DDT XVM UTILITY MANUAL

DEC-XV-UDDTA-A-D

XVM Systems

digital

DDT XVM UTILITY MANUAL

DEC-XV-UDDTA-A-D

digital equipment corporation · maynard. massachusetts

First Printing, December 1975

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

The following are trademarks of Digital Equipment Corporation:

DIGITAL	DECsystem-10	MASSBUS
DEC	DECtape	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EDUSYSTEM	PHA
UNIBUS	FLIP CHIP	RSTS
COMPUTER LABS	FOCAL	RSX
COMTEX	INDAC	TYPESET-8
DDT	LAB-8	TYPESET-10
DECCOMM		TYPESET-11

CONTENTS

TABLES

LIST OF ALL XVM MANUALS

The following is a list of all XVM manuals and their DEC numbers, including the latest version available. Within this manual, other XVM manuals are referenced by title only. Refer to this list for the DEC numbers of these referenced manuals.

BOSS XVM USER'S MANUAL	DEC-XV-OBUAA-A-D
CHAIN XVM/EXECUTE XVM UTILITY MANUAL	DEC-XV-UCHNA-A-D
DDT XVM UTILITY MANUAL	DEC-XV-UDDTA-A-D
EDIT/EDITVP/EDITVT XVM UTILITY MANUAL	DEC-XV-UETUA-A-D
8TRAN XVM UTILITY MANUAL	DEC-XV-UTRNA-A-D
FOCAL XVM LANGUAGE MANUAL	DEC-XV-LFLGA-A-D
FORTRAN IV XVM LANGUAGE MANUAL	DEC-XV-LF4MA-A-D
FORTRAN IV XVM OPERATING ENVIRONMENT MANUAL	DEC-XV-LF4EA-A-D
LINKING LOADER XVM UTILITY MANUAL	DEC-XV-ULLUA-A-D
MAC11 XVM ASSEMBLER LANGUAGE MANUAL	DEC-XV-LMLAA-A-D
MACRO XVM ASSEMBLER LANGUAGE MANUAL	DEC-XV-LMALA-A-D
MTDUMP XVM UTILITY MANUAL	DEC-XV-UMTUA-A-D
PATCH XVM UTILITY MANUAL	DEC-XV-UPUMA-A-D
PIP XVM UTILITY MANUAL	DEC-XV-UPPUA-A-D
SGEN XVM UTILITY MANUAL	DEC-XV-USUTA-A-D
SRCCOM XVM UTILITY MANUAL	DEC-XV-USRCA-A-D
UPDATE XVM UTILITY MANUAL	DEC-XV-UUPDA-A-D
VP15A XVM GRAPHICS SOFTWARE MANUAL	DEC-XV-GVPAA-A-D
VT15 XVM GRAPHICS SOFTWARE MANUAL	DEC-XV-GVTAA-A-D
XVM/DOS KEYBOARD COMMAND GUIDE	DEC-XV-ODKBA-A-D
XVM/DOS READER'S GUIDE AND MASTER INDEX	DEC-XV-ODGIA-A-D
XVM/DOS SYSTEM MANUAL	DEC-XV-ODSAA-A-D
XVM/DOS USERS MANUAL	DEC-XV-ODMAA-A-D
XVM/DOS V1A SYSTEM INSTALLATION GUIDE	DEC-XV-ODSIA-A-D
XVM/RSX SYSTEM MANUAL	DEC-XV-IRSMA-A-D
XVM UNICHANNEL SOFTWARE MANUAL	DEC-XV-XUSMA-A-D

PREFACE

DDT XVM (DDT), which stands for Dynamic Debugging Technique, is an on-line interactive debugging program in the XVM/DOS software system. In the preparation of this manual, it was assumed that the reader is familiar with XVM/DOS e.g., its Monitor and Utility Programs.

Useful manuals to read in conjunction with this one are:

XVM/DOS Users Manual
FORTRAN IV XVM Language Manual
FORTRAN IV XVM Operating Environment Manual
MACRO XVM Assembler Language Manual
Linking Loader XVM Utility Manual

CHAPTER 1
INTRODUCTION

1.1 GENERAL INFORMATION

DDT XVM (Dynamic Debugging Technique) (DDT) is a conversational system program which is available in XVM/DOS. It provides both MACRO and FORTRAN programmers (see Appendix D) with a convenient means for debugging and closely monitoring the operation of their programs. DDT commands entered via the console terminal permit the user to: 1) start a program, 2) suspend its execution at predetermined points, 3) examine the status of memory words, and 4) make additions and corrections using either symbolic or octal code. Under most circumstances the user will be able to stop a "runaway" program. DDT always resides in core with the programs to be debugged, and may be considered as being both a program supervisor and a binary editor. The type of the input information required by DDT or output (printed) from DDT requires the user to be familiar with machine language programming. The format of DDT console terminal input and output is similar to the format used by the MACRO assembler.

