techniques discussed in this manual if you use them as you read about them. If you have a MACRO-20 program that you wish to debug, use it to practice the commands discussed here. If not, type in the program X.MAC listed in Figure 2-1.

2.2 LOADING DDT

If your program is already in memory (for example, as a result of using LINK or the GET command), load DDT with the TOPS-20 DDT command. If you have run your program, and it has terminated normally or abnormally, you can also use the DDT command to load DDT. When DDT is loaded, it displays

```
DDT
```

on your terminal.

2.3 BASIC FUNCTIONS

You must be able to perform certain basic functions to debug a program interactively. Basic DDT functions are:

- accepting your commands
- deleting incorrect commands
- starting the program
- stopping the program at specified locations
- examining and modifying memory
- executing program instructions one at a time
- continuing execution of the program
- exiting DDT

You give commands to tell DDT what functions to perform. DDT does not wait for a line terminator (such as a carriage return) to indicate the end of your command. Instead, DDT reads your commands character by character as you type them in. When you type in a DDT command, you almost never have to press the RETURN key. This manual explicitly indicates the occasions when a command requires you to press the RETURN key.

NOTE

You must press the ESCAPE key as part of typing in many DDT commands. This manual uses the symbol <ESC> to indicate where you press the ESCAPE key. When you press the ESCAPE key, DDT displays a dollar sign ($) on the screen. DDT never displays <ESC> when you press the ESCAPE key.

NOTE

This manual uses the character sequences <BKSP>, <ESC>, <LF>, <RET>, and <TAB> as symbols to indicate where you press the BACKSPACE, ESCAPE, LINE FEED, RETURN, and TAB keys, respectively. This manual also uses the symbol <CTRL/X> to indicate where you simultaneously press the CONTROL key and the key indicated by X. These symbols ALWAYS indicate where you press the keys noted here. You NEVER type the individual characters <BKSP>, <ESC>, <LF>, <RET>, <TAB>, or <CTRL/X>, to type in a DDT command.

Your commands appear on the screen as you type them. Use the DELETE key to delete partially typed-in commands character by character. If you try to delete more characters than you have typed in, DDT displays

```
XXX
```

and waits for your next command.

You can delete an entire command line with <CTRL/U>. When you type <CTRL/U> DDT displays

```
XXX
```

To exit DDT, press

```
<CTRL/Z>
```

The other basic DDT functions are described in the rest of this chapter.

2.3.1 Error Conditions

If DDT cannot execute a command, it displays a message to let you know. The message may be only a single character (such as M or U, for Multiply-defined symbol or Undefined symbol), a question mark (?), or a complete message string. For most errors, DDT also sets a pointer to the error string, so that if DDT does not display it, you can type in a command to display the error string. You can display the error string at any time before another error occurs, at which time DDT updates the pointer (to point to the new error string). To display the error string produced by the last DDT error, type in

```
<ESC>?
```

(press the ESCAPE key, followed by a question mark). You can also display the last TOPS-20 process error. To do this, type in

```
<ESC><ESC>?
```

(press the ESCAPE key twice, followed by a question mark).

See Appendix A (Error Messages) for more information about DDT error messages and commands to give information about error conditions.

2.3.2 Basic Concepts

A very important DDT concept is that of the current location. The current location is a memory location that you have referenced, either implicitly or explicitly, with your last command, and that is the default point of reference of your next command. You can think of the current location as the location "where you are." The symbol "." (period) refers to the address of the current location, and can be used as an argument in DDT commands.

The location counter is a DDT pointer that contains the address of the current location. The location counter performs a function similar to that of a bookmark. Some DDT commands display the contents of a specific location but do not change the address of the current location, in order to maintain a specific point of reference for your next command. Most DDT commands change the address of the current location, and therefore also change the location counter. The commands that do not change the current location are so indicated.

The open location is a memory location that you can modify with your next command. DDT "opens" a location as a result of most commands you give to examine or modify memory. These commands often also change the current location to the word that is "opened", so that the open location is usually also the current location. There is never more than one location open at any given time.

To find the symbolic address of the current location, type in

```
._          (a period followed by an underscore)
```

This causes DDT to display the following:

```
ADDR1+n
```

where ADDR1 is a label defined in your program, and n is the offset of the current location from that label (if the current location is ADDR1, DDT does not display +0).

Another important DDT concept is that of the current quantity. This is a value that represents the contents of the last word that DDT displayed, or the value that you last deposited in memory. The current quantity is the most recent of those values. Many DDT commands take arguments that default to the current quantity.

The location sequence stack is a DDT storage area for the addresses of previous current locations. Certain DDT commands that change the current location store the address of the current location on the location sequence stack. Other DDT commands change the address of the current location to an address that has already been stored on the location sequence stack. The location sequence stack functions similarly to inserting place-markers in a source code listing, in order to return to prior references.

2.3.3 Starting and Stopping the Program

When your program is loaded and DDT is ready to accept your commands, as indicated by DDT appearing on the terminal display, you can begin execution of your program at its start address by typing in

```
<ESC>G
```

Unless you set one or more breakpoints before you start the program, your program runs either to completion or until it commits a fatal error. A breakpoint is a memory location in a program's executable code that has been modified so that if the program attempts to execute the instruction at that location, control passes to DDT before the instruction is executed.

The command to set a breakpoint is

```
addr<ESC>B
```

where addr is the address at which to stop execution. If the user-program PC reaches addr, DDT interrupts execution of the program before the program executes the instruction at the specified address. When DDT interrupts program execution at a breakpoint, DDT opens the breakpoint, and the breakpoint becomes the current location.

While program execution is stopped at a breakpoint, you can display and change the contents of instruction and data words, remove breakpoints, set new breakpoints, and execute instructions one at a time (single-step). As you examine memory, you may find an instruction that is incorrect, and modify it. You can also examine and modify data words in memory. After modifying incorrect instructions and data in memory, you can immediately execute the instructions to check the effects of the modifications, without having to reassemble the source code.

Once you have made your changes, you can continue program execution at the place where execution was interrupted, restart the program at the beginning, or start execution at any other location you choose. The program will run to completion unless it reaches a breakpoint or commits a fatal error.

2.3.4 Examining and Modifying Memory

One command you can use to examine memory is

```
addr/
```

where addr is the address of the memory word you wish to examine (display), and can be numeric or symbolic. DDT displays the contents of the word located at addr. If the opcode field (bits 0-8) of the memory word matches a recognized instruction or user-defined OPDEF, DDT displays the contents of addr as an instruction (or OPDEF). If DDT finds (in the symbol table) any of the values to be displayed, DDT displays those symbols rather than the numeric values. For example, either of the following display lines might appear on your terminal, depending on the address and contents of the word:

```
ADDR1/   MOVE 2,SYM1

ADDR1+5/   SYM1,,SYM2
```

where ADDR1, SYM1, and SYM2 have been defined in the program.

If you type in a symbol that DDT does not find in the symbol table, DDT sounds the terminal buzzer or bell, and displays U on the screen. If you type in a symbol that is defined as a local symbol in more than one module, DDT sounds the terminal buzzer or bell and displays M. You can eliminate the multiply-defined symbol problem by "opening" the symbol table of the module in which the correct symbol is defined. See Chapter 7 (Manipulating Symbols in DDT) for more information.

When searching for a symbol to display, DDT uses global symbols in preference to local symbols. However, DDT searches the "open" symbol table first, and treats local symbols found in the open symbol table as global symbols. If DDT finds only a local symbol that is not in the open symbol table, DDT appends a number sign (#) to the symbol when displaying it. For example, DDT might display

```
ADDR/   MOVE 2,SYM1#
```

See Chapter 7 (Manipulating Symbols in DDT) for more information on symbols and symbol tables.

The command addr/ changes the current location to addr and opens the word at addr. If you omit addr from an examine-memory command, such as addr/, DDT uses the current quantity to determine the address of the location to display. For example, after DDT displays the contents of ADDR1+5 as above, if you type in /, DDT displays the contents of the word located at SYM2. The display line then appears:

```
ADDR1+5/   SYM1,,SYM2   /   VALUE
```

where VALUE is the contents of the word located at SYM2. By default, DDT displays VALUE symbolically if it can.

The command / by itself (without addr) does not change the current location. Both forms of the / command open the location displayed, enabling you to modify the location with your next command.

Another very useful command for examining memory is <TAB>. This command changes the current location to addr (whether given or defaulted), opens the location at addr, and changes the current quantity to the contents of addr. <TAB> also starts a new display line before displaying the contents of addr, making the display easier to read. For example, if you type in <TAB> after DDT displays the address and contents of ADDR1+5 (as above) on your terminal, the terminal display appears:

```
ADDR1+5/   SYM1,,SYM2   <TAB>
SYM2/   VALUE
```

where VALUE is the contents of the word located at SYM2. The current location is SYM2, which is open. (<TAB> does not appear on the screen, but is shown above to indicate where you press the <TAB> key.)

<TAB> also stores the address of the current location (ADDR1+5) on the location sequence stack before changing the current location to the location just displayed (SYM2). DDT uses the location sequence stack to "remember" previous values of the location counter. To "get back" to the previous current location, type in

```
<ESC><RET>
```

In the above example, after you press <TAB> at ADDR1+5, DDT displays the contents of SYM2 and changes the current location to SYM2. When you type in <ESC><RET>, DDT changes the current location to ADDR1+5, opens the location at ADDR1+5, and again displays the contents of ADDR1+5. The terminal display then appears as follows:

```
ADDR1+5/   SYM1,,SYM2   <TAB>
SYM2/   VALUE   <ESC><RET>
ADDR1+5/   SYM1,,SYM2
```

If you use the command addr<TAB>, DDT deposits addr in the open location (if there is one) and closes the location before opening the location at addr and displaying its contents. <TAB> by itself does not deposit anything, but does save the current location on the location sequence stack, making <TAB> more useful than /, depending on what you do next.

You can display and open the word after the current location by typing in

```
<LF>
```

DDT changes the current location to the next word in memory, starts a new line, displays the address of the (new) current location, displays the contents of the current location, and opens the current location. DDT displays the address as a symbol or a symbol plus an offset, if it can find a corresponding symbol in the symbol table. For example, to display the next word in memory after ADDR1+5, type in

```
<LF>
```

DDT changes the current location to ADDR1+6, starts a new line, and displays the address and contents of ADDR1+6. The screen display then appears as follows:

```
ADDR1+5/   SYM1,,SYM2   <LF>
ADDR1+6/   -1,,SYM3
```

Note that DDT does not display the characters <LF>. <LF> does not affect the location sequence stack.

Typing in another <LF> causes DDT to display and open the next word.

To display and open the word previous to the current location, type in

```
<BKSP>
```

DDT changes the current location to the previous word, starts a new line, displays the address and contents of the (new) current location, and opens the current location. <BKSP> does not affect the location sequence stack. DDT also displays

```
^H
```

to indicate that you pressed <BKSP>. For example, if you type in <BKSP> to open and display the word before ADDR1+5, the screen appears as follows:

```
ADDR1+5/   SYM1,,SYM2   ^H
ADDR1+4/   -3,,SYM2
```

To change the contents of the open location, type in

```
expr<RET>
```

where expr can be an instruction, a symbol, or a numeric expression. For example, if you type in the command LABL2/, DDT displays the contents of the memory word at LABL2, and "opens" that word. If the word at LABL2 contains

```
MOVE 1,SYM1
```

and you wish to change SYM1 to SYM2, type in

```
MOVE 1,SYM2<RET>
```

DDT stores the new instruction in the location at LABL2 and "closes" the location. DDT does NOT display <RET>. The terminal display appears as follows (your input is in lowercase):

```
labl2/   MOVE 1,SYM1   move 1,sym2<RET>
```

The current location is still LABL2, but there is no open location. To check whether the instruction is now correct, you can type in

```
./
```

DDT displays the contents of the current location, and the screen display now appears (your input is in lowercase):

```
labl2/   MOVE 1,SYM1   move 1,sym2<RET>
./   MOVE 1,SYM2
```

After typing in a command to display and open a location, if you type in

```
value<LF>
```

DDT stores the new value in the open location, as for value<RET>. DDT then changes the current location to the next location in memory, starts a new display line and displays the address and contents of the (new) current location. DDT also opens the current location. The example above would then appear as follows (your input is in lowercase):

```
labl2/   MOVE 1,SYM1   move 1,sym2<LF>
LABL2+1/   CONTENTS
```

where CONTENTS is the value stored at LABL2+1. Similarly, after opening a location, if you type in

```
value<BKSP>
```

DDT stores the new value in the open location, and changes the current location to the previous location in memory. DDT then starts a new display line, displays the address and contents of the current location, and opens the current location.

2.3.5 Executing Program Instructions

When you have interrupted program execution at a breakpoint, you can execute the next instruction by typing in

```
<ESC>X
```

DDT executes the instruction, displays the results of executing the instruction, and displays the address and contents of the next instruction to be executed. If you have not begun program execution with the <ESC>G command, the <ESC>X command is illegal, and DDT displays ? and sounds the terminal buzzer or bell. <ESC>X changes the current location to the next instruction to be executed. For example, assume that the next instruction to be executed is located at LABEL1, which contains

```
MOVE 1,VARIBL
```

If the word at VARIBL contains SYM1, when you type in <ESC>X, DDT starts a new line and displays

```
   1/   SYM1   VARIBL/   SYM1
LABEL1+1/   instr
```

where instr is the contents of LABEL1+1, and is the next instruction to be executed. You can continue to execute instructions one at a time by typing in successive <ESC>X commands. This is known as single-stepping.

If instr is a call to a subroutine, such as

```
PUSHJ P,SUBRTN
```

you can execute the subroutine without single-stepping each instruction in the subroutine. Type in

```
<ESC><ESC>X
```

and DDT executes the subroutine. If the program does not commit a fatal error before the user-program PC returns to a location +1, +2, or +3 from the instruction that calls the subroutine, DDT displays the address and contents of the location where the program returns and waits for your next command. DDT also changes the current location to the address of the next instruction to be executed. If the PC does not return to a location +1, +2, or +3 from the instruction that calls the subroutine, the program runs until it commits a fatal error, or until it completes normally.

Because DDT controls the execution of each instruction, the subroutine can take a long time to complete. You can check DDT's progress through the subroutine by typing

```
?
```

DDT responds by displaying

```
Executing:  ADDR/   INSTR
```

where ADDR is the address of the next instruction to be executed, and INSTR is the instruction at ADDR.

NOTE

If a subroutine begins at a location that is +1, +2, or +3 from the instruction that calls the subroutine, DDT returns control to you at the first instruction of the subroutine, without executing the subroutine.

To continue execution of the program until the next breakpoint or until program completion, type in

<ESC>P

DDT starts the program running again, beginning with the next instruction to be executed. If you did not single-step any instructions, the program begins by executing the instruction at the breakpoint. If you have executed any instructions by single-stepping, the program continues where you stopped. The effect is as if the program were running without DDT in control. If the program reaches a breakpoint, DDT again interrupts execution. You can then examine and modify memory, set and remove breakpoints, and continue execution.

2.4 A SAMPLE DEBUGGING SESSION USING DDT

This section describes a debugging session using DDT. The program being debugged is X.MAC, shown in Figure 2-1. The program and the sample session are for illustration only. There are many styles of programming and debugging, and these examples are descriptive rather than prescriptive in intent.

You will understand this section and learn the commands described more easily if you type in the program listed in Figure 2-1 and use the commands as they are described. The sample program in Figure 2-1 is designed to pass the address of a table to a subroutine. The table contains three elements. The subroutine is to add the first two elements of the table and store the result in the third element before returning to the main program. There are no input or output routines in the program. The table is initialized using DDT, and the result is checked while in DDT.

```
        SEARCH  MONSYM
        TITLE   X
R0=0                                ;AC0
IDX=6                               ;INDEX REGISTER
P=17                                ;STACK COUNTER
START:: MOVE    P,PWORD             ;Set up stack counter
        MOVEI   IDX,TABLE1          ;Address of table with X & Y
        PUSHJ   P,ADDEM             ;Do the addition
        MOVEI   IDX,TABLE1          ;Address of table
        MOVE    R0,ANSWER(IDX)      ;Answer to R0
        JFCL    0
        HALTF%                      ;All done!
ADDEM:  MOVE    R0,X(IDX)           ;Load X
        ADD     R0,Y(IDX)           ;X + Y
        MOVE    R0,ANSWER(IDX)      ;Store answer
        POPJ    P,                  ;Return
TABLE1: BLOCK   3                   ;3 words
X==0                                ;Offset for X
Y==1                                ;Offset for Y
ANSWER==2                           ;Offset for answer
STKSIZ==10                          ;Stack size
PWORD:  IOWD STKSIZ,STACK           ;Stack pointer
STACK:  BLOCK   STKSIZ              ;Stack
        END     START
```

Figure 2-1: Sample Program X.MAC

Figure 2-2 is an annotated session debugging X.MAC, the program in Figure 2-1. In the annotated session, the DDT terminal display is on the left, user input is in the center in lowercase, and explanatory comments about the session are on the right. This is not always the way it appears on the terminal. Figure 2-3 shows the session as it actually appears on the terminal.

Note that DDT does not display the AC field of an instruction if it is zero. This means that if your program contains the instruction MOVE R0,LABL1, where R0=0, DDT displays the instruction as MOVE LABL1.

NOTE

DDT does not display <LF>, <RET>, or <TAB>. These are shown in the sample session to indicate user input.

Screen Display	User Input	Explanation
@		TOPS-20 prompt.
	debug x<RET>	Begin the session by typing in "debug x<RET>", where x is the name of your MACRO program.
MACRO: X program LINK: Loading [LNKDEB DDT execution] DDT displays the "DDT" prompt.		MACRO reassembles your (if needed), and LINK loads your program with DDT. DDT
	start/	Begin examining code at label "START".
MOVE P,PWORD#		DDT displays the instruction at START.
	<LF>	Press <LF> to display the next instruction.
.JBDA+1/ MOVEI IDX,TABLE1#		The first symbol in this program happens to coincide with .JBDA, a JOBDAT symbol. When DDT scans the symbol table, it finds .JBDA before it finds START, and displays .JBDA instead. DDT still accepts START as an input symbol. Also note the number sign (#) appended to TABLE1 and PWORD. A number sign means that PWORD and TABLE1 are local symbols that are not in the open symbol table.

	.jbda<ESC>k	Type in .jbda<ESC>k to suppress DDT typeout of symbol .JBDA. DDT will display START rather than .JBDA from now on. See Chapter 7 (Manipulating Symbols is DDT) for more information.
	x<ESC>:	Type in the module name (X) followed by <ESC> and a colon to open the symbol table associated with X. DDT will not append any more number signs.
	<TAB>	Press <TAB> to start a new display line, evaluate the current quantity as if it were an instruction, and display the contents of the location addressed by the Y field of the instruction. (Typing in / (slash) displays the same word as <TAB>, but does not start a new line.) <TAB> also saves your place (like a bookmark) on the location sequence stack, so you can get back here easily.
TABLE1/ 0		When you type in the <TAB> command, DDT displays the address and the contents of the location. The first element of the table contains zero. The <TAB> command also opens the location.
	2<LF>	Type in "2" followed by <LF> to deposit the value "2" in the first element, and to open and display the second element.
TABLE1+1/ 0		The second element contains zero.
	3<LF>	Type in "3" followed by <LF> to deposit the value "3" in the second element and open and display the third element. The addition to be performed by the program is 2+3.
TABLE1+2/ 0		The third element (the answer) contains zero.

	<ESC><RET>	Press <ESC>, then press <RET> to return to the address you saved on the location sequence stack with the <TAB> command, above.
START+1/ MOVEI IDX,TABLE1		DDT displays the address and contents of the last location you displayed before you typed in <TAB>.
	<LF>	Press <LF> to look at the next location.
START+2/ PUSHJ P,ADDEM		This is the call to the subroutine that does the computation.
	.<ESC>b	Type in ".", press <ESC>, and type in "b" to set a breakpoint at the current location.
	<ESC>g	Type in <ESC>g to start program execution.
$1B>>START+2/ PUSHJ P,ADDEM		DDT displays the breakpoint number, the address of the breakpoint, and the instruction at the breakpoint. This instruction has not yet been executed.
	<ESC><ESC>x	Press <ESC> twice, then type in "x" to let DDT execute the subroutine.
START+3/ MOVEI IDX,TABLE1		DDT returns from the subroutine at the next instruction, and displays the address and contents of the instruction. If there is a "skip return," DDT displays "<SKIP>" if the program skipped one instruction. If the program skips 2 or 3 instructions, DDT displays "<SKIP n>", where n is the number of instructions skipped.
	<ESC>x	Press <ESC> and type in "x" to execute the instruction.
IDX/ TABLE1 TABLE1		DDT displays the address and contents of IDX (the result of executing the instruction), and also displays "TABLE1" (the result of evaluating the effective address of the instruction).