1.2 OPERATION

When the appropriate request is typed to the Monitor, DDT is loaded into memory below the bootstrap by the Linking Loader. Upon command the Loader relocates and loads the user's main program and subprograms (including symbol table if requested), all requested user library subroutines, all requested I/O device handlers, and all requested FORTRAN Object Time System routines. After loading has been accomplished, the Loader transfers control to DDT.

DDT uses the Monitor to communicate with I/O device handlers and to trap errors during program execution. While DDT is running, Program Interrupt and Automatic Priority Interrupt are enabled, as well as XVM addressing mode, if requested.

The user converses with DDT via the console terminal. Console terminal I/O is done almost exclusively one character at a time in Image alphanumeric mode. This enables the DDT language to contain simple, concise commands.

1.3 CONVENTIONS AND SPECIAL SYMBOLS

Table 1-1 lists special symbols which are used throughout this manual to represent console terminal Keyboard Operators.

NOTE

In examples simulating TTY entry/response operations, the character or text to be entered by the user is underlined (e.g., xxx) to distinguish it from printed output.

Table 1-1. Symbols Used in Text and Examples

TEXT SYMBOL	TELETYPE[1] ECHO	KEY(S) TO BE ACTUATED	OPERATION INITIATED
)	Non-Print	RETURN	Carriage-Return and Line Feed
␣	Non-Print	SPACE Bar	Carriage is advanced one character space
↓	Non-Print	LINE FEED	Platen is advanced to next line
→|	Non-Print	CTRL and I/TAB	Moves carriage to next tab location (normally 8 spaces)
↑	↑	^ or SHIFT and N/	Control entry defined within programs
↑ plus a character (e.g., ↑T)	Char ↑T	CTRL and Character Key CTRL & T	Initiates a program or system control operation defined within the system
$	$	ALTMODE	Terminator whose use is defined within the system program
[	[	[or SHIFT and K/VT	Use defined within program
]	]	] or SHIFT and M	Use defined within program
\	\	RUBOUT or SHIFT and L/FORM	Character Rubout

[1]Teletype is a registered trademark of the Teletype Corporation.

CHAPTER 2
BASIC DDT

This section introduces the main features of DDT to the beginning user. Many programs can be successfully debugged using only the basic commands described below. More detailed descriptions of the basic commands and other less frequently used commands are given in later sections.

To simplify the information presented in this section it is assumed that only the main program of the user's system is to be tested, even though subprograms and library programs may also have been loaded and linked to the main program. This assumption avoids having to describe, at this point, how symbols in one program are differentiated from identical symbols in another program.

2.1 LOADING DDT AND USER PROGRAMS

In XVM/DOS, the Keyboard command DDT (or DDTNS) calls the Linking Loader as well as DDT (DDTNS prevents loading of the user symbol table in order to save memory).

The first response to the terminal is:

PAGE MODE ON:		BANK MODE ON:
LOADER XVM Vnxnnn >	or	BLOADER XVM Vnxnnn >

Typing a command string to the Loader, e.g.,

>P←MAIN,SUBR1,SUBR2 (Ⓢ)

then loads the user program.

For detailed instructions, read the Linking Loader XVM Utility Manual.

When loading is complete, DDT takes control and types:

PAGE MODE ON:		BANK MODE ON:
DDT XVM Vnxnnn >	or	BDDT XVM Vnxnnn >

to indicate its readiness to accept DDT commands.

NOTE

NS is printed before DDT if the symbol table could not be loaded into available core.

In XVM/DOS, .DAT slots -4 (user programs), -5 (user external library, if any), and -1 (system device for loading of DDT and system library routines) must be assigned to appropriate devices for proper loading.

2.2 EXAMINING STORAGE WORDS

To examine the contents of a core word location, the user must "open" the desired location, receive printout of its contents, and, when finished, "close" the location.

2.2.1 Opening a Location

To open a storage word, the user must type its address terminated immediately by a slash (/).

EXAMPLE: To open location ADR+1, type:

ADR+1/

DDT responds to an "open" entry by performing a tab operation, printing the contents of the addressed location and performing a second tab.

EXAMPLE:

Entry	Response
>ADR+1/	→|LAC␣␣TEMP-5→|

NOTE

The above examples assume that the program's symbol table is available to the user; if not, the user must enter the location address in an OCTAL form.

2.2.2 "Last-Opened-Register Pointer"

Once a location is opened, DDT sets its address into a special pointer register termed the "Last-Opened-Register-Pointer". The pointer is represented in DDT by a period (.); thus when location ADR+1 is opened the pointer (.) is set to its address (i.e., .=ADR+1).

2.2.3 Closing/Reopening Locations

Opened registers are closed by entering a carriage-return (↵).

EXAMPLE:

>ADR+1/ →LAC␣TEMP-5→↵

The last-closed register may be reopened at any time by using the "Last-Opened-Register Pointer" (.). The pointer symbol is typed followed immediately by a slash (/).

EXAMPLE:

>./

DDT responds to a "reopen" entry by reopening the register whose address is stored in the pointer location and printing its contents.