START+4/ MOVE 2(IDX)		DDT then starts a new line and displays the address and contents of the next instruction. Note that DDT does not display the zero in the AC field of the instruction.
	<ESC><TAB>	Press <ESC>, then <TAB> to display the contents of the location addressed by the instruction, using any indexing and indirection. (If you omit <ESC>, DDT uses only the Y field, without indexing and indirection.)
TABLE1+2/ 0		The location addressed by the instruction is TABLE1+2, which contains zero. This is the table element that contains the answer, which should be 5.
	<BKSP>	Press <BKSP> to see the previous element in the table (DDT echoes <BKSP> as ^H).
TABLE1+1/ 3		This element contains 3. That is correct.
	<BKSP>	Press <BKSP> again to check the previous element.
TABLE1/ 2		This element contains 2. That is also correct. One way to find the error is to single-step through the program.
	start<ESC>b	Type in "start", press <ESC>, and type in "b" to set a breakpoint at the beginning of the program.
	<ESC>g	Press <ESC> and type in "g" to start the program again.
$2B>>START/ MOVE P,PWORD		DDT displays the breakpoint number, and the address and contents of the instruction at the breakpoint.
	<ESC>x	Press <ESC>, then type in "x" to execute the instruction. This instruction moves a memory word to a register.

```
   P/   -10,,PWORD   PWORD/   -10,,PWORD
```

DDT displays the address and new contents of the register, and the address and contents of the memory word.

```
START+1/   MOVEI IDX,TABLE1
```

DDT then displays the address and contents of the next instruction.

```
                          <ESC>x
```

Press <ESC>, then type in "x" to execute this instruction, which moves an immediate value to a register.

```
   IDX/   TABLE1   TABLE1
```

DDT displays the address and new contents of the register, and the immediate value.

```
START+2/   PUSHJ P,ADDEM
```

DDT then displays the address and contents of the next instruction. Note that when you reach a breakpoint by single-stepping, DDT does not display the breakpoint number.

```
                          <ESC>x
```

Press <ESC>, then type in "x" to execute the instruction.

```
   P/   -7,,STACK
```

DDT displays the address and new contents of the stack pointer used by the PUSHJ.

```
<JUMP>
```

DDT displays "<JUMP>" if the change in PC is less than one or greater than 4.

```
ADDEM/   MOVE 0(IDX)
```

DDT displays the address and contents of the next instruction to be executed.

```
                          <ESC>x
```

Press <ESC> and type in "x" to execute the instruction.

```
   0/   2   TABLE1/   2
```

The instruction moved the contents of the word at TABLE1 (which is 2) to AC0.

```
ADDEM+1/   ADD 1(IDX)
```

DDT displays the next instruction.

```
                          <ESC>x
```

Press <ESC> and type in "x" to execute the instruction.

```
   0/    5   TABLE1+1/    3
```

The instruction added the contents of the word at TABLE1+1 (which is 3) to AC0, which now contains 5. OK.

```
ADDEM+2/    MOVE 2(IDX)
```

DDT displays the next instruction.

```
                          <ESC>x
```

Press <ESC> and type in "x" to execute the instruction.

```
   0/    0   TABLE1+2/    0
```

The instruction moved the contents of the word at TABLE1+2 to AC0. The MOVE instruction at ADDEM+2 should be MOVEM.

```
ADDEM+3/    POPJ P,0
```

DDT displays the next instruction (as a result of the <ESC>x).

```
                          <BKSP>
```

Press <BKSP> to display and open the location with the incorrect instruction.

```
ADDEM+2/    MOVE 2(IDX)
```

DDT displays the previous instruction. This is the incorrect instruction.

```
                movem r0,answer(idx)<RET>
```

Type in the new instruction and press <RET>.

```
                          ./
```

Check the current location to see what you deposited.

```
MOVEM 2(IDX)
```

Looks OK.

```
                          .<ESC>b
```

Set a breakpoint at ".", the current location.

```
                          <ESC>g
```

Restart the program at the beginning.

```
$2B>>START/    MOVE P,PWORD
```

DDT displays the breakpoint information.

```
                          <ESC>p
```

Press <ESC> and type in "p" to proceed from breakpoint 2 to the next breakpoint.

```
$1B>>START+2/    PUSHJ P,ADDEM
```

DDT displays the breakpoint information.

```
                          <ESC>p
```

Proceed from breakpoint 1.

```
$3B>>ADDEM+2/    MOVEM 2(IDX)
```

DDT displays the breakpoint information. This is the instruction you changed.

```
                          <ESC>x
```

Single-step the instruction to watch what it does.

```
   0/   5   TABLE1+2/   5                    The instruction moves the
                                             contents of AC0 to the word
                                             at TABLE1+2.  OK!!

ADDEM+3/   POPJ P,0                          DDT also displays the
                                             address and contents of the
                                             next instruction.

                      start+4<ESC>b          Set a breakpoint at START+4
                                             to check the results.

                          <ESC>p             Proceed from breakpoint 3.

$4B>>START+4/   MOVE 2(IDX)                  DDT displays the breakpoint
                                             information.

                          <ESC>x             Single-step the instruction.

   0/   5   TABLE1+2/   5                    The instruction moves the
                                             contents of the word at
                                             TABLE1+2 to AC0.  The new
                                             value of AC0 is 5.  OK!

START+5/   JFCL 0                            DDT displays the address and
                                             contents of the next
                                             instruction.

                          <CTRL/Z>           Quit.

@                                            You are now back at TOPS-20
                                             command level.
```

Figure 2-2: Annotated Debugging Session

Figure 2-3 shows the session as it actually appears on the terminal. Again, user input is in lowercase. Comments on the right indicate where you type in characters that do not echo.

```
      Screen Display                                          Comments

   @debug x
   MACRO:  X
   LINK:   Loading
   [LNKDEB DDT execution]
   DDT
   start/   MOVE P,PWORD#                                    Type in <LF>.
   .JBDA+1/   MOVEI IDX,TABLE1   .jbda$k   x$:               Type in <TAB>.
   TABLE1/   0   2                                           Type in <LF>.
   TABLE1+1/   0   3                                         Type in <LF>.
   TABLE1+2/   0   $                                         Type in <ESC><RET>.
   START+1/   MOVEI IDX,TABLE1                               Type in <LF>.
   START+2/   PUSHJ P,ADDEM   .$b   $g
   $1B>>START+2/   PUSHJ P,ADDEM   $$x
   START+3/   MOVEI IDX,TABLE1   $x
      IDX/   TABLE1   TABLE1
   START+4/   MOVE 2(IDX)   $                                Type in <ESC><TAB>.
   TABLE1+2/   0   ^H                                        Type in <BKSP>.
   TABLE1+1/   3   ^H                                        Type in <BKSP>.
   TABLE1/   2   start$b   $g
   $2B>>START/   MOVE P,PWORD   $x
      P/   -10,,PWORD   PWORD/   -10,,PWORD
   START+1/   MOVEI IDX,TABLE1   $x
      IDX/   TABLE1   TABLE1
   START+2/   PUSHJ P,ADDEM   $x
      P/   -7,,STACK
   <JUMP>
   ADDEM/   MOVE 0(IDX)   $x
      0/   2   TABLE1/   2
   ADDEM+1/   ADD 1(IDX)   $x
      0/   5   TABLE1+1/   3
   ADDEM+2/   MOVE 2(IDX)   $x
      0/   0   TABLE1+2/   0
   ADDEM+3/   POPJ P,0   ^H                                  Type in <BKSP>.
   ADDEM+2/   MOVE 2(IDX)   movem r0,answer(idx)             Type in <RET>.
   ./   MOVEM 2(IDX)   .$b   $g
   $2B>>START/   MOVE P,PWORD   $p
   $1B>>START+2/   PUSHJ P,ADDEM   $p
   $3B>>ADDEM+2/   MOVEM 2(IDX)   $x
      0/   5   TABLE1+2/   5
   ADDEM+3/   POPJ P,0   start+4$b   $p
   $4B>>START+4/   MOVE 2(IDX)   $x
      0/   5   TABLE1+2/   5
   START+5/   JFCL 0   ^Z
   @
```

Figure 2-3: Terminal Display of Debugging Session

2.5 PROGRAMMING WITH DDT IN MIND

There are a few MACRO-20 programming techniques that make debugging with DDT easier. These techniques primarily concern the use of labels and symbols.

Labels that meaningfully describe (perhaps mnemonically, such as GETCHR for "get character") the function of the code are more helpful when examining code and setting breakpoints than labels that are alphanumerically coded (such as A0001).

When using symbols as offsets into tables, you can prevent DDT from displaying the offset symbol in place of the symbol's numeric value if you define the symbol in this way:

```
symbol==expression
```

The symbol table still contains symbol, and you can use symbol as input to DDT, but DDT does not display symbol on output.

For example, if you have defined

```
OFFSET==3
```

DDT displays the contents of a word that contains the value of 3 as

```
addr/   3
```

rather than

```
addr/   OFFSET
```

where addr is the address of the word. See the MACRO Assembler Reference Manual for more information about defining symbols.

CHAPTER 3

DDT COMMAND FORMAT

3.1 COMMAND SYNTAX

The complete syntax of a DDT command is

```
{arg1<}{arg2>}{arg3}{<ESC>{<ESC>}{arg4}}c{arg5}
```

where arg1, arg2, arg3, arg4, and arg5 are arguments to the command c; arg1, arg2, and arg3 can be any legal DDT expression; arg1 must be followed by a left angle bracket (<), and arg2 must be followed by a right angle bracket (>); arg4 can only be a number; arg5 is a text argument of the form

```
/text/     or     c<ESC>
```

where text is a string of characters (not beginning with <ESC>), the slashes (/) are delimiters that can be any character not contained in text, and c is a single character.

DDT commands never use all five arguments. Each argument is optional or required according to the syntax of the specific command. Most DDT commands are as simple as

```
arg3<ESC>c     or     arg3<ESC>arg4c
```

You can type in alphabetic commands and text arguments in uppercase or lowercase.

An argument to a command can be the result of executing another command. For example, you can type in a command to evaluate a text string, and then type in another command to deposit in memory the result of the evaluation. The entire command line would be

```
"/abcd/<RET>
```

where /abcd/ is the argument to the command " (quotation mark). The function of the quotation mark command is to evaluate the string (abcd) within the delimiters (/) as a left-justified ASCII string. The left-justified ASCII string abcd is then the argument to the command <RET> (entered by pressing the RETURN key). The function of the <RET> command is to deposit an argument (in this case, the string abcd) into the open location. The " command is described in this chapter, and the <RET> command is described in Chapter 4 (Displaying and Modifying Memory).

Most commands produce results that are immediately visible; for example, commands that display the contents of memory locations. However, commands such as those that invoke search functions or execute a range of instructions may not produce immediately visible results. If you type in a question mark (?) while DDT is performing a function invoked by one of these commands, DDT displays a message that tells you what it is currently doing. For example, such a message might be

```
Searching: ADDR/   VALUE
```

where ADDR is the address that DDT is to next test as part of a search, and VALUE is the contents of the memory location at ADDR. Still other commands return values that DDT does not display, but can use as arguments to other commands.

3.2 INPUT TO DDT

You type in arguments to DDT as expressions. An expression can be a single value, or a combination of one or more values with one or more operators.

3.2.1 Values in DDT Expressions

Values in DDT expressions can be:

- octal or decimal integers
- floating point numbers
- symbols
- values that are returned by commands
- text

To type in an octal integer value, simply type in the integer in octal digits. For example:

```
70707065
```

To type in a decimal integer value, type in the integer in decimal digits and follow the value with a decimal point. For example:

```
9876.
```

To type in a floating point number, use regular or scientific notation. For example, you can type in the value .034 as one of the following:

```
.034
3.4E-2
```

Note that 1. is a decimal integer, while 1.0 is a floating point number.

To type in a symbol as a value in an expression, type in the symbol name as defined in your program. To type in an undefined symbol that you can define later, type in

symbol#

where symbol is the symbol that you will later define. See Chapter 7 (Manipulating Symbols in DDT) for more information about using undefined symbols.

You can type in a command that returns a value as a value in an expression. DDT commands that return values, and the values they return are listed in Table 3-1.

Table 3-1: Commands That Return Values

Command	Value Returned	Value Also Known As
.	The address of the current location.	.
<ESC>.	The address of the next user program instruction to be executed.	$.
<ESC><ESC>.	The previous value of "<ESC>.".	$$.
<ESC>nB	The address of the DDT location that contains the address of breakpoint n.	$nB
<ESC>nI	The address of the DDT location that contains the saved machine state flags (user-program context).	
<ESC>nM	The address of DDT "mask" n.	
<ESC>Q	The current quantity.	$Q
<ESC><ESC>Q	The current quantity, with halves swapped.	$$Q
<ESC>nU	The address of the DDT location that contains the argument (or default) that was given in the virtual addressing command: expr<ESC>nU.	

The commands <ESC>nB, <ESC>nI, <ESC>nM, and <ESC>nU (<ESC>nU is legal only in EDDT, KDDT and FILDDT), return values that are the addresses of locations internal to DDT, which contain information that you can use and modify. For brevity, these commands are said to address those internal DDT locations.

For example, the command <ESC>nB returns (but does not display) the address of the DDT location that contains the address of breakpoint n, and the command addr/ (address followed by slash) displays the contents of the location at addr. To display the address of breakpoint n, type in

```
<ESC>nB/
```

where you type in the command <ESC>nB as the expression for DDT to evaluate as addr.

You can type in text to be interpreted in the following ways:

- left-justified text strings of byte-size n, where 6<=n<=36
- left-justified 7-bit ASCII strings
- left-justified SIXBIT strings
- single right-justified text characters of byte-size n, where 6<=n<=36
- single right-justified 7-bit ASCII characters
- single right-justified SIXBIT characters
- Radix-50 words

You can type in text expressions in uppercase or lowercase. DDT translates strings to uppercase for SIXBIT or Radix-50 text as required.

The term long text string refers to an expression in a DDT command that is a string of text characters that requires more than one 36-bit expression for full evaluation. You can type in long text strings in SIXBIT and ASCII as DDT expressions. If you use a long text string as an expression, DDT assumes that you will type in a command that deposits the expression in memory.

DDT evaluates the string one 36-bit expression at a time. After evaluating the first 36-bit expression, DDT deposits the expression in the open location, closes the open location, and opens the next location.

DDT then evaluates the next 36-bit expression contained in the string, and deposits that expression in the (new) open location. This process continues until you type in c, the command. If you type in a command that does deposit to memory, DDT deposits the final 36-bit expression in the open location, and updates the location counter according to the rules of that particular command. The current quantity is the last 36-bit expression that DDT evaluated.

If you do not type in a command that deposits to memory, DDT uses, as the argument to the command, the 36-bit expression that was last evaluated. All other 36-bit expressions that were evaluated as part of the string have been deposited, and the current and open locations were updated accordingly. The current quantity is then the last 36-bit expression that DDT evaluated.

If there is no open location when you begin typing the long text string, DDT evaluates only the first 36-bit expression, ignores the rest of the string, and uses the first 36-bit expression as the argument to the command. The current quantity is then the first 36-bit expression that DDT evaluated in the string. If you type in a command that deposits to memory, it has no effect because there was no open location.

The syntax to type in a left-justified text string with bytes of size n is

```
<ESC>n"/text/
```

where 6<=n<=36 (default = 6), text is the string, and the slashes (/) represent any printing character that is not contained within text. DDT evaluates the string as a series of 36-bit expressions, in n-bit ASCII format with all unused bits zero. Each character is right-justified within the byte; each string of bytes is left-justified within the 36-bit expression. No expression contains a partial byte. If n = 6, DDT evaluates the text string as for the command to type in a SIXBIT string (<ESC>"/text/), below. If n = 7, DDT evaluates the text string as for the command to type in a 7-bit ASCII string ("/text/), below.

For example, if you type in

```
<ESC>8"+abc/def+
```

DDT evaluates one 36-bit expression as the 8-bit ASCII string abc/ in bits 0-31, and bits 32-35 zero. If there is no open location, DDT uses that expression as the argument to the command, and that expression becomes the current quantity.

If there is an open location, DDT deposits abc/ in the open location, closes it, and opens the next location in memory. DDT then evaluates a second 36-bit expression as the 8-bit ASCII string def in bits 0-23, and bits 24-35 zero. The last 36-bit expression evaluated becomes the current quantity.

NOTE

You cannot use this format to type in a string that begins with the ESCAPE character, because <ESC> terminates the command that types in a single right-justified n-bit character (in this case, your intended delimiter).

For compatibility with previous versions of DDT, you can type in a 7-bit ASCII string with the command

"/text/

where text is the string, and the slashes (/) represent any printing character that is not contained within text. DDT evaluates the string as a series of 36-bit expressions, each in 7-bit ASCII format (left-justified), with all unused bits zero.

For example, if you type in

"+abc/def+

DDT evaluates one 36-bit expression as the 7-bit ASCII string abc/d in bits 0-34, and bit 35 zero. If there is no open location, DDT uses that expression as the argument to the command, and that expression becomes the current quantity.

If there is an open location, DDT deposits abc/d in the open location, closes it, and opens the next location in memory. DDT then evaluates a second 36-bit expression as the 7-bit ASCII string ef in bits 0-13, and bits 14-35 zero. The last 36-bit expression evaluated becomes the current quantity.

NOTE

You cannot use this format to type in an ASCII string that begins with the ESCAPE character, because <ESC> terminates the command that types in a single right-justified ASCII character (in this case, your intended delimiter).

For compatibility with previous versions of DDT, you can type in a SIXBIT string with the command

<ESC>"/text/

where text is the string, and the slashes (/) represent any printing character that is not contained within text. DDT evaluates the string as a series of 36-bit expressions, each in SIXBIT format (left-justified), with any unused bits in the last 36-bit expression cleared (zero). DDT translates lowercase characters to uppercase; all other non-SIXBIT characters cause DDT to sound your terminal buzzer or bell and display a question mark.

For example, if you type in

<ESC>">qwertyu>

DDT evaluates one 36-bit expression as the SIXBIT string QWERTY in bits 0-35. If there is no open location, DDT uses that expression as the argument to the command, and that expression becomes the current quantity.

If there is an open location, DDT deposits QWERTY in the open location, closes it, and opens the next location in memory. DDT then evaluates a second 36-bit expression as the SIXBIT character U in bits 0-5, with bits 6-35 zero. The last 36-bit expression evaluated becomes the current quantity.

The syntax to type in a right-justified character of byte size n is

<ESC>n"c<ESC>

where 6<=n<=36 (default = 6), and c is the character.

If n = 6, this command functions as for the command to type in a right-justified SIXBIT character (<ESC>"c<ESC>), otherwise this command functions as for the command to type in a right-justified 7-bit ASCII character ("c<ESC>). These commands are described below.

For compatibility with previous versions of DDT, you can type in a right-justified 7-bit ASCII character with the command

"c<ESC>

where c is the character. DDT evaluates this as one 36-bit expression with the 7-bit ASCII character c in bits 29-35, and bits 0-28 zero.

For compatibility with previous versions of DDT, you can type in a right-justified SIXBIT character with the command

<ESC>"c<ESC>

where c is the character. DDT evaluates one 36-bit expression with the SIXBIT character c in bits 30-35, and bits 0-29 zero. DDT translates lowercase characters to uppercase; all other non-SIXBIT characters cause DDT to sound your terminal buzzer or bell and display a question mark.

The syntax to type in a Radix-50 word is

text<ESC>5"

where text is any string of Radix-50 characters up to six characters long. DDT evaluates one 36-bit expression with bits 0-3 cleared (zero) and the Radix-50 string text in bits 4-35. DDT ignores any characters in text after the sixth.