EXAMPLE: If ADR+1 was the last opened register, the reopen procedure would be:

Entry	Response
./	→LAC␣TEMP-5→

2.3 TYPE-OUT MODES

The examples of 2.2 show DDT typing out the contents of a register in symbolic form (i.e., a symbolic instruction with an address field relative to a symbol). This is the type-out mode initially assumed by DDT. All numeric quantities are printed in octal.

The symbolic mode is useful if the user expects the opened register to contain a machine instruction. However, that register might be interpreted instead, as octal data, as a transfer vector[1], or as symbolic text. Additional commands to DDT are available which specify the form

[1]The term transfer vector means a word which contains an address pointing to some other location in memory. Transfer vectors are used with indirect address machine instructions because memory reference instructions cannot directly access more than 4K (in page mode) or 8K (in bank mode) of memory.

in which data is to be interpreted and printed. These mode commands may be typed whenever control is in DDT and it is waiting for typed input.

2.3.1 Address Modes

There are three address mode commands:

Typed Symbols	Meaning
$R	Print addresses in symbolic form relative to a symbol with the closest value, e.g., ADR+1 (Default).
$A	Print addresses as absolute octal numbers (18-bit value).
$F	Print addresses as octal numbers relative to the lowest location in the program. F signifies "floating" addresses, which correspond to the unrelocated values one sees in an assembly listing. These addresses are printed as a pound sign (#), representing the program's load address, a plus sign (+), and an octal constant. For example,

ADR+1/→|LAC #+33→|

2.3.2 Instruction Modes

Four instruction mode commands specify the form in which the contents of opened memory words are printed.

Command	Meaning
$S	Print symbolic instructions; if the instruction is a memory reference instruction, the address field will be printed according to the current address mode (Default).
$V	Interpret words as transfer vectors. Print the 15-bit value as an address in the form dictated by the current address mode. If the higher order three bits are non-zero, precede the address by the six digit octal value of these three bits concatenated with a plus sign.
$O	Print words as octal quantities. The current address mode has no effect.
$T	Interpret data as packed 5/7 ASCII (IOPS ASCII) text. When a register is opened for examination, the contents of both that register and the following register are printed as five ASCII characters.
$$S	Interpret data as SIXBIT Text.
$$R	Interpret data as RAD50 Text.

EXAMPLE:

Assume that registers ADR and ADR+1 contain, respectively, 206613 and 155102 octal.

```
> $S)
> ADR/ →| LAC TEMP      →| $F)
> ADR/ →| LAC #+42      →| $A)
> ADR/ →| LAC 6613      →| $R)
> ADR/ →| LAC TEMP      →| $O)
> ADR/ →| 206613        →| $V)
> ADR/ →| 200000+TEMP   →| $F)
> ADR/ →| #+42          →| $T)
> ADR/ →| !XYZ!         →|
```

Since #+42 is equivalent to the 15-bit value 6613, the relocation factor (#) is 6613-42 = 6551.

2.4 RETYPE COMMANDS

Often, while examining registers in the prevailing type-out mode, the user finds data which should be interpreted in a different mode. DDT permits the user to request that the data be retyped in another form, without changing the setting of the current mode.

There are four retype commands which, if no expression is typed immediately preceding these commands, use as their argument the value of the last expression typed out by DDT.

Retype Command	Meaning
=	Retype the value as an octal number
←	Retype the value as a symbolic instruction
:	Retype the value as a transfer vector
?	Retype the value of the contents of locations . and .+1 interpreted as a 5/7 ASCII text string.
$?	Retype the value in SIXBIT.
$$?	Retype the value in RAD50.

EXAMPLE

Assume that TEMP is absolute location 6613.

Entry	Response	Entry/Response
>$R)		
>$O)		
>ADR/	→226613	→←LAC* TEMP →
	or	
>ADR/	→226613	→:220000+XYZ →
	or	
>ADR/	→226613	→? %XYZ% →
	or	
>ADR-.=000000		

2.5 MODIFYING STORAGE WORDS

Once a word has been opened, its contents may be changed by typing the desired new contents immediately following the type-out produced by DDT. A carriage return terminator commands DDT to make the indicated modification and to "close" the word. For example,

```
ABC-2/  →|JMP   BEG   →|JMP   BEG+2)
```

After being closed, register ABC-2 contains JMP BEG+2. If the register had been closed without typing a modification expression, it would retain its old contents.

NOTE

> DDT does not permit the user to modify all core locations in order to prevent him from inadvertently modifying the Monitor or DDT itself. Should the user attempt to do so, DDT will respond by typing a question mark.

When a user types an expression to DDT, as in the preceding example, the expression should follow the same format as is recognized by the MACRO assembler. However, instruction and address fields cannot be separated by a tab (→|) as allowed in MACRO.

2.6 INPUT MODES

There are no input mode commands. Type-out modes are necessary since DDT cannot guess which form the user considers most appropriate. Where DDT expects an address quantity, user-typed expressions, such as, ADR+3, are evaluated by DDT and taken as 18-bit values. There is no provision for typing in data to be interpreted as 5/7 ASCII text.

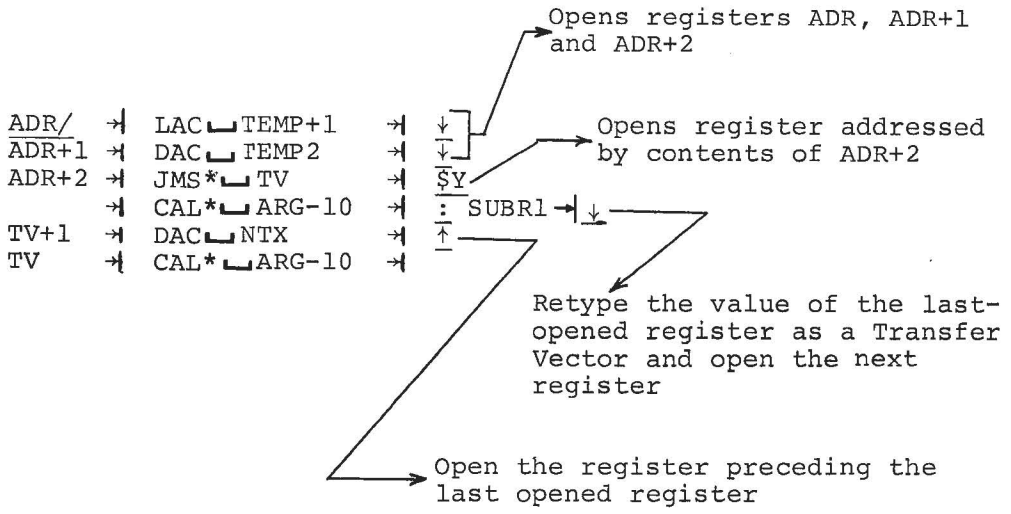
2.7 SEQUENCING

Once a register has been opened and (optionally) modified, it is sometimes convenient to be able to open some other register without having to close the current register and type the address of the new register followed by a slash. DDT contains sequencing commands which (a) modify the open register if an expression was typed, (b) close that register, and (c) open a new register. A few of these sequencing commands are listed below:

Command	Meaning
↓	Opens the next sequential register
↑	Opens the preceding register.
$Y	Opens the register in the same memory bank or page as directly addressed by the address part[1] of the last expression typed in or out.

EXAMPLE:

The following illustrates the usefulness of these commands.



2.8 BREAKPOINTS

One useful function of DDT is its ability to start execution of the user's program and, if one of several pre-set control points in the program is reached, to suspend execution and "break" to DDT control. Such control points are called "breakpoints". Breakpoints are symbolized by the command: $B.

2.8.1 Setting Breakpoints

DDT allows up to four breakpoints, which are entered as numbers 1, 2, 3, and 4.

[1] The indirect bit and the index bit (if in page mode) are ignored.

EXAMPLE: To set a breakpoint at location ADR, the user types

ADR$B (no number assigned) or 2;ADR$B (#2 assigned)

When no breakpoint number is given, DDT chooses, if available, a breakpoint not in use and assigns its number to the entry.

When the command is given to start execution of the user's program, DDT replaces the contents of the assigned break registers with the instruction JMS* 17, where register 17 is an autoindex register which contains a return pointer to DDT. DDT can be made, if necessary, to use some other autoindex register.

2.8.2 Breakpoint Restrictions

Because breakpoint registers are modified so that control can return to DDT, they should not be set on registers whose contents are program-modified or which are used as literals. The user should refrain from placing breakpoints on CAL and XCT instructions until the implications discussed in Section 4 are understood.

2.8.3 Breakpoint Type-Out

When a breakpoint is reached during program execution, control is returned to DDT, which a) restores the original contents of all breakpoint registers, b) prints the number of the breakpoint causing the break, and c) prints the value of the contents of the Accumulator at the time of the break.

DDT automatically saves and restores the contents of all registers when a break occurs and when resuming program execution. The user may examine and modify the saved contents of AC and Link by opening registers A# and L#, respectively, e.g.,

```
>A#/   → 777777 →  0)
>L#/   → 000001 →  0)
```

NOTE

The instruction replaced by the breakpoint instruction is not executed when the break occurs. It is executed only when execution proceeds after a break (see 2.8.5).

2.8.4 Reassigning and Removing Breakpoints

Breakpoints may be reassigned or removed as follows:

a) To reassign a breakpoint, simply type in a new assignment statement using the new address; DDT automatically deletes the previous assignment.

b) To remove all breakpoints, type:

```
0$B
```

c) To remove a specific breakpoint, for instance, breakpoint 3, type:

```
3;0$B
```

2.8.5 Proceeding After a Break

After a break occurs, program execution may be resumed by typing:

```
$P
```

The instruction which is located at the breakpoint is simulated by DDT (except CAL and XCT) and execution proceeds from the register following the breakpoint.

2.9 STARTING A PROGRAM

The command

```
ADR$G
```

will cause DDT to give control to the user's program and to GO to location ADR. If no argument is specified, the command

```
$G
```

will go to the starting address of the program.