For example, if you type in

poiuytr<ESC>5"

DDT evaluates one 36-bit expression with bits 0-3 cleared (zero) and the Radix-50 string POIUYT in bits 4-35. DDT ignores the character r. DDT translates lowercase characters to uppercase. If a character in text is not in the Radix-50 character set but is a DDT command, DDT tries to execute the command. DDT uses, as an argument to the command, the characters in text that precede or follow it, as appropriate to the command. If DDT cannot execute the command, it will sound your terminal buzzer or bell, and display the appropriate error message. If DDT can execute the command, it is possible that a Radix-50 evaluation of some remaining characters can take place, but the results will not be what you intend.

Characters in text not in the Radix-50 character set that are not DDT commands cause DDT to sound your terminal buzzer or bell and display a question mark.

3.2.2 Operators in DDT Expressions

When you type in an expression, DDT evaluates the expression to create a 36-bit quantity but does not necessarily use all 36 bits when it executes the command. For example, you can type in a complete MACRO instruction when giving an argument to a command that requires an address, but DDT uses only the address specified by the instruction (and ignores the rest of the evaluated expression) when it executes the command.

Table 3-2 lists DDT's expression operators and the effects those operators produce on the evaluation. The term value so far represents the accumulated 36-bit value that results from evaluation of the expression to that point.

The nonarithmetic operators allow you to type in expressions in instruction format as well as in data format.

Table 3-2: Effects of Operators When Evaluating Expressions

Operator	Effect on Evaluation
+	Adds the 36-bit value on the left to the 36-bit value on the right, using two's complement addition.
-	Subtracts the 36-bit value on the right from the 36-bit value on the left, using two's complement subtraction.
*	Multiplies the 36-bit value on the left by the 36-bit value on the right, using PDP-10 full-word integer multiplication. DDT uses only the low-order 36 bits of the result.
' (apostrophe)	Divides the 36-bit value on the left by the 36-bit value on the right, using PDP-10 full-word integer division. DDT ignores any remainder. NOTE Apostrophe is DDT's division operator. / (slash) is a DDT command to examine memory, and is never used in DDT to indicate division.
space	Adds the previous expression (normally an opcode) to the value so far, and adds the low-order 18 bits of the value at the right of the space to the low-order 18 bits of the value so far. DDT ignores carries resulting from the addition, and does not change the left half of the value so far.

Table 3-2 (Cont.): Effects of Operators When Evaluating Expressions

Operator	Effect on Evaluation
, (comma)	If you are entering an I/O instruction, DDT shifts the low-order 18 bits of the expression at the left of the comma 26 bits to the left (to the device field of the instruction), otherwise DDT shifts the low-order 18 bits of the expression at the left of the comma 23 bits to the left (to the A field of an instruction). DDT then logically ORs the result into the value so far. NOTE DDT does not check whether the value at the left of the comma is a legitimate device or AC address, and may overwrite other parts of the instruction.
,, (two commas)	Moves the low-order bits of the expression at the left of the commas to bits 0-17, and builds a new 18-bit expression in bits 18-35.
()	Swaps the halves of the expression within the parentheses and adds the resulting expression to the value so far. This makes it possible to enter an instruction that uses an index register. NOTE DDT does not check whether the value within the parentheses is a legitimate AC address, and may overwrite other parts of the instruction.
@	Sets the indirect bit (bit 13) of the value so far. This lets you type in instructions.

To type in an instruction, format the instruction as you would in a MACRO-20 program. For example:

```
MOVE R4,@VAR1+OFFSET(R5)
```

NOTE

Follow an opcode (such as MOVE) with a space, not a <TAB>.

To enter halfwords, type in the values (numbers or symbols) separated by two commas (,,). The halfwords can be symbolic or absolute values. For example:

```
-1,,SYM1
```

NOTE

DDT is not designed to evaluate complicated arithmetic expressions. The nonarithmetic operators are implemented to enable DDT to evaluate expressions you type in as MACRO-20 instructions and halfwords. Using values and operators for other purposes may not produce the results you intend.

CHAPTER 4

DISPLAYING AND MODIFYING MEMORY

4.1 DISPLAY MODES

A major function of DDT is displaying the contents of memory words, both data and instructions. You can choose whether to display the contents of memory words as symbols or as numeric values. You can also select the radix in which DDT displays numeric values.

DDT displays symbols, labels, and most messages in uppercase.

4.1.1 Default Display Modes

There is no sure way for DDT to distinguish between instruction and data words, or between data words of different formats.

DDT displays memory words in symbolic mode by default. Symbolic mode is described in Table 4-1. DDT tests for the condition on the left, and if the condition is met, displays the word in the format described on the right. DDT performs the tests in descending order (as they appear in the table).

Table 4-1: Evaluation of Symbolic Display Mode

Condition	DDT Displays	Example
Bits 0-18 are all set.	A negative number in the current radix.	-45
The 36-bit value is defined in the user program symbol table.	The symbol.	SYMBL1 HALT
The opcode field is zero.	Halfwords.	345,,-27
The opcode and I, X, and Y fields, or the opcode and A fields match an OPDEF in the user program symbol table.	The OPDEF.	CORE 6,
The opcode matches a definition in DDT's internal hardware instruction table.	The instruction.	MOVE 3,SYMBL
No match.	Halfwords.	3445,,-23

By default, DDT displays numeric values in octal. Leading zeros are always suppressed.

4.1.2 Selecting Display Modes

You can select display modes to control:

- the format in which DDT tries to interpret the contents of memory locations; for example, as instructions, or as floating-point numbers.
- whether addresses are displayed as symbolic or numeric values.
- the radix in which numeric values are displayed.

In addition, you can specify these modes on a short-term (temporary mode) or long-term (prevailing mode) basis.

A prevailing display mode remains in effect until you select another prevailing mode, but may be overridden by a temporary mode until you type in a command that restores the prevailing display mode. DDT commands that restore the prevailing display mode are:

- {expr}<RET> (deposit expr and close location)
- <ESC>G (start program execution)
- <ESC>P (proceed from a breakpoint)
- <ESC>W, <ESC>E, <ESC>N (perform a search)
- <ESC>Z (zero memory)
- instr<ESC>X (execute instr)
- <ESC>V (watch a location)

The syntax of commands that set the prevailing mode is

```
<ESC><ESC>mode
```

where mode is one of the display modes shown in Table 4-2.

The syntax of commands that set a temporary mode is

```
<ESC>mode
```

where mode is one of the display modes shown in Table 4-2.

The current display mode is the mode (prevailing or temporary) in which DDT will display the next word (unless you type in a command to change the display mode).

DDT has two "masks" that control the action of two of the display modes.

<ESC>3M is a command that addresses a DDT location that contains the output byte size mask. When the current display mode is O, each bit that is set in the mask indicates the position of a low order bit of a byte in the word being displayed. In this mode, bit 35 is always assumed to be set. For example, if the output byte size mask contains

    510410100400 (octal)

the byte sizes specified are, from left to right, 1, 2, 3, 4, 5, 6, 7, and 8. When displaying a word in O mode that contains 777777,,777777, and the current radix is 8, DDT displays

    1,3,7,17,37,77,177,377 The default value of the output byte size mask is zero, specifying one 36-bit byte.

You can set the output byte size mask with the command

    expr<ESC>3M

where expr evaluates to the bit pattern required.

You can also examine and change the output byte size mask with the examine and deposit commands described later in this chapter. This manual uses the symbol $3M to refer to the mask addressed by the command <ESC>3M.

<ESC>2M is a command that addresses a DDT location that contains the maximum symbolic offset. When DDT displays an address in R(elative) mode, it displays the address symbolically, that is, as a symbol, or as a symbol + the numeric offset of the address from that symbol. The maximum symbolic offset (minus 1) determines the maximum offset address that DDT displays symbolically, and defaults to 1000 (octal). DDT displays addresses beyond that offset in A(bsolute) mode. For example, assume that the maximum symbolic offset is 2, and that you are examining subroutine ADDEM in program X.MAC (refer to Figure 2-1), using <LF> to display instructions in sequence. DDT displays

```
ADDEM/   MOVE 0(6)
ADDEM+1/   ADD 1(6)
addr/   MOVE 2(6)
```

where addr is the absolute address (for example, 14414) of the location.

You can set the maximum symbolic offset with the command

    expr<ESC>2M

where expr evaluates to the offset required. This manual uses the symbol $2M to refer to the mask addressed by the command <ESC>2M.

You can also examine and change the maximum symbolic offset with the examine and deposit commands described later in this chapter.

DDT display modes and the commands that select them are described in Table 4-2.

Table 4-2: DDT Display Modes

Format Modes	
Mode	**Effect**
C	Displays memory word as numbers in the current radix (see Radix Modes).
F	Displays memory word as a floating point decimal number.
2F	Displays two contiguous memory words as a double precision floating point decimal number.
H	Displays memory word as two halfword addresses (see Address Modes) separated by two commas (,,).
O	Displays memory word as numeric bytes of sizes that are specified by the $3M mask.
nO	Displays memory word as n-bit numeric bytes (left-justified, with trailing remainder byte, as required).
S	Displays memory word in symbolic mode (default).
1S	Searches DDT's internal hardware opcode table before searching the user's symbol table, otherwise follows rules for S (symbolic) mode.
1T	Displays memory word(s) as a byte pointer (one-word local, one-word global, or two-word byte pointer, as specified by the byte pointer format).
nT	Displays memory word as ASCII text, using n-bit bytes (5<=n<=36). n=5: RADIX50 n=6: SIXBIT n=7: 7-bit ASCII (1) 8<=n<=36: n-bit ASCII (2)

Table 4-2 (Cont.): DDT Display Modes

Address Modes	
Mode	Effect
A	Displays addresses as absolute values in the current radix.
R	Displays addresses as values relative to symbols (default). DDT displays the offsets in the current radix. The maximum offset is controlled by the value stored in the $2M mask, and defaults to 1000 (octal).
Radix Modes	
Mode	Effect
nR	Displays numeric values in radix n (default=8), where n is a decimal number greater than 1. If n=8, DDT displays the word as octal halfwords, otherwise DDT displays the word as one number. If n=10 (decimal), DDT displays a decimal point after the value.

(1) If bits 0-28 are all zero, DDT assumes the word contains a single right-justified 7-bit ASCII character. Otherwise, DDT displays the word as a left-justified 7-bit ASCII string, and if bit 35 is set, DDT displays @ immediately following the five 7-bit ASCII characters.

(2) DDT ignores all bits in each n-bit byte other than the rightmost 7 bits, and ignores any leftover bits at the right end of the word. To display a right-justified 7-bit ASCII character in a word that has one or more of bits 0-28 set, type in <ESC>36T.

4.2 DISPLAYING EXPRESSIONS

DDT has three commands you can use to display expressions in different modes. They are:

; (semicolon)

= (equal sign)

_ (underscore)

The syntax of these commands is

{expr}c

where expr is the expression to be displayed (expr defaults to the current quantity), and c is one of the above commands. These commands are useful for redisplaying the current quantity without affecting the current display mode. Table 4-3 lists the commands to display expressions and their effects.

Table 4-3: Commands to Display Expressions

Command	Effect
;	Displays the current quantity in the current display mode.
expr;	Displays expr in the current display mode.
=	Displays the current quantity as a number in the current radix.
expr=	Displays expr as a number in the current radix.
_	Displays the current quantity in 1S mode.
expr_	Displays expr in 1S mode.

4.3 DISPLAYING BYTE POINTERS

If you set the display mode to 1T, DDT displays the contents of the memory location as a byte pointer. DDT can display one-word local, one-word global, and two-word byte pointers. DDT displays the P and S fields, and the address as determined by the I, X, and Y fields of the byte pointer.

In section zero, DDT displays only one-word byte pointers (local and global).

For example, if the contents of the location at ADDR2 is 100702,,addr, where addr is the value of symbol LABL2, the following illustrates one-word local byte pointer display:

```
addr2/   100702,,addr   <ESC>1t;   10 7 LABL2(2)
```

The following illustrates one-word global byte pointer display, where addr is the value of symbol LABL2:

```
1,,addr2/   610002,,LABL2   <ESC>1t;   44&7 2,,LABL2
```

The following illustrates two-word global byte pointer display, where addr is the value of symbol LABL2 (DDT echoes <BKSP> as ^H):

```
1,,addr2/   440740,,0   <LF>
1,,addr2+1/   3,,addr   <ESC>1t^H
1,,addr2/   44 7 3,,MAIN. <2>
```

4.4 DISPLAYING AND DEPOSITING IN MEMORY

DDT allows you to display the contents of memory locations and deposit a new value in the open location. In performing these functions, you must understand the concept of the open location, the current location, the location sequence stack, and the current quantity.

The open location is a memory location (or AC) that can be modified by the next command. There is never more than one location open at a time. DDT always closes the open location before opening another.

The location counter contains the address of a word in memory that has been referenced (implicitly or explicitly) by the previous command, and that is the default point of reference for the next command. That word is known as the current location. DDT uses the address of the current location as the default address in most commands. The current location is often, but not always, the open location.

Most DDT commands change the current location to a word specified by an address given (explicitly or by default) in the command. Commands that do not are so indicated.

"." (period) is a command that returns (but does not display) the address of the current location.

When you first enter DDT, the current location is zero.

The location sequence stack is a "ring" of seventeen words, each containing the address of a prior current location, or of a match found during a search. The present value of the current location is not placed in the ring.

Entries are made to and retrieved from the location sequence stack in a last-in, first-out manner. Most commands that change the location counter by values other than +1 and -1 cause DDT to store the address of the current location (before the change) on the location sequence stack. Addresses of matching locations found during searches are also stored on the location sequence stack.

When DDT stores a new value in the next word on the stack, the new value becomes the current location stack entry. This is similar to PUSHing entries on a stack.

When the current location stack entry is the last location on the location sequence stack, DDT stores a new value on the stack by "wrapping around" to the beginning of the stack and overwriting the value in the first location on the stack. The first location on the stack then contains the current location stack entry.

Certain DDT commands change the address of the current location to the current location stack entry, and then change the current location stack entry to the previous entry. This is similar to POPping entries off a stack, and allows you to "return" to locations that have previously been the current location.

When the first location on the location sequence stack contains the current location stack entry and DDT changes the address of the current location to the current location stack entry, DDT "wraps around" to the end of the stack, and the value contained in the last word of the stack becomes the current location stack entry (whether or not the stack was previously "full").

The current quantity is a value that is the more recent of:

- the last 36-bit quantity that DDT displayed (an expression or the contents of a memory location)
- the last expression that you typed in as an argument to a command that deposits to memory

This value is also known as the last value typed. <ESC>Q is a command that returns (but does not display) the current quantity. DDT issues an implicit <ESC>Q to return this value for use as the default argument for some commands.

You can give the current quantity as an argument to a command by typing in the command <ESC>Q as the argument.

The command <ESC><ESC>Q returns the current quantity with the right and left halves swapped.

This manual uses the term $Q to refer to the value that is returned by the command <ESC>Q, and the term $$Q to refer to the value that is returned by the command <ESC><ESC>Q.

Some commands calculate the address of the location to be opened from an expression given or defaulted in the command. Other commands use the address of the current location or entries on the location sequence stack.

The general syntax of these commands is

{expr}{<ESC>}c

where expr is any legal DDT expression, and c is the command. See Section 3.2.1, Values in DDT Expressions, for a discussion of long text strings as values in DDT expressions.

Table 4-4 summarizes the commands and their effects. Complete descriptions of the commands follow the table.

Table 4-4: DDT Commands to Display Memory

Command	Display Content	Display Mode	Open Word	Change "."	Deposit Expr	Loc/Seq Stack
/	Yes	Current	Yes	Yes(1)	No	PUSH(1)
[	Yes	Numeric	Yes	Yes(1)	No	PUSH(1)
]	Yes	Symbolic	Yes	Yes(1)	No	PUSH(1)
!	No	Suppress	Yes	Yes(1)	No	PUSH(1)
\	Yes(2)	Current	Yes	No	Yes(1)	No
<TAB>	Yes(2)	Current	Yes	Yes	Yes(1)	PUSH
<RET>	No	Restore	No	No	Yes(1)	No
<LF>	Yes(2)	Current	Yes	Yes .+1	Yes(1)	No
<BKSP> or ^	Yes(2)	Current	Yes	Yes .-1	Yes(1)	No
<ESC> <TAB>	Yes	Current	Yes	Yes	Yes(1)	POP
<ESC> <RET>	Yes	Restore Current	No	Yes	Yes(1)	POP
<ESC> <LF>	Yes	Current	Yes	Yes Old.+1	Yes(1)	POP
<ESC> <BKSP> or <ESC>^	Yes	Current	Yes	Yes Old.-1	Yes(1)	POP

(1) If you type in expr.

(2) If not suppressed by !.

4.4.1 Commands That Use the Current Location

The commands <RET>, <LF>, and <BKSP> use the address of the current location to determine the next address of the current location. These commands do not make entries to the location sequence stack.

These commands are described in detail on the next pages.

{expr}<RET>

- deposits expr (if given) in the open location
- closes the open location
- starts a new display line
- resets the current typeout mode to the prevailing typeout mode
- does not change the address of the current location

{expr}<LF>

- deposits expr (if given) in the open location
- closes the open location
- increments the location counter
- opens the current location
- starts a new line and displays the address of the open location, followed by:
  - [(left square bracket), if the current display mode is C (numeric), and the prevailing mode is not
  -] (right square bracket), if the current display mode is S (symbolic), and the prevailing display mode is not
  - ! (exclamation point), if display has been suppressed by !
  - / (slash), in all other cases
- the contents of the open location (unless display has been suppressed by !)

{expr}<BKSP> and {expr}^

- deposit expr (if given) in the open location
- close the open location
- decrement the location counter
- open the current location
- start a new line and display the address of the open location, followed by:
  - [(left square bracket), if the current display mode is C (numeric), and the prevailing mode is not
  -] (right square bracket), if the current display mode is S (symbolic), and the prevailing display mode is not
  - ! (exclamation point), if display has been suppressed by !
  - / (slash), in all other cases
- the contents of the open location (unless display has been suppressed by !)

4.4.2 Commands That Use the Location Sequence Stack

The commands <ESC><RET>, <ESC><LF>, and <ESC><BKSP> use the current location stack entry to determine the next address of the current location.

Repetitions of these commands refer to successively earlier entries on the stack, until you again address the most recent entry.

These commands do not make entries to the location sequence stack.

These commands are described in detail on the following pages.

{expr}<ESC><RET>

- deposits expr (if given) in the open location
- closes the open location
- changes the value contained in the location counter to the current location stack entry
- opens the current location
- starts a new line and displays the address and contents of the open location in the current display mode
- causes the previous entry on the location sequence stack to become the current location stack entry

NOTE

If display is suppressed as a result of using the ! command, the command {expr}<ESC><RET> restores the current display mode, which can be either a temporary or prevailing display mode.

{expr}<ESC><LF>

- deposits expr (if given) in the open location
- closes the open location
- changes the value contained in the location counter to the current location stack entry
- increments the location counter
- opens the current location
- starts a new line and displays the address of the open location, followed by:
  - [(left square bracket), if the current display mode is C (numeric), and the prevailing mode is not
  -] (right square bracket), if the current display mode is S (symbolic), and the prevailing display mode is not
  - ! (exclamation point), if display has been suppressed by !
  - / (slash), in all other cases
- displays the contents of the open location (unless display has been suppressed by !)
- causes the previous entry on the location sequence stack to become the current location stack entry

{expr}<ESC><BKSP> and {expr}<ESC>^

- deposit expr (if given) in the open location
- close the open location
- change the value contained in the location counter to the current location stack entry
- decrement the location counter
- open the current location
- start a new line and display the address of the open location, followed by:
  - [(left square bracket), if the current display mode is C (numeric), and the prevailing mode is not
  -] (right square bracket), if the current display mode is S (symbolic), and the prevailing display mode is not
  - ! (exclamation point), if display has been suppressed by !
  - / (slash), in all other cases
- display the contents of the open location (unless display has been suppressed by !)
- cause the previous entry on the location sequence stack to become the current location stack entry

{expr}<ESC><LF>

- deposits expr (if given) in the open location
- closes the open location
- changes the value contained in the location counter to the current location stack entry
- increments the location counter
- opens the current location
- starts a new line and displays the address of the open location, followed by:
  - [(left square bracket), if the current display mode is C (numeric), and the prevailing mode is not
  -] (right square bracket), if the current display mode is S (symbolic), and the prevailing display mode is not
  - ! (exclamation point), if display has been suppressed by !
  - / (slash), in all other cases
- displays the contents of the open location (unless display has been suppressed by !)
- causes the previous entry on the location sequence stack to become the current location stack entry

{expr}<ESC><BKSP> and {expr}<ESC>^

- deposit expr (if given) in the open location
- close the open location
- change the value contained in the location counter to the current location stack entry
- decrement the location counter
- open the current location
- start a new line and display the address of the open location, followed by:
  - [(left square bracket), if the current display mode is C (numeric), and the prevailing mode is not
  -] (right square bracket), if the current display mode is S (symbolic), and the prevailing display mode is not
  - ! (exclamation point), if display has been suppressed by !
  - / (slash), in all other cases
- display the contents of the open location (unless display has been suppressed by !)
- cause the previous entry on the location sequence stack to become the current location stack entry

4.4.3 Commands That Use an Address Within the Command

The commands:

```
     /       (slash)
     [       (left square bracket)
     ]       (right square bracket)
     !       (exclamation point)
     \       (backslash)
  <TAB>
```

use an expression given in the command (either explicitly or by default) to determine the addresses of the current location and the open location.

The complete syntax of these commands is

```
{expr}{<ESC>{<ESC>}}c
```

where expr may be an address, ".", a symbol, or any expression that is legal in DDT, and c is the command.

When you use the commands /, [,], !, \, and <TAB>:

- If you omit expr
  - DDT uses the current quantity as a default.
  - <TAB> enters the address of the current location on the location sequence stack and changes the current location to the address determined from the current quantity.
- If you type in expr, DDT enters the address of the current location on the location sequence stack (except \).
- DDT treats expr (whether given or defaulted) as if it were in instruction format and performs the effective address calculation as follows:
  - If you omit <ESC>, DDT does not perform indexing or indirection.
  - If you include one <ESC>, DDT treats expr as an IFIW (instruction format indirect word), and uses the I and Y fields of expr to perform indexing and indirection when appropriate.
  - If you use <ESC><ESC> in a nonzero section, DDT utilizes EFIWs (extended format indirect words), as appropriate, when performing effective address calculations, and can thereby calculate 30-bit addresses.
  - If you use <ESC><ESC> in section zero, DDT treats the command as if you had typed in one <ESC>. See Chapter 12 (Extended Addressing), for a description of this form of these commands.

The commands /, [,], !, \, and <TAB> always do the following:

- close the open location
- open the location at the address indicated by expr
- change the current quantity to the value displayed (all commands except !)

The following pages describe the effects of each of these commands.

/

- closes the open location
- opens the location at the address calculated from the current quantity
- displays the contents of the open location in the current display mode
- sets the current quantity to the value displayed

expr/

- closes the open location
- opens the location at the address calculated from expr
- enters the address of the current location on the location sequence stack
- changes the current location to the location at the address calculated from expr
- displays the contents of the open location in the current display mode
- sets the current quantity to the value displayed

[

- closes the open location
- opens the location at the address calculated from the current quantity
- displays the contents of the open location in numeric mode in the current radix
- sets the current display mode to numeric mode in the current radix
- sets the current quantity to the value displayed

expr[

- closes the open location
- opens the location at the address calculated from expr
- enters the address of the current location on the location sequence stack
- changes the current location to the location at the address calculated from expr
- displays the contents of the open location in numeric mode in the current radix
- sets the current display mode to numeric mode in the current radix
- sets the current quantity to the value displayed

]

- closes the open location
- opens the location at the address calculated from the current quantity
- displays the contents of the open location in symbolic mode
- sets the current display mode to symbolic mode
- sets the current quantity to the value displayed

expr]

- closes the open location
- opens the location at the address calculated from expr
- enters the address of the current location on the location sequence stack
- changes the current location to the location at the address calculated from expr
- displays the contents of the open location in symbolic mode
- sets the current display mode to symbolic mode
- sets the current quantity to the value displayed

!

- closes the open location
- opens the location at the address calculated from the current quantity
- does not display the contents of the open location
- suppresses display of the open location by the \, <TAB>, <LF>, and <BKSP> commands (any other display command restores the current display mode)
- does not change the current quantity

expr!

- closes the open location
- opens the location at the address calculated from expr
- enters the address of the current location on the location sequence stack
- changes the current location to the location at the address calculated from expr
- does not display the contents of the open location
- suppresses display of the open location by the \, <TAB>, <LF>, and <BKSP> commands (any other display command restores the current display mode)
- does not change the current quantity

\

- closes the open location
- opens the location at the address calculated from the current quantity
- displays the contents of the open location in the current display mode (unless display has been suppressed by !)
- sets the current quantity to the value displayed

expr\

- deposits expr in the open location
- closes the open location
- opens the location at the address calculated from expr
- does not change the address of the current location (and does not enter the address of the current location on the location sequence stack)
- displays the contents of the open location in the current display mode (unless display has been suppressed by !)
- sets the current quantity to the value displayed

<TAB>

- closes the open location
- opens the location at the address calculated from the current quantity
- enters the address of the current location on the location sequence stack
- changes the current location to the location at the address calculated from the current quantity
- starts a new line and displays the address of the open location (which is also the current location)
- displays the contents of the open location in the current display mode (unless display has been suppressed by !)
- sets the current quantity to the value displayed

expr<TAB>

- deposits expr in the open location
- closes the open location
- opens the location at the address calculated from expr
- enters the address of the current location on the location sequence stack
- changes the current location to the location at the address calculated from expr
- starts a new line and displays the address of the open location (which is also the current location)
- displays the contents of the open location in the current display mode (unless display has been suppressed by !)
- sets the current quantity to the value displayed

You can treat expr as an IFIW (instruction format indirect word), and use any indexing and indirection specified by expr to compute the effective address of the location to be opened. Use the command form

{expr}<ESC>c

where c is /, [,], !, \, or <TAB>.

For example, assume the following conditions as indicated by the display commands:

Command	Display	Explanation
LABL1/	SYM1	Displays contents of LABL1.
LABL1+1/	SYM2	Displays contents of LABL1+1.
SYM2/	SYM3	Displays contents of SYM2.
2/	1	Displays contents of AC 2.
@LABL1(2)/	SYM1	DDT uses Y field only.
@LABL1(2)<ESC>/	SYM3	<ESC> causes indexing and indirection.

Note that DDT does not start a new line unless you type in <TAB>, <RET>, <LF> or <BKSP>, or until the display wraps around the end of the line. DDT also displays three spaces (or a tab, depending on the TTY control mask) before and after its output. Thus, an actual DDT terminal display might be the following (user input is lowercase; <LF> and <TAB> do not appear on the screen, but are shown to indicate where you pressed the corresponding keys):

```
2/   1   labl1/   SYM1   <LF>
LABL1+1/   SYM2   <TAB>
SYM2/   SYM3   sym4/   MOVE 1,@LABL1(2)   <ESC><TAB>
SYM2/   SYM3
```

You can treat expr as an EFIW (extended format indirect word) and use any indexing and indirection specified by expr to compute the (global) effective address of the location to be opened. Use the command form

{expr}<ESC><ESC>c

where c is /, [,], !, \, or <TAB>.

4.5 DISPLAYING ASCIZ STRINGS

You can display memory as an ASCIZ string. The command

addr<ESC>0T

where addr defaults to the open location (if there is one, otherwise addr defaults to the current location), displays memory, beginning with addr, as an ASCIZ string. The display stops when DDT finds a zero byte, or when you type in any character, which DDT displays, but otherwise ignores. The current location remains unchanged.

4.6 ZEROING MEMORY

To deposit the same value in each of a string of memory locations (useful for initializing memory to zero), type in

addr1<addr2>{expr}<ESC>Z

where expr is any legal DDT expression, addr1 is the first word to receive expr, and addr2 is the last. Follow addr1 with a left angle bracket (<) and addr2 with a right angle bracket (>). Both addr1 and addr2 are required. If you omit expr, it defaults to zero. Prior to execution, DDT enters the address of the current location on the location sequence stack and closes the open location. When DDT has completed execution of the command, the current location is the word at addr2. There is no open location. This command restores the prevailing display mode.

If you type in

?

while DDT is executing the <ESC>Z command, DDT displays

Depositing: addr/ value

where addr is the location where DDT will make the next deposit, and value is the contents of addr before the deposit.

If you type in any other character, DDT stops executing the <ESC>Z command, and waits for your next command. The current location is the last memory location that received a deposit. There is no open location. The character that you type in to terminate the <ESC>Z command is otherwise ignored.

4.7 AUTOMATIC WRITE-ENABLE

If you attempt to deposit a value in a location that is write-protected, DDT returns the message

```
?NOT WRITABLE
```

This is the TOPS-20 default condition.

To allow DDT to modify write-protected memory, type in

```
<ESC>{0}W
```

If you now attempt to deposit a value in a location that is write-protected, DDT removes the protection, deposits the value, and then reinvokes the protection.

Note that you cannot use this command to enable patching in FILDDT.

To prevent DDT from modifying write-protected memory, type in

```
<ESC><ESC>{0}W
```

The zero in the above commands is optional and has no effect on the operation of the commands. DDT allows the zero for compatibility with prior versions of DDT.

4.8 AUTOMATIC PAGE CREATION

If you attempt to deposit a value in a location within a nonexistent page, DDT creates the page and deposits the value. If you attempt to deposit a value within a nonexistent section, DDT creates the section as well as the page. This is the default condition.

To prevent DDT from creating a page when you attempt to deposit a value within a nonexistent page, type in

```
<ESC><ESC>1W
```

If you now attempt to deposit a value in a location within a nonexistent page, DDT returns the error message

```
CAN'T CREATE PAGE
```

To allow DDT to create the page (and the section, as required) when you attempt to deposit a value within a nonexistent page, type in

```
<ESC>1W
```

4.9 PAGE ACCESS

You can get information about the access requirements of pages and sections in the program you are debugging. This information is similar in form and content to the information produced when you use the TOPS-20 INFORMATION (ABOUT) MEMORY-USAGE command.

The command syntax is

```
{{arg1<}arg2}{<ESC>}<ESC>L
```

where arg1 and arg2 are section numbers. Using one <ESC> causes DDT to display access information about the section and about individual pages. Using <ESC> twice causes DDT to display access information only about the section(s). If you include both arg1 and arg2, DDT displays the information for all sections that your program and DDT are using, in the range arg1 to arg2, inclusive. If you include only arg2, DDT displays access information for that section only. If you omit both arg1 and arg2, DDT displays access information for all sections that your program and DDT are using.

For example, the command <ESC>L might produce the following display:

```
 Section 0   Read, Write, Execute, Private
000-012   Read, Copy-on-write, Execute
014-025   Read, Copy-on-write, Execute
770       Read, Execute
771       Read, Write, Execute, Private
 Section 37   Read, Write, Execute, Private
700-701   Read, Copy-on-write, Execute
703-727   Read, Copy-on-write, Execute
736-737   Read, Write, Execute, Private
740-753   Read, Execute
```

The command <ESC><ESC>L might produce the following display:

```
 Section 0   Read, Write, Execute, Private
 Section 37   Read, Write, Execute, Private
```

The text for each kind of access is:

Text	Explanation
Read	page is readable
Write	page is writable
Copy-on-write	page is copy-on-write
Execute	page is executable
Private	page is private
Zero	page is allocated but zero (FILDDT only)

See the SET PAGE-ACCESS command in the TOPS-20 Commands Reference Manual for more information about access to pages.

4.10 WATCHING A MEMORY LOCATION

If you wish to have DDT monitor or "watch" a memory location while your program is running, and display the location whenever its contents change, type in

addr<ESC>V

where addr is the address of the location to be watched, and defaults to the current location. When you type in the command, DDT starts a new line and displays

ADDR/ VALUE

where ADDR is the address of the location being watched, and VALUE is the contents of the location. DDT displays VALUE in the current display mode. This command restores the prevailing display mode when you terminate the watch.

DDT checks ADDR every "jiffy" (about 20 milliseconds), and displays the address and contents of ADDR whenever those contents change. (Executive mode EDDT and KDDT watch ADDR continuously.)

If you type in a question mark (?) while DDT is watching, DDT displays

Watching: ADDR/ VALUE

where ADDR is the address of the location being watched, and VALUE is the contents of ADDR.

To terminate the watch, type in any other character. DDT stops monitoring the location, starts a new display line, echoes the character you type in, starts another line, and waits for more input. The character that you type in to terminate the watch is otherwise ignored.

Because any input character terminates the watch, you cannot continue execution and watch your own user program. The <ESC>V command is useful to watch activity in a separate process (such as the running monitor or other job, for which you must be using EDDT, MDDT, or FILDDT). The page that contains the location you wish to watch must be mapped into your own process (the one that contains DDT and your program).

4.11 TTY CONTROL MASK

You can control certain aspects of DDT's display by setting DDT's TTY control mask. The command <ESC>1M returns a value that is the address of the DDT location that contains this mask. Table 4-5 summarizes the features controlled by the bits in the TTY control mask.

Table 4-5: TTY Control Mask

Bit	Value	Effect
16	0	When interrupting program execution at a breakpoint, DDT displays the address and contents of the breakpoint (default).
	1	When interrupting program execution at a breakpoint, DDT displays only the address of the breakpoint.
17	0	DDT displays 3 spaces when spacing output (1).
	1	DDT displays output fields at tab stops (1).
34	0	The terminal does not have a tab mechanism (2).
	1	The terminal has a tab mechanism (2).
35	0	DDT echoes deleted characters (3).
	1	DDT backspaces over deleted characters (3).

(1) If bit 17 is zero (default), DDT displays 3 spaces between output fields (such as between the address of a location and the contents of the location), and at the end of display lines. If bit 17 is set, DDT lines up the output fields in columns beginning at tab stops (see bit 34).

Figure 4-1 illustrates the two different modes.

(2) If bit 34 is set, DDT displays a tab character (<CTRL/I>) between fields. If bit 34 is zero, DDT displays enough spaces to start the field at the next tab stop. When starting up, DDT checks whether your terminal can handle TAB characters (<CTRL/I>), and sets this bit accordingly.

(3) When starting up, DDT checks whether your terminal can backspace to delete characters, and sets this bit accordingly.

To change the settings of the TTY control mask, use the command

```
expr<ESC>1M
```

where expr evaluates to the required bit pattern.

You can also open the location addressed by <ESC>1M with one of the DDT display commands, and deposit an expression that contains the new bit settings.

Figure 4-1 is an illustration of the effects of bit 17 in the TTY control mask. The code being examined is the first few lines of X.MAC, listed in Figure 2-1. The example is not a complete debugging session; only enough is shown to illustrate the effects of bit 17 of the TTY control mask. The numbers at the left of the DDT display lines are to assist you in following the commentary below. User input is in lowercase.

Line	Screen Display
1	DDT
2	start/ MOVE P,PWORD x$: .$b $g
3	$1B>>START/ MOVE P,PWORD $x
4	P/ -10,,STACK PWORD/ -10,,STACK
5	START+1/ MOVEI IDX,TABLE1 $x
6	IDX/ TABLE1 TABLE1 $1m/ 2 1,,2
7	start$g
8	$1B>>START/ MOVE P,PWORD $x
9	P/ -10,,STACK PWORD/ -10,,STACK
10	START+1/ MOVEI IDX,TABLE1 $x
11	IDX/ TABLE1 TABLE1

Commentary

Line 1:

DDT is loaded and waiting for a command.

Line 2:

Type in start/ to examine location start.

Type in x<ESC>: to open the symbol table for module X.

Type in .<ESC>b to set breakpoint at location START.

Type in <ESC>g to begin execution.

Line 3:

DDT displays breakpoint information.

Type in <ESC>x to execute the next instruction.

Line 4:

DDT displays results of executing the instruction.

Line 5:

DDT displays the next instruction.

Type in <ESC>x to execute the instruction.

Line 6:

DDT displays the results of executing the instruction.

Type in <ESC>1m/ to display and open the TTY control mask.

DDT displays the mask. Bit 34 is set.

Type in 1,,2<RET> to set bits 17 and 34.

Line 7:

Type in start<ESC>g to restart the program.

Line 8:

DDT displays the breakpoint information.

Type in <ESC>x to execute the instruction.

Line 9:

DDT displays the results of executing the instruction.

Line 10:

DDT displays the next instruction.

Type in <ESC>x to execute the next instruction.

Line 11:

DDT displays the results of executing the instruction.

Figure 4-1: DDT Session Showing Columnar Output with Commentary

CHAPTER 5

CONTROLLING PROGRAM EXECUTION

5.1 BEGINNING EXECUTION

To begin execution of your program, type in

```
<ESC>G
```

Your program will run, beginning at its start address. If you have not set any breakpoints, your program runs to completion, or until it makes a fatal error. You can then use the TOPS-20 DDT command to reenter DDT and examine your program.

You can start or continue program execution at any address with the command

```
addr<ESC>G
```

where addr is the address at which you want to begin program execution.

5.2 USING BREAKPOINTS

A breakpoint is a program location that has been altered such that if your program PC reaches the address of the breakpoint, your program transfers control to DDT.

When you set a breakpoint, DDT stores the address of the breakpoint in an internal table. When you command DDT to begin or continue program execution, DDT stores the instructions from all breakpoints in the table, and replaces them with JSRs into a DDT entry table.

While program execution is suspended at a breakpoint, you can examine and modify memory, remove breakpoints, insert new breakpoints, execute individual instructions, and continue program execution.

During this time, the command "<ESC>." returns the value that is the address of the next instruction to be executed. The command "<ESC><ESC>." returns a value that is the previous value returned by "<ESC>.".

When you first receive control at the breakpoint, "<ESC>." returns the address of the breakpoint and "<ESC><ESC>." returns zero. Before you start execution with <ESC>G, "<ESC>." and "<ESC><ESC>." are illegal commands. If you try to execute them, DDT sounds the terminal buzzer or bell and displays a question mark.

NOTE

This manual uses the term "$." to represent the value returned by the command "<ESC>.", and the term "$$." to represent the value returned by the command "<ESC><ESC>.".

You can set up to 12 breakpoints at a time (this is a DDT assembly parameter) in your program. These breakpoints are numbered 1 through 12. There is also one breakpoint (the unsolicited breakpoint, numbered zero) that can be used by your MACRO program to "call" DDT.

When your user-program PC reaches a breakpoint, your program executes the JSR into DDT. When this occurs, DDT does the following:

- saves your user-program context
- replaces the JSR instructions at all breakpoints with the original program instructions
- displays the breakpoint number, breakpoint address, and the contents of the breakpoint (depending on bit 16 of the TTY control mask)
- sets "$." to the breakpoint address
- sets "$$." to zero
- enters the address of the current location (set before you started the program or proceeded from a breakpoint) on the location sequence stack
- changes the current location to the breakpoint
- waits for you to give a DDT command

Each breakpoint has several internal DDT locations associated with it, which contain information to control DDT action with respect to the breakpoint. You can examine and modify these DDT locations with the same DDT commands that you use to examine and modify locations in your user program. <ESC>nB is a command that returns the value that is the address of the first DDT word associated with breakpoint n. The symbol $nB is used here to represent that address.

Table 5-1 contains a list of the breakpoint locations of interest to you, and their contents.

Table 5-1: Breakpoint Locations of Interest

Location	Contents
$nB	Address of breakpoint n.
$nB+1	Instruction for conditional breakpoint n.
$nB+2	Proceed count for conditional breakpoint n.
$nB+3	Address of a location to be opened and displayed when the breakpoint is reached.

When you command DDT to restart or continue program execution, DDT does the following:

- saves the program instructions from all breakpoints
- replaces the program instructions at all breakpoints with JSR instructions to DDT
- if you have not executed the instruction at the breakpoint with <ESC>X, simulates execution of the instruction at the breakpoint
- restores your user-program context
- performs a JRSTF (if in section zero, otherwise XJRSTF) to the next instruction to be executed

5.2.1 Setting Breakpoints

To set a breakpoint, type in

addr<ESC>{n}B

where addr is the address where you want to suspend execution and n is the number of the breakpoint. You can use ".", the command that returns the address of the current location, for addr. DDT defaults n to the lowest unused breakpoint number.

If you do not specify n, it defaults to the lowest available (unset) breakpoint. If you have already set twelve breakpoints, DDT displays "?" and sounds the terminal buzzer or bell.

If you specify n, it must be greater than zero and less than 13. DDT restores the original contents of any (previously set) breakpoint designated as breakpoint n before setting new breakpoint n.

You cannot set more than one breakpoint at the same address. DDT simply sets the same breakpoint again, even if you explicitly specify a breakpoint number the second time.