2.10 STOPPING A PROGRAM

In spite of the fact that the user may have judiciously set breakpoints in a program, execution may never reach those points if, for instance, the program enters an infinite loop. The user may break out of such a loop and return control to DDT by typing the character CTRL T (↑T).

2.11 ERRORS

If a typing mistake is made by the user and DDT has not taken action, typing either RUBOUT or CTRL U will erase the entire input and allow it to be retyped.

If control is returned to DDT, causing it to print DDT and version number, an illegal action was specified by the program either by having a breakpoint in a routine operating with interrupt off or by obtaining a Monitor error. If an error of this type occurs, the user program must be restarted from the beginning.

If a user enters an undefined symbol as part of a command string, DDT will type the letter U and will ignore the entire string. Whenever the user attempts to perform an illegal command, DDT responds by typing a question mark (?).

CHAPTER 3
DDT LANGUAGE AND SYNTAX

This section describes the rules governing the formation of DDT commands and of expressions, which are used as arguments for these commands.

3.1 COMMAND STRUCTURE

DDT commands take none, one, or two arguments, depending on the specific command. When two arguments are required, they are separated by a semicolon or by an open (left) parenthesis. A command's arguments always precede the command characters. The command may be either a single control character (such as ↑, =, or /), or the character ALTMODE (echoed as $) followed by another character, such as, $A, $T, or $S.

The following examples illustrate the forms which DDT commands may take:

```
arg/            /single argument commands
arg=
=               /no argument
$B              /both args missing [two required for B]
arg$B           /arg 1 missing
arg;$B          /arg 2 missing
arg;arg$B       /both args
```

Note that the $B command is shown four times. It requires two arguments. In the absence of either or both arguments, DDT supplies a predefined value as a default argument for each missing argument. The default arguments used depend on the specific command which they accompany.

If DDT cannot recognize a command string, it will type a ? and ignore the string.

Appendix A contains a detailed list of the commands which are recognized by DDT.

3.2 ARGUMENTS

Arguments to DDT commands are, in general, symbolic expressions which consist of syllables (symbols or numbers) separated by operators.

3.2.1 Syllables

A syllable may consist of one to six characters from the radix 50 octal character set:

a) A thru Z
b) 0 thru 9
c) period (.)
d) % and #

The special characters single and double quote (' and ") may occur anywhere within a symbol. Their significance is explained further on. Syllables are delimited by any non-radix-50 character.

A symbol must contain at least one non-octal character; otherwise, it is taken as an octal number instead of a symbol. Since DDT interprets all numeric input, and outputs all numeric output, in octal radix, the digits 8 and 9 are treated as an extension of the alphabet. In a symbol of more than six characters, all characters beyond the sixth are ignored. In a number of more than six octal digits, the last six digits are retained and those digits preceding the last six are discarded.

The following illustrates symbols and numbers:

ITEM	TYPE	INTERPRETATION
8	A symbol	8 is not an octal digit.
%1.#	A symbol	. % and # are radix 50 characters.
1A	A symbol	Symbols need not begin with a letter.
123456A	A symbol	Although the A is discarded because it is the 7th character, its presence declares the character string to be a symbol.

ITEM	TYPE	INTERPRETATION
123456Z	A symbol	Same symbol as 123456A since only the first six characters are retained.
7	A number	Same as 000007.
1234567	A number	Same as 234567 since the 1 is discarded.

3.2.2 The Symbol Table

In order to evaluate symbols within an expression, DDT must find the symbol and its definition in a symbol table (see memory map in Appendix C). This symbol table has two parts: one part contains the definitions of standard machine instruction mnemonics (e.g., LAC=200000, ADD=300000), the definitions of special DDT symbols (e.g., A# [the saved accumulator], L# [the saved Link], etc.), and any definitions of symbols created by the user in commands to DDT. This part of the symbol table resides in the area of memory occupied by DDT.

The second part of the symbol table resides in lower core and is built there by the Linking Loader. It consists of several groups of internal symbol definitions, one group for each user subprogram loaded. A "header" at the beginning of each group gives the subprogram's file name and its load address (relocation factor). (If the command $DDTNS is given to the Monitor, the Loader will store the headers in the symbol table, but not the internal symbols. This is done to save memory space.)

Subprograms may have internal symbols which are also used as internal symbols in other subprograms. Thus, the same symbol may appear several times in the DDT symbol table, each time with a different value. The following explains how DDT, in the face of multiple-symbol definitions, decides which value to assign to a symbol when it appears in a user-typed expression.

Normally, DDT searches the entire symbol table for the symbol and its 18-bit value and takes the first match it finds. DDT scans the symbol table in the following order: (1) the instruction mnemonics, (2) the special DDT symbols, (3) the user symbols defined at runtime, (4) the symbols for the "current subprogram", and (5) the symbols in all other subprograms.

Initially, the current subprogram is defined as the main program (the first to be loaded - the first to appear in the Loader's command string). The concept is that one normally debugs one subprogram at a time; therefore, it is natural that when the user types a symbol SYM, he is referring to SYM as defined in the subprogram on which he is currently working and not as defined in other subprograms. The user can, by using the header command ($H), define any subprogram as being "current" thus controlling the manner in which a symbol is defined by DDT.