You cannot set a breakpoint at AC zero.

Assume the following conditions:

- location LABL1+3 contains the instruction MOVE 1,LABL2
- breakpoint 2 is set at LABL1+3

If your program reaches LABL1+3 it executes the JSR to DDT, and DDT does the following:

- saves your user-program context
- restores the original program instructions to the breakpoints
- sets "$." to LABL1+3
- sets "$$." to zero
- enters the address of the current location on the location sequence stack
- changes the current location to LABL1+3 (the breakpoint)
- opens location LABL1+3
- displays: $2B>>LABL1+3/ MOVE 1,LABL2

To set a breakpoint and have DDT display an additional location when your program reaches the breakpoint, type in

```
addr1<addr2<ESC>{n}B
```

where addr1 is the location to be displayed, and addr2 is the location of the breakpoint. Follow addr1 with a left angle bracket (<).

Assume the following conditions:

- location LABL1+3 contains the instruction MOVE 1,LABL2
- location LABL3 contains value SYMBL1
- breakpoint 2 was set by the command

```
LABL3<LABL1+3<ESC>B
```

If your program reaches LABL1+3 it executes the JSR to DDT, and DDT does the following:

- saves your user-program context
- restores the original program instructions to the breakpoints
- sets "$." to LABL1+3
- sets "$$." to zero
- enters the address of the current location on the location sequence stack
- changes the current location to LABL1+3 (the breakpoint)
- enters the address of the current location (the breakpoint) on the location sequence stack
- changes the current location to LABL3
- opens location LABL3
- displays: $2B>>LABL1+3/ MOVE 1,LABL2 LABL3/ SYMBL1

Note that, because DDT placed the breakpoint address on the location sequence stack, you can type in <ESC><RET> to change the current location back to the breakpoint.

To display the address of any breakpoint, type in

```
<ESC>nB/
```

where n is the address of the breakpoint. DDT displays the address of breakpoint n, and you can use the examine commands to open and display the instruction at breakpoint n. If breakpoint n is not set, DDT displays zero.

To remove breakpoint n, type in

```
0<ESC>nB
```

To remove all breakpoints, type in

```
<ESC>B
```

5.2.2 Proceeding from Breakpoints

After your program has reached a breakpoint, you can continue execution at "$." by typing in

<ESC>P

DDT saves the program instructions from all breakpoints, replaces the program instructions with JSRs to DDT, restores your user-program context, and if you have not executed any program instructions with the <ESC>X command, simulates execution of the instruction at the breakpoint. DDT then executes a JRSTF (in section zero, otherwise DDT executes an XJRSTF) to the next instruction to be executed.

You can cause the program to start execution at a different location with the {addr}<ESC>G command, where addr defaults to the program's start address.

Once your program has reached a breakpoint and DDT has interrupted execution, you can cause DDT to continue execution but NOT stop at that breakpoint until your program has reached that breakpoint a specified number of times. To do this, type in

expr<ESC>P

where expr is the proceed count. DDT places expr at location $nB+2, where n is the number of the breakpoint at which your program has stopped. DDT resumes execution of your program. Each time your program reaches breakpoint n, DDT decrements the proceed count stored at $nB+2. Your program continues execution until:

- it reaches a different breakpoint
- it terminates normally
- it commits a fatal error
- the proceed count reaches zero

Each breakpoint has an associated automatic proceed flag. If this flag is set and the program reaches the breakpoint, DDT decrements the proceed count at $nB+2 (where n is the number of the breakpoint) and displays the breakpoint information if the proceed count is less than one. DDT then automatically continues program execution.

The <ESC>P command clears the automatic proceed flag associated with the breakpoint at which DDT has suspended program execution.

To set a breakpoint and set the associated automatic proceed flag, type in

```
{addr1<}addr2<ESC><ESC>{n}B
```

where addr2 is the address of the breakpoint and may be ".", addr1 is an (optional) additional location to be displayed, and n is optional and defaults to the lowest unused breakpoint.

Each time your program reaches breakpoint n, DDT decrements the associated proceed count, and if the result is less than one, displays

```
$nB>>addr2/   instr
```

where n is the breakpoint number, addr2 is the address of the breakpoint, and instr is the contents of the word at addr2.

If you typed in addr1< when you gave the command, DDT displays

```
$nB>>addr2/   instr   addr1/   contents
```

where n is the breakpoint number, addr2 is the address of the breakpoint, instr is the contents of the word at addr2, addr1 is the additional location to be displayed, and contents is the contents of the word at addr1.

DDT then automatically continues program execution until:

- your program reaches a different breakpoint
- your program terminates normally
- your program commits a fatal error
- you type any character while your program is at breakpoint n

You can interrupt the automatic proceed function if you type in a character while your program is at breakpoint n. DDT then resets the automatic proceed flag and suspends program execution at the breakpoint. DDT echoes the character that you typed in, which is otherwise ignored.

To proceed from a breakpoint and set the associated automatic proceed flag, give the command

```
{expr}<ESC><ESC>P
```

where expr is the proceed count. DDT stores the proceed count at $nB+2.

5.2.3 Conditional Breakpoints

To cause DDT to interrupt program execution at a breakpoint only if a specific condition is satisfied, you must store a single test instruction or a call to a test routine in DDT's breakpoint table. You can use a test routine in your program, or one that you enter in DDT's patching area. See Chapter 8 (Inserting Patches with DDT) for more information about the patching area. To type in the test instruction (or the call to the test routine), open the DDT location addressed by the command <ESC>nB+1 by typing in

```
<ESC>nB+1/
```

where n is the number of the breakpoint. You must type in n, or DDT interprets the command as <ESC>B, and removes all breakpoints. Deposit the test instruction or the call to the test subroutine. The instruction must result in a PC (program counter) skip when the condition to trigger the breakpoint is true. If your program reaches breakpoint n, DDT executes the instruction at $nB+1. DDT then proceeds as follows:

- If a program counter skip does occur, DDT interrupts execution at breakpoint n.
- If the instruction does not cause a program counter skip, DDT decrements the proceed count at $nB+2. If the result is zero or less, DDT interrupts execution at breakpoint n.
- If the conditional instruction is a call to a subroutine that returns by skipping over two instructions, DDT does not interrupt program execution.

If DDT interrupts execution because the test instruction resulted in a program counter skip, DDT displays only one angle bracket after the breakpoint identification, as

```
$3B>LABL1/   MOVE 1,LABL2
```

5.2.4 The "Unsolicited" Breakpoint

You can cause your MACRO program to "call" DDT by inserting the following instruction in your program:

```
JSR $0BPT##
```

The two number signs (##) appended to $0BPT in your MACRO program declare the symbol as EXTERNAL.

NOTE

> The "$" in "$0BPT" represents the dollar sign character, which is part of the symbol, and is not the DDT echo of the ESCAPE key.

For your program to "call" DDT, you must load RDDT.REL with your program or you will get a LINK error when you load your program (?LNKUGS undefined global symbol). Load RDDT.REL with your program as follows (your input is in lowercase; MYPROG is the name of your program; the last line indicates that DDT is loaded and ready to accept your commands):

```
@link
*sys:rddt.rel,myprog/go
@save myprog.exe
```

Load your program after RDDT.REL to ensure that the saved program module has your program's start address. Start your program running with the START or RUN commands. If your program executes the JSR instruction, DDT interrupts program execution and displays

```
$0B>>addr+1/   instr
```

where addr+1 is the first location after the JSR $0BPT instruction, and instr is the contents of that location.

5.3 EXECUTING EXPLICIT INSTRUCTIONS

To execute a specific instruction, type in the instruction followed by <ESC>X, as follows:

```
instr<ESC>X
```

For example:

```
MOVE 1,@LABL1(3)<ESC>X
```

After executing the instruction, DDT starts a new line and displays:

- <> if in-line execution of instr would result in skipping no instructions.
- <SKIP> if in-line execution of instr would result in skipping 1 instruction.
- <SKIP 2> if in-line execution of instr would result in skipping 2 instructions.
- <SKIP 3> if in-line execution of instr would result in skipping 3 instructions.

NOTE

"In-line execution" means execution of the instruction as part of normal program flow. The execution of instructions with this command has no effect on your user-program PC.

This command restores the prevailing display mode.

5.4 SINGLE-STEPPING INSTRUCTIONS

After your program has transferred control to DDT from a breakpoint, you can execute program instructions one at a time. This is called "single-stepping."

"<ESC>." is a command that returns the address of the next instruction to be executed.

To execute the instruction whose address is returned by "<ESC>.", type in

```
<ESC>X
```

For example, breakpoint 3 is set at LABL1+3. If your program PC reaches LABL1+3, control passes to DDT, which displays

```
$3B>>LABL1+3/   ADD 1,LABL2(2)
```

Examining the environment, you learn the following:

- AC 1 contains 1
- AC 2 contains 3
- LABL1+4 contains MOVEM 1,@LABL2(3)
- LABL2+3 contains SYM3

as shown by the following terminal display (DDT does not display <LF> or <ESC>):

```
$3B>>LABL1+3/   ADD 1,LABL2(2)   <ESC>\   SYM3   <LF>
LABL1+4/   MOVEM 1,@LABL2(3)   1/   1   <LF>
2/   3
```

If you now type in the command <ESC>X, DDT does the following:

- changes "$$." to LABL1+3
- executes the instruction at LABL1+3
- changes "$." to LABL1+4
- changes the current location to LABL1+4
- opens LABL1+4
- displays

```
   1/   SYM3+1   LABL2+3/   SYM3
LABL1+4/   MOVEM 1,@LABL2(3)
```

If single-stepping an instruction results in a value of ($. minus $$.) not equal to 1, DDT also begins a new line and displays:

- <SKIP> if ($. minus $$.) = 2
- <SKIP 2> if ($. minus $$.) = 3
- <SKIP 3> if ($. minus $$.) = 4
- <JUMP> if ($. minus $$.) is greater than 4 or less than 1

before displaying the address and contents of the next instruction to be executed. For example, the following shows a typical terminal display where you type in <ESC>X to single-step the first instruction at a breakpoint (DDT echoes <ESC> as $):

```
$4B>>LABL1+5/   AOSN 3   /   0    <ESC>x
   3/   1
<SKIP>
LABL1+7/   MOVEM 1,LABL2
```

If the instruction that you execute is followed by an ERCAL or an ERJMP instruction and DDT executes the instruction successfully, DDT displays <ERSKP> before displaying the next instruction to be executed. For example:

```
ADDR1/   GTJFN   <ESC>x
   <ERSKP>
ADDR1+2/   instr
```

where instr is the instruction at location ADDR1+2.

If DDT does not execute the instruction successfully, it displays (in the case of an ERJMP)

```
ADDR1/   GTJFN   <ESC>x

ADDR1+1/   ERJMP   EROUTN
   EROUTN
<JUMP>
EROUTN/   instr
```

where instr is the instruction at location EROUTN.

5.5 EXECUTING SUBROUTINES AND RANGES OF INSTRUCTIONS

To execute a series of n instructions beginning with the instruction whose address is returned by the command "<ESC>.", type in

```
n<ESC>X
```

where n is the number of instructions to execute.

DDT then does the following for each instruction:

- starts a new display line
- executes the instruction
- displays the address of any register or memory location referenced by the execution of the instruction, and the contents of those locations after execution of the instruction
- changes the current location to the next instruction to be executed
- opens the current location
- displays the address and contents of the next instruction to be executed
- changes "$." to the address of the next instruction to be executed
- changes "$$." to the address of the instruction just executed

To suppress typeout of all but the last instruction executed, use the command

```
n<ESC><ESC>X
```

where n is the number of instructions to execute.

To continue program execution until the PC (program counter) enters a range of instructions, type in

```
{addr1<}{addr2>}<ESC><ESC>X
```

where addr1 is the lower end of the range, and addr2 is the upper end. Addr1 defaults to 1 + "$." and addr2 defaults to addr1 + 3. Follow addr1 with a left angle bracket (<) and addr2 with a right angle bracket (>).

This command also indicates skips and jumps.

This command is useful for executing a loop or a subroutine call quickly and without typeout.

For example, breakpoint 3 is at location LABL1.

```
$3B>>LABL1/   PUSHJ 17,SUBRTN   <ESC><ESC>X  ;Type in <ESC><ESC>X
<SKIP>                                       ;SUBRTN returns + 2
LABL1+2/   ADD 1,2
```

If you type in a question mark (?) while DDT is executing an <ESC><ESC>X command, DDT displays

```
Executing: addr/   instr
```

where addr is the address of the next instruction to be executed, and instr is the instruction.

To terminate the execution of the series of instructions, type in any character other than ? (question mark). DDT does the following:

- echoes the character
- displays <SKIP>, <SKIP 2>, <SKIP 3>, or <JUMP>, as appropriate
- starts a new display line
- changes the current location to the address of the next instruction to be executed
- displays the address and contents of the current location
- opens the current location
- waits for your next command

5.6 SINGLE-STEPPING "DANGEROUS" INSTRUCTIONS

DDT classifies the following as "dangerous" instructions:

- instructions that can modify memory
- instructions that can cause an arithmetic trap
- instructions that can cause a stack overflow
- a monitor call or I/O instruction

Before single-stepping one of these instructions, DDT saves and replaces the original instructions at the breakpoints with JSRs to DDT, and restores the full user-program context (including interrupt system and terminal characteristics) before executing the instruction. After executing the instruction, DDT replaces the JSRs at the breakpoints with the original program instructions, and saves the full user-program context.

DDT does not check whether the instruction actually results in one of these conditions, only whether the opcode is in the class of instructions that can cause these effects. This can make executing subroutines and ranges of instructions under DDT control extremely time-consuming.

To execute an instruction, a series of instructions, or a subroutine without checking for dangerous instructions, substitute "1X" for "X" in any of the DDT commands described in this chapter. For example, to execute a subroutine or series of instructions without checking for dangerous instructions, use the command

{addr1<}{addr2>}<ESC><ESC>1X

where addr1 is the lower end of the range, and addr2 is the upper end. Addr1 defaults to 1 + "$." and addr2 defaults to 3 + addr1. Follow addr1 with a left angle bracket (<), and addr2 with a right angle bracket (>).

DDT executes your commands much faster when not checking for dangerous instructions, but if the execution of an instruction causes a software interrupt, the error and trap handling mechanism may not function correctly. In addition, program instructions that change or rely on terminal or job characteristics that are also used by DDT can cause unpredictable results.

5.7 USER-PROGRAM CONTEXT

When DDT interrupts your program's execution at a breakpoint, and after it has executed a dangerous instruction during an <ESC>X or <ESC><ESC>X command, it saves the user-program context. The command <ESC>I+n, where 0<=n<=10 (octal), returns the address of the word that contains the information for "function" n (if n = 0, you can omit "+n" from the command). You can use this address to display and modify these values. Most of these values are useful only in executive mode. DDT displays the address of the word that contains the information for function n as

```
$I+n
```

where 1<=n<=10 (octal). If n = 0, DDT displays only $I. This command is illegal in FILDDT.

Table 5-2 lists the functions and the associated user-context values.

Table 5-2: User-Program Context Values

FUNCTION	VALUE
0	Executive mode CONI PI.
1	Executive mode PI channels turned off.
2	Executive mode CONI APR.
3	User PC flags.
4	User PC address.
5	EPT page address.
6	UPT page address.
7	CST base virtual address.
10	SPT base virtual address.

DDT restores the user-program context whenever you execute <ESC>G, <ESC>P, and when you execute <ESC>X, or <ESC><ESC>X of dangerous instructions.

Functions 5 through 10 (octal) affect DDT's interpretation of your program's virtual address space. You can alter DDT's interpretation of your program's virtual address space with the physical and virtual addressing (<ESC>nU) commands described in Chapter 11 (Physical and Virtual Addressing Commands). However, any alterations that you make do not become part of your user-program context, and do not affect TOPS-20's interpretation of your program's virtual address space.

DDT also saves and restores the user-program ACs as part of the user-program context. DDT stores the contents of the ACs in an internal "register" block. Any references you make to addresses 0-17 refer to the relative locations in DDT's internal register block. These actions are totally transparent to you.

CHAPTER 6

SEARCHING FOR DATA PATTERNS IN DDT

With DDT you can search for memory locations that contain a specific value, and conversely, for words that do not contain a specific value. You can also set a mask to indicate to DDT that only specified bits are to be considered when performing the search. In addition, you can search for words that reference a specific address. You can specify a range within which to perform the search, or default the range to all of your program's address space. In either case, DDT compares the contents of each location within the range with the specified value.

To search for words that match a specific value, type in

```
{addr1<}{addr2>}expr<ESC>W
```

where expr is the value for which DDT is to search (expr can be any legal DDT expression), and addr1 and addr2 delimit the range in which the search is to be conducted. Follow addr1 with a left angle bracket (<) and addr2 with a right angle bracket (>). Addr1 defaults to zero and addr2 defaults to 777777 in the current section, unless you are in FILDDT, and:

- the target is an .EXE file and you are using normal virtual addressing OR
- the target is a disk structure or data file.

In these cases, addr2 defaults to the last word of the target. See Chapter 9 (FILDDT), and Chapter 11 (Physical and Virtual Addressing Commands), for more information.

SEARCHING FOR DATA PATTERNS IN DDT

When you search for a specific value, DDT does the following:

- compares each location (after ANDing it with the search mask) within the search range with the 36-bit value resulting from evaluating expr
- starts the search by comparing the contents of addr1 with expr
- stops the search after comparing the contents of addr2 with expr
- displays (on a new line) the address and contents of each location that matches expr
- enters the address of each matching location on the location sequence stack
- sets the current location to addr2
- displays a blank line to indicate the search is over
- restores the prevailing display mode

NOTE

If DDT finds more matching locations than there are words on the location sequence stack, each new entry overwrites the earliest entry.

If you type in a question mark (?) while DDT is performing this search, DDT displays

Searching: addr/ value

where addr is the address of the location that will next compare, and value is the contents of addr.

To terminate the search, type in any character other than question mark (?). DDT stops searching, and waits for more input. The character that you type in to terminate the search is otherwise ignored.

SEARCHING FOR DATA PATTERNS IN DDT

To search for words that do NOT match a specified value, type in

{addr1<}{addr2>}expr<ESC>N

where expr is the value that is not to be matched (expr can be any legal DDT expression), and addr1 and addr2 delimit the range within which DDT is to search. Follow addr1 with a left angle bracket (<) and addr2 with a right angle bracket (>). Addr1 defaults to zero and addr2 defaults to 777777 in the current section unless you are in FILDDT, and:

- the target is an .EXE file and you are using normal virtual addressing OR
- the target is a disk structure or data file.

In these cases, addr2 defaults to the last word of the target. See Chapter 9 (FILDDT), and Chapter 11 (Physical and Virtual Addressing Commands), for more information.

DDT functions as for the <ESC>W command, except:

- DDT searches for and displays the address and contents of any word within the address range that does NOT match the 36-bit value resulting from evaluating expr.
- DDT enters the locations of non-matching words on the location sequence stack.

This search command restores the prevailing display mode.

NOTE

If DDT finds more nonmatching locations than there are words on the location sequence stack, each new entry overwrites the earliest entry.

If you type in a question mark (?) while DDT is performing this search, DDT displays

Searching: addr/ value

where addr is the address of the location that will next compare, and value is the contents of addr.

To terminate the search, type in any character other than question mark (?). DDT stops searching, and waits for more input. The character that you type in to terminate the search is otherwise ignored.

SEARCHING FOR DATA PATTERNS IN DDT

To search for references to an address, type in

{addr1<}{addr2>}expr<ESC>E

where addr1 and addr2 delimit the range of the search, and expr contains the address for which DDT is to search. Follow addr1 with a left angle bracket (<) and addr2 with a right angle bracket (>). Addr1 defaults to zero and addr2 defaults to 777777 in the current section. Expr is any legal DDT expression.

DDT performs an IFIW effective address calculation on the value contained in each word within the range, and uses the 30-bit result to determine whether there is a match.

Thus, if bits 14-17 (the X field of an instruction) or bit 13 (the I field of an instruction) are nonzero, indexing or indirection may result in DDT finding different search results at different times.

DDT does not check whether the value is actually an instruction before performing the effective address calculation.

This search command restores the prevailing display mode.

NOTE

If DDT finds more matching locations than there are words on the location sequence stack, each new entry overwrites the earliest entry.