The special character single quote ('), which may appear anywhere within a symbol, declares the symbol to be a file name. The value of such a flagged symbol is the relocation factor for the subprogram with that file name. In such a case, DDT only searches for a header with a matching symbol.

The special character double quote ("), which may appear anywhere within a symbol, declares that symbol to be an address tag. Specifically, it declares the symbol not to be an instruction; thus, DDT will bypass the instruction mnemonic table when searching for the value of such a symbol. This provision is made because it is legal, in the MACRO assembly language, to define address tags which have the same mnemonics as instructions, e.g., →|JMP→|JMP, where the second JMP is an address. So, for example, if the user wishes to examine a register labelled JMP, he types the following to DDT:

```
JMP"/
```

Both ' and " are flags (not to be confused as being symbol constituents) much as the character # in MACRO may be located anywhere within a symbol to declare it to be a variable. Within an expression, any symbol following a space is treated as if it contained the double quote (") flag, that is, treated as an address tag. Thus, for example, in

```
JMP  3+ADR
```

the search for symbol ADR bypasses the instruction mnemonic table.

The special characters ' and " are ignored when they appear within octal numbers.

NOTE

Symbols defined in a MACRO program by the use of = (as in SYM=100) are not passed on to DDT in the user's symbol table.

The user should be aware that FORTRAN IV passes local symbols to DDT which are identical to global symbols. In each case these local addresses contain transfer vectors which are equal in value to the corresponding global symbol. These transfer vectors exist as long lists toward the end of FORTRAN programs and are used in the same way that MACRO uses external global transfer vectors.

The preceding text discussed evaluation of symbols typed by the user. When DDT prints out expressions (for example, the contents of an opened register), it must perform the inverse process of taking a numeric value and converting it into symbolic form, usually involving a symbol table search. The output form is determined by the prevailing type-out modes and is discussed below.

3.2.3 Expressions

Expressions consist of one or more syllables separated by operators and terminated by a character which is neither a legal syllable constituent nor an operator. When calculating the value of a user-typed expression, DDT evaluates the expression from left to right, combining values according to the intervening operators. ASCII text, which can be output by DDT, cannot be input by the user and is, therefore, not considered as a form of input expression. An expression may, of course, consist of only one syllable.

DDT assumes, when evaluating an input expression, that the expression is preceded by 0+. This implies that the value of the first argument in a two-argument command is zero when the argument separator is typed with no preceding argument, e.g.,

;ADR$B is equivalent to 0;ADR$B

If the argument separator is missing, DDT supplies a default argument.

The following operators define the ways syllable values may be combined.

OPERATOR	MEANING
+	Add the two values in two's complement arithmetic (overflow is ignored)
␣	Add the two values in two's complement; but, from now until the end of the expression, do not change the instruction part of the accumulated value (bits 0 through 5).
*	Add the two values in two's complement and "exclusive or" the result with 20000 (the indirect bit). (Overflow is ignored).
-	Negate the following value if this is a unary minus. Otherwise, subtract the following value from the preceding value in two's complement. (Overflow is ignored.)
&	Form the "AND" of the two values bit by corresponding bit.
\	Form the "exclusive or" of the two values bit by corresponding bit.
!	Form the "inclusive or" of the two values bit by corresponding bit.

The following subheadings describe input and output expression according to the various type-out modes. Note, however, that the prevailing type-out modes have no bearing on user-typed expressions.

3.2.4 Symbolic Instruction Mode

To obtain the relocated value of any instruction in the current program as loaded by DDT, the user may type an expression identical in most respects with the original MACRO instruction. If all the symbols are defined in the original program, new instructions may be formed. Tab, however, may not be used to separate OP code from address field (space is used exclusively). The following are legal instructions, provided that the addresses are defined:

1. JMP␣JMP (JMP is an address in the current program)
2. DAC␣1+AD (Although AD evaluates to 18 bits, only 13 bits are used because of the space)
3. CLA!CLL
4. EAE+1002 (DDT does not contain EAE mnemonics)
5. IOT+314 (DDT does not contain IOT mnemonics)
6. LAW␣-1 (Same as LAW 17777)

7. XCT*␣AD

8. LAC ␣ C,X Indexed instruction

9. CAL␣775

10. AAC␣-1 (Once AAC, AXS, or AXR is encountered, the remainder of the expression will not alter the value of bit 0 through 8.)

On output if DDT is in symbolic mode, the following procedure is used to output an expression representing an 18 bit computer word: the instructions are broken down into categories appropriate to their OP codes.

1. If the OP code (bits 0-3) is 00 (CAL), the instruction is output as an octal number with leading 0 suppression (e.g., 775).

2. If the OP code is 74 and bit 4 is off, the instruction is output as an inclusive ORed microcoded operate instruction (740000 types out as NOP, 754000 as CLA!CLL).