If you type in a question mark (?) while DDT is performing this search, DDT displays

Searching: addr/ value

where addr is the address of the location that will next compare, and value is the contents of addr.

To terminate the search, type in any character other than question mark (?). DDT stops searching, and waits for more input. The character that you type in to terminate the search is otherwise ignored.

SEARCHING FOR DATA PATTERNS IN DDT

<ESC>M is a command that addresses a DDT location that contains a search mask used to prevent specified bits in the memory word from being considered during the search. This mask is used only by <ESC>W and <ESC>N, not by <ESC>E. DDT logically ANDs the search mask with the memory word before making the comparison, but does not change the memory word. If DDT finds a match, it displays the entire word.

DDT sets the search mask to 777777,,777777 (compare all 36 bits) by default.

To set the search mask, type in

```
expr<ESC>M
```

where expr evaluates to the required bit pattern.

For example, to search for all of the RADIX50 references to MAIN., you could use the following three commands:

```
<ESC><ESC>5t              ;Set typeout mode to RADIX50.
37777,,777777<ESC>m       ;Ignore the left 4 bits.
main.<ESC>5"<ESC>w        ;Type in RADIX50 symbol, start search.
```

The following screen display is a typical result of the above search (DDT does not display the comments on the right):

```
4112/   4 MAIN.           ;DDT displays match found.
4775/   0 MAIN.           ;DDT displays match found.
                          ;Search over, DDT displays blank line.
```

You can also examine and modify the search mask with the examine and deposit commands described in Chapter 4 (Displaying and Modifying Memory).

CHAPTER 7

MANIPULATING SYMBOLS IN DDT

7.1 OPENING AND CLOSING SYMBOL TABLES

Each separate program module has its own symbol table. When displaying a value symbolically, if more than one symbol is defined with that value, DDT displays the first global symbol found. When searching for a symbol, DDT searches the "open" symbol table first. For display purposes, DDT treats local symbols found in the open symbol table as global symbols. DDT appends a number sign (#) to local symbol names that it finds in a symbol table that is not open. For example, DDT might display

```
SYMBL1#
```

where SYMBL1 is a local symbol that DDT found in a symbol table that is not open.

If you type in an expression that contains a symbol that is defined in more than one of your program modules, DDT uses the value of the symbol that is contained in the open symbol table. If the symbol is not defined in the open symbol table, or if there is no open module and no global definition of the symbol, DDT displays

```
M
```

To open the symbol table of a program module, type in

```
name<ESC>:
```

where name is the name of the program module as specified by the TITLE pseudo-op in your MACRO-10 program (or the equivalent mechanism in a higher-level language program). DDT closes any currently open symbol table and opens the symbol table associated with module name.

To find the name of the module associated with the open symbol table, type in

```
<ESC>1:
```

If there is an open symbol table, DDT displays the name of module associated with the open symbol table. For example, if the symbol table for module X is open, the screen display is as follows (DDT echoes <ESC> as $, and does not display any spaces between the command and the module name):

```
$1:/X
```

If there is no open symbol table, DDT displays three spaces (or a tab, depending on the TTY control mask), and waits for your next command.

To close the open symbol table, type in

```
<ESC>:
```

7.2 DEFINING SYMBOLS

To redefine a symbol or to create a new symbol in the current symbol table, type in

```
expr<symbol:
```

where expr is any legal DDT expression, and symbol is the symbol name.

To define symbol as the local address of the open location, type in the command

```
symbol:
```

DDT defines symbol with a value of 0,,addr, where addr is the address of the open location. If there is no open location, DDT uses the address of the last location that was open. DDT defines symbol as a global symbol even if it was previously a local symbol. If you previously used symbol as an undefined symbol, DDT inserts the correct value in all the places you referenced symbol, and removes symbol from the undefined symbol table.

To define symbol as the global (30-bit) address of the open location, type in

```
.<symbol:
```

7.3 KILLING SYMBOLS

To remove a symbol from the symbol table, type in

```
symbol<ESC><ESC>K
```

DDT removes symbol from the symbol table, and no longer displays symbol or accepts symbol as input.

7.4 SUPPRESSING SYMBOL TYPEOUT

To prevent a symbol from being displayed, type in

```
symbol<ESC>K
```

where symbol is the symbol to be suppressed. DDT still accepts symbol as input, but no longer displays symbol as output.

To suppress the last symbol that DDT displayed (in an address, in the contents of a memory word, or in the evaluation of an expression), type in

```
<ESC>D
```

DDT suppresses the last symbol displayed, and then redisplays the current quantity. DDT does not display its usual three spaces between the command and the displayed value.

In the following example, assume that symbol SIZE is defined as 3. User typein is lowercase (<LF> does not appear on the terminal screen).

```
start/   JFCL 0    <LF>
LOOP/    AOS 1    <LF>
LOOP+1/    MOVE 2,1    <ESC>dMOVE 2,1    <LF>
START+3/    MULI 2,SIZE    <ESC>dMULI 2,3
```

To reactivate a symbol for typeout, redefine the symbol. For example, to reactivate the display of symbol SIZE, above, type in

```
size<size:
```

Note that SIZE is now defined as a global symbol, even if it was previously a local symbol.

7.5 CREATING UNDEFINED SYMBOLS

It is sometimes convenient to use symbols that have not yet been defined. To create an undefined symbol, type in

```
symbol#
```

where symbol is the undefined symbol name. DDT enters symbol in the undefined symbol table. When you later define the symbol, DDT enters it into the defined symbol table, removes it from the undefined symbol table, and enters the correct value in all locations where you referenced the symbol.

You can use undefined symbols only as parts of expressions that you are depositing to memory. Undefined symbols can be either fullword or right-halfword values; they cannot be used as the A or X fields of an instruction, or as the left-halfword of an expression.

7.6 SEARCHING FOR SYMBOLS

To determine the modules in which a symbol is defined, type in

```
symbol?
```

where symbol is the name of the symbol. DDT displays the name of each program module in which symbol is defined. If the symbol is a global symbol, DDT displays a "G", as in the following:

```
sym?
MAIN.  G
```

DDT does not display G following a local symbol found in the open symbol table. When DDT has searched the entire symbol table, it displays a blank line. To search for all the symbols that begin with a specific character pattern, use the command

```
sym<ESC>?
```

where sym is the character pattern for which you are searching, and may be one to six characters long. DDT searches your symbol tables and displays all symbols that begin with that pattern. DDT also displays all modules in which the symbol is found, whether the symbol is global, and the value of the symbol. In addition, if the symbol represents a value in which only one bit is set, DDT displays the number of the bit. For example, the command

```
fdb<ESC>?
```

might cause the following display:

```
FDBIN    INOUT   G   3
FDBOUT   INOUT   G   2   (1B34)
FDB      MOD1        7
```

7.7 LISTING UNDEFINED SYMBOLS

To get a list of all currently undefined symbols, type in

```
?
```

DDT displays a list containing each undefined symbol.

7.8 LOCATING SYMBOL TABLES WITH PROGRAM DATA VECTORS

DDT Version 43 can access symbol tables pointed to by JOBDAT, by PDVs (program data vectors) and by values you store in DDT.

The command <ESC>5M returns the address of a DDT location that contains information to direct DDT to the current symbol table. The symbol $5M refers to the memory location at the address returned by the <ESC>5M command.

If the value contained in $5M is negative (bit 0 is set), the right-halfword contains the number of the section that contains the JOBDAT area.

If the value contained in $5M is positive (bit zero is clear, and the value in $5M is nonzero), $5M contains the 30-bit address of the PDV currently in use by DDT.

If $5M contains zero, DDT uses values (which can be stored by the user) pointed to by locations 770001 and 770002 of UDDT, to determine which symbol table(s) to use. The algorithm that DDT uses is described below.

To set $5M to a PDV address, type in

```
addr<ESC>5M
```

where addr is the the PDV address. If you know the PDV name, you can type in

```
<ESC><ESC>:/name/
```

where name is the name of the PDV, and the slashes (/) represent any characters that do not appear in name. If name is a null string, DDT searches for a PDV with no name or a null name. DDT ignores any characters in name beyond a length of 39.

DDT searches for a PDV named name, and places its address in $5M. If DDT does not find the PDV, it displays ? and sounds the terminal buzzer or bell.

You can learn the names of the PDVs associated with your program by using the following sequence of TOPS-20 commands:

```
@GET program-name
@INFORMATION VERSION
```

To display the name of the PDV addressed by $5M, type in

```
<ESC><ESC>1:
```

If $5M contains the address of a PDV, DDT displays the name of the PDV; otherwise, it does nothing.

Whenever DDT is entered from its start address or from a breakpoint, if $5M is zero, DDT initializes $5M according to the following rules:

- If XDDT was started by the UDDT stub, AND the location addressed by location 770001 in the stub has bit zero set:
  - DDT uses the location addressed by location 770001 (in the stub) as an IOWD pointer to a symbol table in the section that contains the stub.
  - DDT uses the location addressed by location 770002 (in the stub) as as IOWD pointer to the undefined symbol table in the section that contains the stub.
- If XDDT was not started by the UDDT stub, OR the location addressed by location 770001 in the stub has bit zero clear:
  - If no PDVs exist, DDT sets $5M to -1,,n, where n is:
    * the section that contains the UDDT stub (if the stub exists) OR
    * the section that contains the entry vector (if an entry vector exists) OR
    * section zero.
  - If there is one (only) PDV, DDT sets $5M to the address of the PDV.
  - If there is more than one PDV, DDT examines word .PVSYM of each PDV in ascending memory order (DDT first looks at the PDV closest to 0,,0). DDT then sets $5M to:
    * the address of the first (lowest in memory) PDV that contains a .PVSYM word that contains a global address (if there is one).
    * the address of the first (lowest in memory) PDV that exists in or above the section containing the entry vector (if there is one).
    * the address of the first (lowest in memory) PDV.

NOTE

DDT ignores its own PDV when setting $5M.

CHAPTER 8

INSERTING PATCHES WITH DDT

To replace the instruction at the open location with a series of instructions and test the new instructions without reassembling your program, you can use the DDT patch function. DDT deposits (in a patching area) the replaced instruction, the new series of instructions, and zero or more JUMPA instructions back to the main line of your program. DDT also deposits (in the location that contains the replaced instruction) a JUMPA instruction to the first word of the patch.

To insert a patch that will be executed before the instruction at the open location, type in

{expr}<ESC><

where expr is the start of the patching location, and defaults first to PAT.., then to PATCH. KDDT and MDDT default to FFF (an area created during the monitor build), PAT.., and PATCH, in that order. If you do not type in expr, and DDT finds none of the default symbols, DDT uses the value contained in JOBDAT location .JBFF as the address to begin the patch. If expr is a symbol (or the default), DDT updates the symbol table when you terminate the patch, so that the symbol identifies the first word after the patch that you just terminated.

If the left half of expr is zero, DDT defaults the section to the section that contains the open location. If the left half of expr is a value that is not the section that contains the open location, DDT displays the message

?CAN'T PATCH ACROSS SECTIONS

If there is no open location when you start the patch, DDT displays "?" and sounds the terminal buzzer or bell.

NOTE

If expr is an AC address, or resolves to a value less than 0,,140, DDT displays "?" and sounds the terminal buzzer or bell.

INSERTING PATCHES WITH DDT

When you give a command to start a patch, DDT saves the address of the open location, closes the open location, changes the current location to the first word in the patching area, and opens that word. DDT also displays the address and contents of the first word of the patching area. For example:

```
<ESC><
PAT../   0
```

You can now type in the patch, using deposit instructions (the expr<LF> format is probably most useful). DDT updates the current and open locations according to the rules for the command that you use.

To terminate the patch, type in

```
{expr}<ESC>{n}>
```

where expr is the last word of the patch you are typing in, and n is the number of returns possible from execution of the patch. The default for n is 2, allowing for a return to 1 + the address of the instruction being replaced, and for a "skip return" to 2 + the address of the instruction being replaced.

When you terminate the patch, DDT deposits the instruction being replaced into the first location following the current location, unless:

- display is not suppressed by ! AND
- the current location is zero AND
- the current location is closed OR you omitted expr

in which case DDT deposits the instruction being replaced into the current location. This prevents the patch from containing unintended null words.

DDT deposits n JUMPA instructions in the locations immediately following the one in which it deposited the original program instruction. The first JUMPA instruction has 1 in its A field, and jumps to 1 + the address of the replaced instruction, the second JUMPA instruction has 2 in its A field and jumps to 2 + the address of the replaced instruction, and so on. The AC numbers are used for identification purposes only. Any JUMPA instruction beyond the sixteenth contains 17 in its A field.

DDT then changes the current location to the location that was open when you initiated the patch, deposits in the current location a JUMPA instruction to the first word of the patch that you typed in, and displays the address, original contents, and new contents of the current location. The current location is "open", and can be modified by your next command.

If you default expr, or type in a symbol in the {expr}<ESC>< command, when you terminate the patch, DDT redefines the symbol that identifies the start of the patch. If DDT used the value contained in JOBDAT location .JBFF as the address of the patching area, DDT changes the values contained in .JBFF and the left half of JOBDAT location .JBSA. In all cases, the new value is the address of the memory location after the last word of the patch.

By default, there are 100 (octal) words in the patching area. DDT does not check whether your patch overflows the patching area. You can control the size of the patching area with the /PATCHSIZE switch in LINK.

NOTE

DDT allows you to use other DDT commands while you are in the process of typing in a patch. DDT does not check whether the current and open locations are in the patching area, or whether you are typing in patch instructions in sequence. When you terminate the patch, DDT deposits the instruction being replaced in the current location regardless of whether the current location is in the patching area.

To insert a patch that will be executed after the instruction at the open location, type in

{expr}<ESC><ESC><

where expr is the address of the patching location (PAT.. is the default). The results are the same as inserting the patch before the instruction as above, except:

- When you open the patch DDT deposits the replaced instruction in the first word of the patch.
- When you terminate the patch, DDT deposits the first JUMPA instruction (rather than the instruction being replaced) in the first location following the current location unless:
  - display is not suppressed by ! AND
  - the current location contains zero AND
  - the current location is closed OR you omitted expr

  in which case DDT deposits the first JUMPA instruction in the current location. This is to prevent the patch from containing unintended null words.

NOTE

If expr is an AC address, or resolves to a value less than 0,,140, DDT displays "?" and sounds the terminal buzzer or bell.

Figure 8-1 illustrates the patching function. The program being patched is X.MAC (see Figure 2-1). The patch inserts a SKIPN instruction that is to be executed before the instruction at START+4.

DDT Output	User Input	Explanation
START+4/ MOVE 2(IDX)		As a result of your last command, DDT displays the contents of START+4.
	<ESC><	Type in <ESC>< to start the patch.
PAT../ 0		DDT displays the address and contents of the first word word of the patch area.
	.=	Check the address of the current location (PAT..).
176		DDT displays the current address of PAT...
	skipn 1,0<ESC>1>	Type in the new instruction, and terminate the patch with a normal return and no skip return by typing in <ESC>1>.
PAT..+1/ 0 MOVE 2(IDX)		DDT displays the address and contents of the next word of the patch area, then deposits and displays the instruction from START+4 in the next word of the patch area.
PAT..+2/ 0 JUMPA 1,START+5		DDT displays the next word of the patch area, then deposits a JUMPA instruction to 1 + the address of the replaced instruction.
START+4/ MOVE 2(IDX) JUMPA STACK+17		DDT displays the address and original contents of the replaced instruction, then deposits and displays a JUMPA instruction to the first word of the patch. START+4 is the current location, and is "open".
	pat..=	Check the address of the patch area.
201		DDT updated "PAT..".

Figure 8-1: Annotated Patching Session

Figure 8-2 shows the terminal display as it actually appears when you insert the patch described above. Your input is in lowercase.

```
START+4/   MOVE 2(IDX)    $<
PAT../   0    .=176    skipn 1,0$1>
PAT..+1/   0    MOVE  2(IDX)
PAT..+2/   0    JUMPA 1,START+5
START+4/   MOVE 2(IDX)    JUMPA STACK+17    pat..=201
```

Figure 8-2: Terminal Display of Patching Before an Instruction

Figure 8-3 shows the terminal display when inserting the same patch after the instruction at START+4. You type in the instruction in the form expr<LF> (user input is lowercase). Note the use of the patch termination command without expr and without n (n defaults to 2).

```
START+4/   MOVE 2(IDX)    $$<
PAT../   0    MOVE 2(IDX)
PAT..+1/   0    pat..=176    skipn 1,0
PAT..+2/   0    $>
PAT..+2/   0    JUMPA 1,START+5
PAT..+3/   0    JUMPA 2,START+6
START+4/   MOVE 2(IDX)    JUMPA STACK+17    pat..=202
```

Figure 8-3: Terminal Display of Patching After an Instruction

To abort the patch you are typing in, type in

```
<ESC>0<
```

DDT displays 3 spaces (or a tab, depending on the TTY control mask) and changes the current location to the location that was open when you initiated the patch. The symbol that denotes the start of the patching area is unchanged. Any deposits that you made as part of the patch remain in the patching area. This allows you to restart the same patch, or to write over the patch with a new one.

CHAPTER 9

FILDDT

9.1 INTRODUCTION

FILDDT is a utility used to examine and change disk files and physical disk blocks. You can also use FILDDT to examine monitor crash dumps, and to examine the running monitor. With FILDDT, you can look at .EXE files as if they had been loaded with the monitor GET command, or as if they were binary data files.

In selecting a disk file, a disk, or the monitor with FILDDT, you are really establishing the virtual address space that FILDDT accesses. When discussing the contents of that virtual address space, where the contents can be any of the above objects, this chapter uses the term target.

Once you have accessed a target, you can examine and modify it with the DDT examine and modify commands (you cannot modify the running monitor with FILDDT), and then save it with your modifications. You can use all of DDT's commands for examining and modifying memory, but you cannot use any commands that cause the execution of program instructions, such as <ESC>X, <ESC>G, and so on. If you attempt to execute a program instruction, FILDDT sounds the terminal buzzer or bell.

9.2 USING FILDDT

There are two command levels in FILDDT. This document refers to these two levels as FILDDT command level and DDT command level.

FILDDT command level accepts FILDDT commands to control session parameters and to select the target. FILDDT command level employs TOPS-20 command recognition and help. When at FILDDT command level, FILDDT displays the prompt

```
FILDDT>
```

Once you access a target, FILDDT enters DDT command level. At DDT command level, use DDT commands to examine and modify the target.

The syntax of a FILDDT command-level command is

```
command {file-spec{/switch...}}
```

where command is a FILDDT command-level command, file-spec is a TOPS-20 file specification, and switch invokes a specific function or parameter about the function that you can perform (enable patching, for example). You can use FILDDT commands to invoke functions and parameters that are invoked by analogous FILDDT switches.

With a FILDDT command you can:

- request HELP on FILDDT
- specify the target to be examined
- invoke FILDDT functions
- establish certain parameters about the functions that you can perform
- enter DDT command level
- exit FILDDT

A FILDDT command can have more than one of the above effects.

The commands and switches are described in detail in the rest of this chapter.

To start FILDDT, type the TOPS-20 command

```
FILDDT
```

FILDDT enters FILDDT command level and prompts

```
FILDDT>
```

You can now use the FILDDT commands described on the following pages.

9.2.1 FILDDT Commands

There are two classes of FILDDT-level commands; those that select the target that FILDDT is to access and those that establish what function FILDDT is to perform for the target (enable patching, extract symbols, and treat an .EXE file as data). The following are the targets (virtual address spaces) that FILDDT can access, and the commands that select them:

• disk files	GET
• disk structures	DRIVE, STRUCTURE
• the running monitor	PEEK

To examine and patch disk structures, you must have WHEEL, OPERATOR, or MAINTENANCE privileges enabled. To examine the running monitor, you must have WHEEL or OPERATOR privileges enabled.

Following are the parameters you can invoke for the target and the commands and switches that select them:

• treat file as pure binary data	ENABLE DATA-FILE	/DATA
• enable patching	ENABLE PATCHING	/PATCH
• enable thawed access	ENABLE THAWED	/THAWED
• load symbol table only from file	LOAD	/SYMBOL

To get HELP, type in

```
HELP
```

FILDDT displays a very brief description of the FILDDT commands and redisplays the FILDDT> prompt.

To return to TOPS-20 command level from FILDDT command level, type in

```
EXIT
```

9.2.2 Symbols

To enhance performance, FILDDT uses a symbol table that it builds in its own address space, rather than one which exists in the target address space.