3. If the OP code is 76 (74 and bit 4 is on), the instruction is output as LAW ␣N, where N is an octal number representing the two's complement of the instruction (e.g., LAW␣-1 for 777777).

4. If the OP code is 64, the instruction is output as EAE+N, where N is an octal number representing bits 4-17 of the instruction (e.g., EAE + 1002 for LACQ). If the DDT symbol table contains an exact match, that match will be output.

5. OP code 70 must be broken down further (an exact match in the DDT symbol table will be output as in 4). If the instruction is an AAC, AXS, or AXR instruction, the corresponding mnemonic followed by space would be output, followed by bits 9-17 as a signed octal number. Otherwise, the number would be output as IOT+N, where N is an octal number representing bits 4-17 of the instruction, (e.g., IOT+314 for IORS and AXS␣1 for 725ØØ1).

6. The remaining OP codes are memory reference instructions. The corresponding instruction mnemonic is output. If the indirect bit is set, a * is output following the mnemonic. Then a space is typed, followed by the address referred to directly by the instruction, typed in the current address mode. The bank bits for the address are the same as the bank bits of "point" (the address of the last opened register), unless an auto-index register is referred to indirectly (e.g., LAC*␣1Ø). If the index bit is set, the address is followed by ,X. The following are examples of memory reference instructions printed by DDT:

1. JMP␣JMP (JMP is an address in the current program)
2. DAC␣AD+1 (Relative address mode)
3. XCT*␣AD
4. LAC␣AB,X (Indexed address)
5. DAC␣21253 (Last opened register is in bank 1; absolute address mode; same as Ø41253)
6. DAC␣#+2 (Floating address mode; address relative to beginning of current program [#])

3.2.5 Octal Number Mode

The conventions for numerical input were explained in 3.2.1. On output, octal numbers without leading-zero suppression are used to represent instructions in octal mode, (e.g., Ø6ØØ1Ø for DAC*␣1Ø).

3.2.6 Transfer Vector Mode

In this mode, bits Ø-2 of the word to be output are ignored if they are zero. If bits Ø-2 are non-zero, they are printed as an 18-bit octal number preceding the address, e.g.,

LABEL/ →|3ØØØØØ+ALPHA →| .

If the address mode is relative, the symbol table is searched according to the current header for the address which is nearest in value to the value of bits 3-17 of the word to be output. If its value is within 1ØØ octal of the word to be output, this address symbol is typed followed by either + or - and the difference as an octal number. Since the same symbol in the symbol table may represent more than one address, care must be taken on input of addresses. Since transfer vectors in FORTRAN are defined locally and globally with the same symbol, the following seemingly legal attempt to restore the value of a location fails:

TAN/→| TAN →| TAN↵

The contents of local symbol TAN now contains, not the global address of the beginning of the tangent routine, but the address of the transfer vector TAN (its own address). On input, local symbols of the current subprogram always take precedence over globals. In absolute address mode, the 18-bit address is output in octal.

In floating address mode, the address is output as relative to the symbol #, which represents the first address of the subprogram loaded into the area of memory containing "point" (the last opened register). In symbolic address mode, the address may be printed as in floating mode, if a symbol within 1ØØ octal of the address is not found in the symbol table. Likewise, an address to be printed in floating mode will be printed in absolute mode, if it is less than the current value of #.

3.2.7 ASCII Text (Output Only)

The locations . and .+1 are interpreted as ASCII 5/7 packed data if typed in this mode (e.g., ABCDE is output for 4Ø6Ø5Ø, 342212).

CHAPTER 4
DEBUGGING WITH DDT

All DDT commands and features are described in this section. References to earlier sections are made to avoid redundancy.

4.1 LOADING A PROGRAM

See SECTION 2.1.

4.2 STARTING A PROGRAM

When the Linking Loader has loaded DDT and the user's programs and gives control to DDT, DDT sets the value of the special symbol SA# (start address) equal to the starting address of the main program.

To command DDT to start the user's program at a given location, type:

K$G

where K is an expression whose 15-bit value is the address to which DDT will Go. In the absence of K, the default argument SA# (start address) is used. To first change the value of SA# and then go there, type:

;K$G

If K is missing, (i.e., ;$G), SA# will not change. The user may enter his program any place between P#' and C#. (See Section 4.3.2E .) Any other address will be rejected and DDT will type a question mark.

The following two points should be remembered when reading the subsequent section on the "proceed count" and "proceeding after breakpoint":

1. The $G command always sets the proceed count to 1.
2. The $P (proceed) command will also start the program when there has been no breakpoint break from which to proceed.

If a started program seems to be malfunctioning and the user wants to go back to DDT, then typing ↑T (CTRL T) will cause the Monitor to give control to DDT, which will give the go ahead signal (>).