FILDDT automatically extracts symbols to build its internal symbol table from the first .EXE file it loads during a session. Once FILDDT has an internal symbol table, it ignores any symbols in subsequently loaded .EXE files unless you use the LOAD command or the GET command with the SYMBOL switch.

9.2.3 Commands to Establish Formats and Parameters

ENABLE DATA-FILE

If you specify an .EXE file, DDT (by default) loads the file in virtual memory as if it were to be executed. You can use the ENABLE DATA-FILE command to look at an .EXE file as if it were a data file. FILDDT then loads the entire file (including the .EXE directory block) as a binary file, starting at virtual location zero. You can accomplish the same thing by appending the /DATA switch to the file-spec when you use the GET command.

ENABLE PATCHING

The ENABLE PATCHING lets you modify the target. You can also enable patching by appending the /PATCH switch to the file-spec when you use the GET command. If you do not enable patching, you can only examine the target. If you attempt to modify the target but have not enabled patching, FILDDT displays

```
? Patching is not enabled
```

Note that you cannot enable patching in FILDDT with the <ESC>W command.

ENABLE THAWED

The ENABLE THAWED command lets you examine and modify (if you enable patching) files that require thawed access. You can also use the /THAWED switch when loading the file with the GET command.

LOAD

The LOAD command tells FILDDT to copy the symbol table from the file named by file-spec. Once FILDDT has built its internal symbol table, FILDDT displays

```
[n symbols loaded from file]
```

where n is the number of symbols that FILDDT extracted from the file. FILDDT then again prompts you with FILDDT>.

You can also load symbols from the file you specify in the GET command by appending the /SYMBOL switch to the file-spec.

If the file you specify is not an .EXE file, FILDDT displays

```
% Not in .EXE format -- Data file assumed.
? Symbols cannot be extracted from a data file
```

FILDDT then redisplays its prompt.

9.2.4 Commands to Access the Target and Enter DDT

DRIVE

The DRIVE command allows you to access a physical structure directly. This may be useful if the home block has been damaged. To access disk structures with FILDDT, you must have WHEEL, OPERATOR, or MAINTENANCE privileges enabled.

If you wish to be able to patch the disk structure, you must give the ENABLE PATCHING command before using the DRIVE command.

To access the disk structure, type in

```
DRIVE c k u
```

where c is the channel, k is the controller, and u is the unit, in decimal.

If the unit is part of a mounted structure, FILDDT displays

```
[Unit is part of structure name]
```

where name is the logical name of the disk structure.

If FILDDT successfully access the unit, FILDDT enters DDT command level and displays

```
[Looking at unit u on controller k on channel c]
```

where c is the channel, k is the controller, and u is the unit, in decimal.

GET

The GET command tells FILDDT to load the file you name, invoke any parameters for which you specify switches, and enter DDT command level. Legal switches are /DATA, /PATCH, /SYMBOL, and /THAWED, and correspond to the ENABLE DATA-FILE, ENABLE PATCHING, LOAD, and ENABLE THAWED commands, respectively.

If FILDDT extracts symbols and builds an internal symbol table, it displays

```
[n symbols loaded from file]
```

where n is the number of symbols loaded.

When FILDDT has loaded the file, it displays

```
[Looking at file file-spec]
```

where file-spec is the TOPS-20 file specification of the file.

If FILDDT does not find the file, it displays

```
? Invalid file specification, message
```

where message is a TOPS-20 error string.

PEEK

Use the PEEK command to examine the running monitor. To use FILDDT to examine the running monitor, you must have WHEEL or OPERATOR privileges enabled.

Once you have invoked FILDDT, if you wish to be able to use monitor symbols when looking at the running monitor, you must use the LOAD command first, as:

```
LOAD SYSTEM:MONITR.EXE
```

You cannot patch the running monitor with FILDDT.

To examine the running monitor, type in

```
PEEK
```

FILDDT displays

```
[Looking at running monitor]
```

and enters DDT command level.

NOTE

You cannot use FILDDT to PEEK at the running monitor unless you are using normal virtual addressing. If you are PEEKing the monitor and change memory mapping to a mode other than normal virtual addressing with the n<ESC>0U, n<ESC>1U, n<ESC>2U, or $$U commands, FILDDT does not give an error. However, every page in the monitor then appears to DDT to be non-existent. In this case, most attempts to reference memory causes DDT to display ?, sound the terminal buzzer or bell, and set the error string to "CAN'T PEEK PHYSICAL". Searches do not cause errors, but never discover matches.

STRUCTURE

The STRUCTURE command allows you to access a disk structure by its logical name. To access disk structures with FILDDT, you must have WHEEL, OPERATOR, or MAINTENANCE privileges enabled.

If you wish to be able to patch the disk structure, you must give the ENABLE PATCHING command before using the STRUCTURE command.

To examine a disk structure, type in

```
STRUCTURE name
```

where name is the logical name of the structure. If the structure contains more than one physical disk, you can access the entire logical structure.

If FILDDT successfully accesses the structure, it enters DDT command level and displays

```
[Looking at file structure name]
```

where name is the logical name of the structure.

9.2.5 Exiting FILDDT

When you are through examining and modifying the target, save the modified file by typing in

<CTRL/E>

FILDDT closes the file, saving any changes that you have made, and returns to FILDDT command level.

Any symbol table that you have loaded (explicitly or by default) remains loaded until you specify another with the LOAD command or the /SYMBOL switch.

If you have modified symbols, FILDDT also modifies the symbol table of the disk file, if one of the following occurred:

- FILDDT automatically loaded the symbol table.
- you loaded the symbol table and entered DDT command level by typing in

  GET file-spec/SYMBOL

To close the file, save all modifications (as with <CTRL/E>, above) and exit from FILDDT, type in

<CTRL/Z>

When you exit FILDDT, you can save FILDDT with its internal symbol table. This saves time if you often use FILDDT to debug a specific file (such as the monitor) that has a very large symbol table.

Start FILDDT, load the symbol table, then exit. Use the TOPS-20 SAVE command to create a copy of FILDDT to be used with that specific file.

For example (your input is in lowercase):

```
@filddt
FILDDT>load system:monitr.exe
[34472 symbols loaded from file]
FILDDT>exit
@save
 FILDDT.EXE.1 Saved
```

CHAPTER 10

PRIVILEGED MODES OF DDT

NOTE

This chapter makes no attempt to explain internal monitor mechanisms. This chapter assumes that you are aware of various monitor contexts. Certain monitor contexts may interfere with or be interfered with by DDT context switching. It is up to you to be aware of these. Note also that internal monitor locations that are used in examples in this chapter are subject to change in subsequent monitor releases.

10.1 MDDT

MDDT is used to debug and patch the running monitor during timesharing, and is an integral part of the swappable monitor. To run MDDT, you must have WHEEL or OPERATOR privileges enabled.

To invoke MDDT, start DDT and then execute the MDDT% JSYS. For example (user input is lowercase):

```
@enable
$ddt
DDT
mddt%<ESC>x
MDDT
```

You can also invoke MDDT by running a MACRO-20 program that executes the MDDT% JSYS.

MDDT runs in the executive virtual address space of the process that executed the MDDT% JSYS. While in MDDT, you are still running in user context, you are running under timesharing, and your process is subject to being swapped out, as is any other user process.

If for some reason you cannot access system files, you can enter MDDT through the MEXEC, as follows:

```
@enable
$<CTRL/E>QUIT
MX>/
```

where the / (slash) command to MEXEC enters MDDT.

To exit MDDT, type in

```
mretn<ESC>g
```

or type in <CTRL/Z>.

10.2 KDDT

You can run KDDT in executive mode or in user mode. KDDT in executive mode is used to debug parts of the monitor which can not be debugged interactively such as those modules that deal with physical memory or paging. KDDT in user mode is used to debug and patch the monitor .EXE file, which you can then save for BOOTing at a later time.

KDDT is part of the resident monitor. When running KDDT in executive mode, you can exercise any normal DDT functions, such as changing memory and setting breakpoints. When you stop at a breakpoint and control passes to KDDT, timesharing (if in effect) ceases.

To run KDDT in executive mode use the /E command when BOOTing the monitor. For example (user input is in lowercase):

```
BOOT>/e                 ;Type in /e
[BOOT: LOADING] [OK]
EDDT                    ;EDDT is loaded and waiting for your command
```

Your debugging may be easier if you lock the swappable monitor in core. You can do this by executing the instruction that calls monitor routine SWPMLK. For example:

```
CALL SWPMLK<ESC>X
```

To run KDDT in user mode:

```
@get system:monitr.exe
@start 140
DDT
```

You can use KDDT in user mode to patch the monitor (.EXE file) which will be booted the next time the system is BOOTed up.

After you have started KDDT as above, use the DDT patching commands to insert your patch. When your patch is complete, exit KDDT with <CTRL/Z> and use the TOPS-20 SAVE command to save the patched version of the monitor. For example:

```
<ESC><                      ;ESCAPE key followed by a left angle
                            ;bracket.
 ....
 ....                       ;Type in the patch.
<ESC>>                      ;ESCAPE key followed by right angle
                            ;bracket.
<CTRL/Z>                    ;Exit KDDT
@save system:monitr.exe     ;Save the new version.
```

10.3 EDDT

You can use EDDT to debug user programs that run in executive mode. You must load EDDT.REL with your program, as follows:

```
@LINK
SYS:EDDT.REL,PROG/GO
@SAVE PROG
```

where PROG is the name of your MACRO-20 program.

CHAPTER 11

PHYSICAL AND VIRTUAL ADDRESSING COMMANDS

11.1 INTRODUCTION

FILDDT, EDDT and KDDT can do their own page mapping. The commands described in this chapter allow you to set parameters to govern the interpretation of the address space that you are examining. You can control the mapping of the address space you are examining by choosing to use or bypass the user process table (UPT) or the executive process table (EPT). You can choose which special pages table (SPT) to use, and which hardware register block to use. Other commands allow you to emulate either KI-paging or KL-paging, control address relocation, and set memory protection limits. In each of the following commands, the argument (page, addr, n) defaults to zero.

Note that the DDT commands <ESC>G, <ESC>P, and <ESC>X have side effects that affect your control over physical and virtual addressing. In addition to their normal functions, these commands also do the following:

- restore normal virtual addressing as if <ESC>U had been given (<ESC>X does NOT do this)
- set the FAKEAC flag (as if <ESC>U had been given)
- clear the relocation factor (as if 0<ESC>8U had been given)
- reset the address-protection address to infinity (377777,,777777)
- restore the active hardware register block to the one in use before any <ESC>4U command was given

11.2 COMMANDS THAT SWITCH BETWEEN PHYSICAL AND VIRTUAL ADDRESSING

Command	Description

<ESC>U

This command enables memory mapping by standard TOPS-20 virtual addressing. When you give this command, DDT restores the virtual addressing conditions that were in effect before any {<ESC>}<ESC>nU (where 0<=n<=2) commands were given, and sets DDT's FAKEAC flag, thereby forcing DDT to interpret memory addresses 0-17 as DDT's own internal "registers", in which the user's registers were saved.

<ESC><ESC>U

This command enables DDT to use actual physical addresses when accessing memory, and clears DDT's FAKEAC flag, causing DDT to interpret memory addresses 0-17 as the hardware registers 0-17. This command is meaningful only when using KDDT in executive mode, or when using FILDDT to look at the running monitor or a core dump. Although DDT accepts <ESC><ESC>U at other times, this command then produces the same effect as <ESC>U.

11.3 COMMANDS THAT CONTROL VIRTUAL ADDRESSING

The general syntax of the following virtual addressing commands is

```
arg<ESC>nU
```

where n is the function number of the command, and arg is dependent on the function (see the function descriptions below).

Functions 0, 1, and 2 enable you to control memory mapping by selecting the executive process table (EPT), user process table (UPT), or the section map through which mapping occurs. Setting a mapping condition with any one of these functions (0, 1, and 2) also has the effect of clearing the effects of any prior use of one of these functions (0, 1, and 2).

You can also specify the offset into the special pages table (SPT) with functions 0, 1, and 2 by using the command

```
arg<ESC><ESC>nU
```

where arg is the SPT offset, and 0<=n<=2. This form is legal only if KL-paging is in effect.

NOTE

All forms of <ESC>B and <ESC>X are illegal if you have used the page mapping functions (0, 1, or 2) and have not restored standard mapping with the <ESC>U command.

Command	Description

page<ESC>0U

This command causes memory mapping to occur through the executive process table (EPT) that is located at physical page page.

offset<ESC><ESC>0U

This command produces the same effect as page<ESC>0U (above), except that offset is an offset (in words) into the SPT.

page1<page2<ESC>0U

This command is an exception to the general syntax, and is legal only under KI-paging. You can select both the user page table (UPT) and the executive page table (EPT) with this command, where page1 is the page number of the UPT, and page2 is the page number of the EPT. Follow page1 with a left angle bracket (<).

page<ESC>1U

This command causes memory mapping to occur through the user process table (UPT) that is located at physical page page. With this command, you can bypass the EPT.

offset<ESC><ESC>1U

This command produces the same effect as page<ESC>1U (above), except that offset is an offset (in words) into the SPT.

page<ESC>2U

This command causes mapping to occur through the section map at physical page page. This command is legal only if KL-paging is in effect.

offset<ESC><ESC>2U

This command produces the same effect as page<ESC>2U (above), except that offset is an offset (in words) into the SPT. This command is legal only if KL-paging is in effect.

n<ESC>3U

This command determines whether DDT interprets references to memory locations 0-17 as references to hardware registers, or to DDT's own internal "registers" (which normally contain the user-program ACs), by setting or resetting DDT's FAKEAC flag.

If n=0, reset FAKEAC flag (use the hardware registers 0-17).
If n is nonzero, set FAKEAC flag (use DDT's internal registers 0-17).

If you type in a nonzero value for n, DDT stores the value -1.

n<ESC>4U

This command tells DDT to copy hardware register block n (0<=n<=7) to its own internal register block, set the FAKEAC flag, and use hardware register block n as its own registers. If the FAKEAC flag is set when you give this command, DDT first restores the contents of its internal register block to the hardware register block from which they were copied. This command is legal in executive mode EDDT and KDDT only. Note that the microcode uses register block 7, and any attempt to use this block produces an almost immediate system crash.

addr<ESC>5U

This command copies the 20 (octal) word block located at addr to DDT's internal "registers" and sets the FAKEAC flag.

addr<ESC>6U

This command sets the special pages table (SPT) to addr.

addr<ESC>7U

This command sets the core status table address (CST) to addr.

addr<ESC>8U

This command sets the address relocation factor to addr. DDT adds addr to all user addresses that you type in.

addr<ESC>9U

This command read-and-write-protects all addresses above addr (before adding relocation factor).

n<ESC>10U

This command controls whether KI paging is enabled or cleared.

If n is nonzero, KI paging is enabled.
If n=0, KI paging is cleared.

If you type in a nonzero value for n, DDT stores the value -1.

n<ESC>11U

This command controls whether KL paging is enabled or cleared.

If n is nonzero, KL paging is enabled.
If n=0, KL paging is cleared.

If you type in a nonzero value for n, DDT stores the value -1.

You can interrogate DDT to determine the last virtual addressing command that was given for a specific function. The command

<ESC>nU

where 0<=n<=11, returns the address of a DDT location that contains the argument that was given if the command for that function was used, and returns the default value if that function was not used. If you typed in a nonzero argument to a command that requires zero or nonzero values (or if the default is nonzero), this location contains -1. You can use DDT commands to examine this location.

The command

<ESC><ESC>nU

where 0<=n<=2, returns the address of a DDT location that contains information that indicates which function you used, and whether you set a page address or an offset. You can use DDT commands to examine this location. This command is illegal for all functions where n>2. If you did not type in any commands affecting functions 0-2 since the last <ESC>U command, the right half of this DDT location word contains zero. Otherwise, the right half contains n+1, where n is the number of the command function you used. If you set a page address (with arg<ESC>nU), bit 1 of this word is zero. If you set an offset (with arg<ESC><ESC>nU), bit 1 of the word is set.

CHAPTER 12

EXTENDED ADDRESSING

12.1 LOADING DDT INTO AN EXTENDED SECTION

If your program is loaded in a nonzero section, merge DDT with your program with the TOPS-20 command

```
@DDT
```

DDT is loaded into the highest-numbered free (nonexistent) section. If your program has a TOPS-10-style entry vector in section zero, the EXEC merges the UDDT stub into the section that contains the entry vector, and places that section's JOBDAT symbol table pointers (.JBSYM and .JBUSY) into the DDT locations pointed to by UDDT locations 770001 and 770002. UDDT then loads XDDT into the highest-numbered free section. You can load DDT into a specific section with the EXEC's /USE-SECTION switch, as:

```
@DDT/USE-SECTION:n
```

where n is the (octal) section number of a section that does not already exist.

You can load DDT into an existing section with the /USE-SECTION and /OVERLAY switches. You must be careful, however, that your program does not use any pages that DDT uses. See the TOPS-20 COMMANDS REFERENCE MANUAL for more information about the use of these switches.

12.2 EXAMINING AND CHANGING MEMORY

The commands /, [(left square bracket),] (right square bracket), !, \ and <TAB> (see Section 4.4.3) open a memory location at an address calculated from an expression typed in or defaulted in the command. The syntax of the command is

```
{expr}{<ESC>{<ESC>}}c
```

where c is the command, and expr is any legal DDT expression (expr defaults to $Q, the current quantity).

In nonzero sections, you can cause DDT to utilize all indirection and indexing indicated by EFIWs (extended format indirect words) to calculate 30-bit global addresses by using the format

```
{expr}<ESC><ESC>c
```

This format also recognizes and utilizes instruction format indirect words (IFIW). These commands are thoroughly described in Chapter 4 (Displaying and Modifying Memory).

12.3 BREAKPOINTS

If DDT is running in a nonzero section, breakpoints can be set in any section.

12.3.1 The Breakpoint Block

To set breakpoints in a section external to the one containing DDT, DDT requires an area of contiguous storage in the section containing the breakpoint. This area is known as the "breakpoint block". The extra storage is required for saving global addresses for transfer of control between your program and DDT, and also for the execution of single-stepped instructions that reference memory locations that are not in their section.

Each section within your program space that contains a breakpoint must have one breakpoint block. Breakpoint blocks are located at the same relative local address within each section (the default is 777700), and are 100 (octal) words in size.

Each breakpoint block is always contiguous within one section. Breakpoint blocks never extend across section boundaries and never "wrap around" the end of a section to the beginning of the section.

DDT creates a breakpoint block in each section as required, if inter-section breakpoints are enabled (see below).

You (or your program) can reference memory within a breakpoint block, but any information stored there can be overwritten by DDT.

12.3.2 Enabling and Disabling Inter-section Breakpoints

The section-relative (18-bit) address of the breakpoint block(s) is stored in an internal DDT location. The command <ESC>4M returns the address of that DDT location. The symbol $4M refers to the DDT location at the address returned by <ESC>4M. Inter-section breakpoints are enabled as long as $4M contains the address of the breakpoint block. At startup, DDT enables inter-section breakpoints by default.

To change the address of the breakpoint block, type in

```
n<ESC>4M
```

where n is the address of the breakpoint block, and can be any legal DDT expression (20<=n<=777700). DDT uses only the right half of n, and changes only the right half of the DDT location at $4M.

By default, the section-relative breakpoint block address is 777700, placing the breakpoint block at the top of the section. To display the address of the breakpoint block, type in

```
<ESC>4M/
```

Inter-section breakpoints are disabled when $4M contains zero.

NOTE

In MDDT, $4M defaults to MDDBLK. In KDDT, $4M defaults to EDDBLK. Each symbol denotes the start of a 100-word (octal) block contained in page zero of the monitor. Page zero of the monitor is mapped into every section that contains monitor code.

To disable inter-section breakpoints, type in

0<ESC>4M

While inter-section breakpoints are disabled, you cannot set a breakpoint in a section external to DDT, and any breakpoints already set in such a section are lost when you begin program execution with <ESC>G, or continue program execution with <ESC>P. For each breakpoint lost, DDT displays

% CAN'T INSERT $nB - IN NON-DDT SECTION

where n is the breakpoint number.

While inter-section breakpoints are disabled, DDT cannot execute the <ESC>X command when:

- you try to execute the instr<ESC>X command, and the default section is not the section that contains DDT, OR
- you try to single-step a dangerous instruction and the user-program PC is not in the section that contains DDT

When this occurs, DDT displays

% CAN'T $X, NO $4M

12.4 DISPLAYING SYMBOLS IN NONZERO SECTIONS

DDT normally uses right-halfword values when searching symbol tables for symbols to display. However, code linked in a nonzero section has symbols defined with the section number in the left-halfword. DDT uses a 30-bit value when searching for a symbol in the following circumstances:

- when displaying the address of a location
- when displaying the contents of a location as an address
- when displaying the Y field of an instruction

When displaying an address, DDT searches for a symbol defined with the 30-bit value of the address. If such a symbol is not found, DDT displays the address in halfword format.

When displaying the Y field of an instruction, DDT searches for a symbol defined with a 30-bit value consisting of:

- the section number of the address of the word being displayed
- the section-relative address contained in the Y field of the instruction

If DDT does not find a symbol defined with that 30-bit value, it looks for a symbol defined with the 18-bit value contained in the Y field of the instruction.

Assume a program with the following conditions:

```
Symbol LABL1 is defined as 0,,300
Symbol LABL2 is defined as 3,,300
Location 1,,300 contains 3,,300
Location 1,,301 contains 2,,300
Location 3,,400 contains 200040,,300
       (MOVE contents of location 300 to AC 1)
```

When displaying the contents of location 1,,300, DDT displays

```
1,,LABL1/   LABL2
```

When displaying the contents of location 1,,301, DDT displays

```
1,,LABL1+1/   2,,LABL1
```

When displaying the contents of location 3,,400, DDT displays

```
LABL2+100/   MOVE 1,LABL2
```

12.5 DEFAULT SECTION NUMBERS

To reduce the need to type in the section number as part of the address when you specify a location, DDT uses a default section number when you do not specify one. DDT has two section defaulting options:

- permanent default section
- floating default section

The command <ESC>6M returns the address of an internal DDT location that contains section default information. The symbol $6M refers to the DDT location at the address returned by the command <ESC>6M.

When DDT is in section zero, the default section number is always zero, regardless of the contents of $6M.

NOTE

When you use KDDT in user-mode, $6M defaults to 0,,0. In all other cases, $6M defaults to 1,,0.

12.5.1 Permanent Default Section

If the value contained in $6M is positive (bit zero is reset), the permanent default section option is in effect. DDT then takes the left half of $6M as the section number of any address that you type in without a section number.

Set the permanent default section by typing in

```
n,,0<ESC>6M
```

where n is the section number, and can be any legal DDT expression.

12.5.2 Floating Default Section

If the value contained in $6M is negative (bit zero is set), the floating default section option is in effect. This is the default option (at start-up, DDT initializes $6M to -1). DDT selects the floating default section as follows:

- If you enter DDT from its normal start address, DDT sets the default section to:
  - the section that contains the program entry vector (if there is one) OR
  - section zero.
- If you enter DDT from a breakpoint, DDT sets the default section to the section that contains the breakpoint.
- If you open a local address between 20 and 777777, DDT sets the default section to the section that contains the open address.
- If you type in an address that contains a section number (including a symbol that is defined with a section number), DDT sets the default section to the one in the address you typed in.

If you exit DDT with <CTRL/C> or <CTRL/Z>, and then reenter DDT, the current location does not change. If you give a command that takes the current location as its default address argument, DDT sets the floating default section to the section of the current location.

In the following example, the DDT screen display is on the left, and explanatory comments are on the right. The entry vector is in section 1. Symbol START is not defined with a section number. User typein is in lowercase.

Screen Display		User Input	Explanation
		3,,place/	Examine location 3,,PLACE.
LABL1			DDT displays the contents.
		<LF>	Type in <LF> to examine the next location.
3,,PLACE+1/	LABL1+2		DDT displays the next location. The floating default section = 3.
		<CTRL/C>	Exit with <CTRL/C>. The current location is 3,,PLACE+1.
@			TOPS-20 prompts you.
		@ddt	Reenter DDT.
DDT			DDT is loaded and ready for your command. The floating default section is 1, because the entry vector is in section 1.
		<LF>	Type in <LF> to examine the next location.
3,,START+2/	LABL1+4		DDT displays the address and contents of the next location. DDT doesn't use the floating default section, because your <LF> command defaults addr to the current location, and uses its section number (3).
		start/	Examine location START. DDT uses the floating default section number because symbol START is defined with no section number.
JFCL 0			DDT displays the contents.
		<LF>	Type in <LF> to examine the next location.
1,,START+1/	MOVE 1,LABL1		DDT displays the address and contents of the location.

12.6 EXECUTING SINGLE INSTRUCTIONS

Instructions that are executed by means of the command

```
instr<ESC>X
```

where instr is the instruction for DDT to execute, are executed within the current default section. If that section is not the one that contains DDT, DDT uses the breakpoint block in that section to execute instr. If the floating default section option is in effect and you are unsure of the current default section, use the addr/ command to open a location in the section in which you wish DDT to execute instr. This sets the default section to the section specified by addr.

If DDT is to execute the instruction in a section other than the one that contains DDT, inter-section breakpoints must be enabled.

If you try to execute instr outside DDT's section while intersection breakpoints are disabled, DDT sounds the terminal buzzer or bell, displays "?" and sets its error string to:

```
CAN'T $X, NO $4M
```

12.7 ENTERING PATCHES IN EXTENDED SECTIONS

You cannot type in a patch if a patching area does not exist in the section that contains the word to be replaced. To ensure that there is a patching area for each section that contains user-program code, do one of the following:

- reserve the same part of each section for patches, and define the patch symbol as 0,,addr, where addr is the local address of the patching area
- use only one patching area and map it into all the sections that contain user-program code. Define the patch symbol as 0,,addr, where addr is the local address of the patching area.
- define a different symbol for each section's patching area, and use the symbol appropriate to the section being patched

If the left half of expr is zero, DDT defaults the section to the section that contains the open location. If the left half of expr is a value that is not the section that contains the open location, DDT displays

```
?CAN'T PATCH ACROSS SECTIONS
```

APPENDIX A

ERROR MESSAGES

DDT and FILDDT have many error messages to indicate when and why they cannot successfully execute your command. DDT sometimes (and FILDDT usually) displays the appropriate message on the terminal, but at other times displays only a question mark. When only a question mark is displayed, a location internal to DDT usually points to a text string that is the error message. To display the DDT error message, type

<ESC>?

You can also display the last TOPS-20 error message for DDT's process. Type

<ESC><ESC>?

To display a specific TOPS-20 error message, type

code<ESC><ESC>?

where code is the TOPS-20 error code.

To display the last process error message for another process, type

fork<<ESC><ESC>?

where fork is the handle of the process. Note that there is a left angle bracket (<) between fork and the first <ESC>. Following is a list of DDT messages together with explanations of what the messages indicate.

? ABOVE PROTECTION REGISTER LIMIT

The address of the location you tried to display or modify is above the protection register limit, which is set by n<ESC>9U.

? ACTUAL REFERENCE FAILED

A memory reference failed unexpectedly (the page exists and is readable, but the reference failed anyway).

? ADDRESS GREATER THAN 777777

An address to be mapped through a section table has a nonzero section number. This can occur only if you specified a section table with the n<ESC>{<ESC>}2U command.

? Bad format for .EXE file

You specified a file that appears to have an .EXE directory, but the directory is badly formatted or DDT cannot read it because of some other reason.

? BAD $4M VALUE

You used the n<ESC>4M command where 777700<n<20.

? BAD POINTER ENCOUNTERED

DDT does not recognize the type code contained in a page map pointer. This can occur only if you are trying to do your own virtual address mapping, and used the expr<ESC>{<ESC>}nU command, where 0<=n<=2.

? CAN'T BE WRITE ENABLED

Even though you have automatic write-enable turned on, DDT is unable to write-enable a page that exists and is write-protected.

? CAN'T CREATE PAGE

DDT attempted to create a page and failed, or else cannot attempt to create the page (see the <ESC>1W command).

? CAN'T DEPOSIT INTO SYMBOL TABLE BECAUSE .....

You tried to define or kill a symbol, but DDT was unable to modify the symbol table. Look up the second part of the error message in this appendix.

? CAN'T DEPOSIT INTO SYMBOL TABLE BECAUSE DEPOSIT FAILED

You tried to define or kill a symbol, but DDT was unable to modify the symbol table, and cannot identify the specific reason.

% CAN'T INSERT $nB BECAUSE .....

DDT is not able to access the location where you inserted your breakpoint. Look up the second part of the error message in this appendix. This situation occurs before DDT tries to execute <ESC>G, <ESC>P, <ESC>X, or <ESC><ESC>X.

% CAN'T INSERT $nB BECAUSE IS IN DIFFERENT SECTION

DDT is not able to access the location where you inserted your breakpoint because inter-section breakpoints are not enabled (<ESC>4M contains zero). This error occurs before DDT tries to execute <ESC>G, <ESC>P, <ESC>X, or <ESC><ESC>X. To enable inter-section breakpoints, deposit the breakpoint block address in the location addressed by the command <ESC>4M.

% CAN'T INSERT $nB BECAUSE MEM REF FAILED

DDT is not able to access the location where you inserted your breakpoint. DDT is not able to identify the reason. This occurs before DDT tries to execute <ESC>G, <ESC>P, <ESC>X, or <ESC><ESC>X. A typical occurrence is when you have a breakpoint set in the swappable monitor (set in KDDT in executive mode), but the swappable monitor is not locked in memory.

? CAN'T PATCH ACROSS SECTIONS

You tried to insert a patch in a section other than the one that contains the patching area.

? CAN'T PEEK PHYSICAL

You attempted to PEEK at the monitor but have specified other than normal virtual addressing (FILDDT only).

% CAN'T REMOVE $nB BECAUSE .....

DDT is not able to access the location where you inserted your breakpoint. Look up the second part of the error message in this appendix. This error occurs when your program enters DDT from a breakpoint.

% CAN'T REMOVE $nB BECAUSE IS IN DIFFERENT SECTION

DDT is not able to access the location where you inserted your breakpoint. because inter-section breakpoints are not enabled (<ESC>4M contains zero). This error occurs when your program enters DDT from a breakpoint. To enable inter-section breakpoints, deposit the breakpoint block address in the location addressed by the command <ESC>4M.

% CAN'T REMOVE $nB BECAUSE MEM REF FAILED

DDT is not able to access the location where you inserted your breakpoint. DDT is not able to identify the reason. This error occurs when your program enters DDT from a breakpoint. A typical occurrence is when you have a breakpoint set in the swappable monitor (set in KDDT in executive mode), but the swappable monitor is not locked in memory.

% CAN'T SET BREAKPOINT, $4M NOT SET

You attempted to set a breakpoint in a section other than the one containing DDT while inter-section breakpoints were not enabled.

? CAN'T $X, NO $4M

Inter-section breakpoints are not enabled, and:

- you tried to execute the command instr<ESC>X but the default section is not the section that contains DDT, OR
- you tried to single-step a dangerous instruction but the user-program PC is not in the section that contains DDT

? Garbage at end-of-command

FILDDT encountered extra text at a place in the command where there should have been only <RET>.

? I/O error

Some kind of I/O error occurred when FILDDT attempted to read or write to the unit specified in a DRIVE or STRUCTURE command.

? Illegal channel number

You entered a DRIVE command that contained an illegal channel number.

? Illegal controller number

You entered a DRIVE command that contained an illegal controller number.

? Illegal unit number

You entered a DRIVE command that contained an illegal unit number.

? Incorrect symbol table pointer

FILDDT is unable to read the symbol table specified by the symbol table pointer in the file.

? Input device must be a disk

The device you specified is not a disk.

? Insufficient memory to read EXE file directory

FILDDT does not have enough free memory to read in the directory section of the .EXE file that you specified.

? Insufficient memory to read PDV list

FILDDT does not have enough free memory to read in the list of PDVs in the .EXE file that you specified.

NOTE

FILDDT sometimes runs out of memory when you use the <CTRL/E> command to save files without exiting FILDDT. If this is the case, exit with the <CTRL/Z> command, and then restart FILDDT with the TOPS-20 FILDDT command.

? INVALID DDT INTERNAL ADDRESS

You addressed an internal location that is not defined. This is most likely to occur after you use a command that returns a value (such as <ESC>M) to examine a DDT location and then use <LF> or <BKSP> to look at nearby memory.

? Invalid file specification, message

where message is a TOPS-20 error string. FILDDT could not parse a filespec given to a LOAD or GET command.

? Invalid guide phrase input

where input is a guide (or noise) phrase that you typed in, and does not match FILDDT's guide phrase.

M

You entered a symbol that is defined in more than one module. You can select the correct symbol by opening the symbol table associated with that module, using the command module<ESC>:.

? MDDT BREAKPOINT BLOCK ALREADY IN USE

Only one fork may have breakpoints set in MDDT at one time. You attempted to set a breakpoint in MDDT while another fork had already set an MDDT breakpoint.

? Missing or extra units in structure

The number of units with the name supplied in a STRUCTURE command does not agree with the number of units in the first structure with that name returned by MSTR%.

? No keyword input

where input is a word that you typed in. You entered ENABLE without the DATA, PATCHING, or THAWED qualifier.

? NO READ ACCESS

You tried to display a word in a page to which you do not have read access.

? No such command as input

where input is a word that you typed in. You entered a command that FILDDT does not recognize.

? No such file structure

COMND% and DEVST% think you supplied a disk name in a STRUCTURE command, but no unit with that name was returned by MSTR%.

? Not enough memory for file pages

FILDDT does not have enough free memory for its file page buffers.

NOTE

FILDDT sometimes runs out of memory when you use the <CTRL/E> command to save files without exiting FILDDT. If this is the case, exit with the <CTRL/Z> command, and then restart FILDDT with the TOPS-20 FILDDT command.

? Not enough memory for symbols

FILDDT does not have enough free memory to read in the symbol table from the specified .EXE file. See the note above.

? NOT IN CORE

You tried to map through a page map pointer (in a UPT, SPT, or section table) that addresses a page that is swapped out. This can occur only if you are trying to do your own virtual address mapping, and used the expr<ESC>{<ESC>}nU command, where 0<=n<=2.

% Not in .EXE format -- Data file assumed.

A GET command without a /DATA switch or a previous ENABLE DATA-FILE command specified a file which is not in .EXE file format. FILDDT assumes it is a data file.

? NOT WRITABLE

You tried to modify a word in a write-protected page. To enable writing on protected pages, use the <ESC>0W command.

? PAGE DOES NOT EXIST

You tried to display a word in a nonexistent page.

? Patching is not enabled

You attempted to modify (with FILDDT) a file, a disk, or the monitor, but did not use the /PATCH switch or the ENABLE PATCHING command.

% Patching the running monitor is illegal

You entered an ENABLE PATCHING command and then gave a PEEK command.

? PEEK FAILED

You tried to PEEK at the monitor, but do not have WHEEL or OPERATOR privileges enabled.

% Symbols cannot be extracted from a data file

You used the command GET filnam/SYMBOL. Either the file specified by filnam is not an .EXE file, or you previously used the command ENABLE DATA-FILE.

? Symbols cannot be extracted from a data file

You used the command LOAD filnam, and the file specified by filnam is not an .EXE file.

U

You entered a symbol that DDT cannot locate in any symbol table. Cure this by entering the correct symbol, or by defining the symbol with the command {expr<}symbol:.

? UNEXPECTED MOVEM FAILURE

DDT could not deposit to memory even though the page exists exists and is write-enabled.

NOTE

In the following messages, unit is one of the following:

- "Unit" (if you used the DRIVE command)
- "Unit u on controller k on channel c" (if you used the STRUCTURE command, where u, k, and c are the arguments you entered)

% Unit has a bad BAT block

The unit that you specified in a DRIVE or STRUCTURE command has a bad BAT block.

% Unit has a bad HOME block

The unit that you specified in a DRIVE or STRUCTURE command has a bad HOME block.

? Unit is off line

The unit that you specified in a DRIVE or STRUCTURE command is off line.

% Unit is write locked

You used an ENABLE PATCHING command and then specified a write-locked unit in a DRIVE or STRUCTURE command.

% Update of file's symbol table failed

FILDDT was unable to write the modified symbol table back to the file after you gave a <CTRL/Z> or <CTRL/E> command. This may also occur when you use the n<ESC>5M command.

GLOSSARY

bit

Bit is a contraction of "binary digit". A bit is the smallest unit of information in a binary system of notation. It is the choice between two possible states, usually designated as zero and one. Bits of data are often used as flags to indicate on/off or yes/no conditions.

breakpoint

A breakpoint is a location in a program's executable code that has been modified so that if the program attempts to execute the instruction at that location, control passes to DDT before the instruction is executed.

breakpoint block

The breakpoint block is a contiguous block of memory required by DDT for utilizing inter-section breakpoints in a program running under extended addressing.

current display mode

The current display mode is the mode in which DDT displays the next word (unless there is an intervening command that changes the current display mode). Also known as the current typeout mode.

current quantity

The current quantity is the most recent of:

- the last 36-bit quantity that DDT displayed
- the 36-bit evaluation of the last expression that you entered as an argument to a command that deposits to memory

This value is often used as the default argument for the next command. Also known as the last value typed.

current location

The current location is a memory word that has been referenced by an earlier DDT command. The address of the current location is the default address for most DDT commands.

current location stack entry

The location that will become the current location as a result of the next <ESC><RET> command.

current radix

The current radix is the radix in which DDT displays numeric values.

current typeout mode

See current display mode.

debugging

Debugging is the process of finding and removing programming errors from programs.

EDDT

EDDT is the DDT variant that is used to debug executive-mode programs.

FILDDT

FILDDT is the DDT variant that is used to examine and modify disk files and disk structures. FILDDT is also used to examine (but not modify) the running monitor.

jiffy

A jiffy is a unit of time defined as one AC (alternating current) cycle. If your line power has a frequency of 60 Hz., a jiffy is one sixtieth of a second (about 16 milliseconds). If your line power has a frequency of 50 Hz., a jiffy is one fiftieth of a second (20 milliseconds).

KDDT

KDDT is the DDT variant used to debug the monitor. You can set breakpoints, single-step instructions, and perform any other DDT function.

last value typed

See current quantity.

location

A location is a numbered or named place in storage or memory where a unit of data or an instruction can be stored. This manual also uses the terms word and memory word.

location counter

The location counter is a memory word that contains the address of the current location.

location sequence stack

The location sequence stack is a stack in which DDT stores the addresses of locations used earlier. DDT uses the stack to access these locations again without having you explicitly enter the address of each of the locations. DDT references these addresses in a last-in, first-out manner.

MDDT

MDDT is the DDT mode used to examine and patch the running monitor.

open location

The open location is a memory word that you can modify with your next DDT command.

prevailing display mode

The prevailing display mode is a user-defined default display mode. DDT displays memory words in the prevailing mode unless you specify a temporary display mode. You can restore the prevailing mode with the <RET> command. See Chapter 4 (Displaying and Modifying Memory) for a list of other commands that restore the prevailing display mode.

reset

Reset refers to the zero condition of a bit or flag. A bit that is zero is said to be reset. To reset is the verb that refers to the act of turning the bit off, "clearing" the bit, or making it zero.

set

Set refers to the nonzero condition of a bit or flag. A bit that is nonzero is said to be set. To set is the verb that refers to the act of turning the bit on, or making it nonzero.

single-stepping

Single-stepping is the process of executing program instructions one at a time using DDT, to verify the result of each instruction.

target

Target refers to the contents of the virtual address space that FILDDT is accessing. The virtual address space may contain a disk structure, a disk file, or the running monitor.

temporary display mode

The temporary display mode is a short-term, user-selected display mode which overrides the prevailing display mode. Temporary display mode remains in effect until you enter <RET> or <TAB>. Also known as the temporary typeout mode.

temporary typeout mode

See temporary display mode.

INDEX

TOPS–20
DDT Manual
AA–M273A–TM

READER'S COMMENTS

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

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

Did you find errors in this manual? If so, specify the error and the page number.

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

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) _____

Name _____ Date _____

Organization _____ Telephone _____

Street _____

City _____ State _____ Zip Code _____
or Country

-- 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

SOFTWARE PUBLICATIONS
200 FOREST STREET MRO1–2/L12
MARLBOROUGH, MA 01752

– Do Not Tear – Fold Here and Tape ----------

Cut Along Dotted Line