4.3 REGISTER EXAMINATION AND MODIFICATION

4.3.1 Type-Out Mode Commands

The type-out mode commands (see Section 2.3 and also Appendix A, Sections B and C) dictate the form in which DDT will print instructions and addresses, unless specifically overridden by the command being executed. When first loaded, DDT assumes $S (symbolic instruction mode) and $R (relative symbolic address mode). The commands to set the type-out modes do not take arguments; if arguments are supplied, they will be ignored. If a supplied argument contains an undefined symbol, DDT will type:

```
U  →| >
```

to indicate that the preceding symbol is undefined and that the command was ignored. The undefined symbol is encountered, in this case, before DDT has a chance to determine that the command requires no argument. This is because DDT does not buffer the command text but, instead, interprets it as it comes in, character by character. This pertains to all command input to DDT.

When the type-out mode command has been accepted and executed, DDT will respond by performing a carriage-return, line-feed and typing:

```
>
```

Whenever DDT is expecting command input, the type-out modes may be changed without affecting other DDT operations, e.g., an open register remains open.

4.3.2 Special Symbols and Concepts

Register examination and modification requires the user to carefully coordinate certain basic concepts.

A. The single character "pointer" (.) usually represents the address of the last register to have been opened. "Pointer" is often used as a default argument to DDT commands. Under the following conditions, the value of "pointer" will be modified as indicated:

1. Immediately after a breakpoint is reached and control is returned to DDT, the value of "pointer" will be the address of the register whose contents are typed out after the break (a feature to be discussed later).

2. After ↑T (CTRL T) has been typed and control has returned to DDT, the value of "pointer" is the address of the instruction that was to have been executed. The user may determine at what location program execution was suspended simply by typing:

   .= or .:

   which are two forms of retype commands. The user may restart the program at the pointer where it was interrupted by typing:

   $P

   provided that the program was not interrupted while control was in the Monitor. Device handlers are not loaded with the user program units or DDT.

3. When DDT is started or restarted it prints:

```
DDT XVM Vnxnnn
>
```

   At this time the value of "pointer" is set from the value of the starting address, SA#, which is initially the starting address of the main program.

4. After one of the search commands (discussed below) has been executed, the value of "pointer" is set to the address of the last register typed out during the search. If no register was typed out, the value of "pointer" is not changed.

B. The "current location", which is represented by the symbol . in MACRO programs, differs at times from the value of "pointer" used by DDT. The current location does not have a special symbol associated with it. When a register is opened for examination by any command, except $Z, the current location and the value of "pointer" are the address just opened. The values of "pointer" and the current location will differ when the sequencing command $Z is used (see below).

C. The areas of memory occupied by each user subprogram, including the main program, are defined by the "header" file names which appear in the symbol table built by the Loader and which contain the relocation factors (load addresses) of each subprogram. The free memory space which exists immediately above the Loader-built

symbol table (see memory map in Appendix C) is treated as a pseudo-program to which DDT has assigned the file name P#' (recall that the ' declares the symbol to be a file name). This symbol represents the first location (relocation factor) of the patch file area (free core). This area is used to insert additional code at run-time.

D. The special symbol # represents the first location of the sub-program which occupies the area of memory containing the address "point". The one exception, which does not change the value of #, is when "point" has been modified following the execution of a search command. The sub-program containing the address "point" will be referred to as the current sub-program. The implications of this are best explained by example, using some of the basic commands described in Section 2. Each line in the example is numbered and the comments are keyed to these line numbers:

Example

```
1       >$F↵
2       >PRGA'=003000          → ↵
3       >PRGB'=004000          → ↵
4       >PRGB'/  → LAC   #+100 → #=004000  → ↵
5       >.=004000→ .:BEGIN     → ↵
6       >PRGA'/  → DAC   #+3   → #=003000  → ↵
7       >.=003000→ .:START     → ↵
8       >BEGIN/  → LAC   #+100 → #=004000  → ↵
9       >.=004000→
```

In line 1 the command sets the address mode to floating (#+ number).

In line 2, the value of the file name PRGA' is 3000 (program PRGA starts at location 3000).

In line 3, the value of PRGB' is 4000 (program PRGB starts at location 4000).

On line 4, the register whose address is the same as the value of PRGB' (register 4000) is opened and its contents are printed as a symbolic instruction with a floating address (#+100). The value of # and . at this point is 4000, which means that the current program is PRGB and that its relocation factor (#) is 4000. Therefore, the LAC instruction, if executed, would load the accumulator with the contents of location 4100.

On line 5, . is retyped as the symbol BEGIN. Therefore, PRGB has a local symbol called BEGIN which refers to the first location in that program. Assume for the moment that PRGA and PRGB do not use any identical symbols.

On line 6, PRGA' is opened. This changes the value of . to the value of PRGA' and makes PRGA' the current program. Since . is now located in PRGA and no longer in PRGB, the value of # is changed to 3000, the relocation factor of PRGA.

On line 7, . is retyped as the symbol START, a local symbol in PRGA.

On line 8, the register BEGIN is opened. If BEGIN had been defined locally in PRGA, the value of # would not have changed (since . would still have been located in PRGA). Since BEGIN is located in PRGB, PRGB is made the current program and # is changed accordingly.

When searching the symbol table for the value of a symbol, DDT searches the local symbols of the current program before those of other programs and uses the first match it finds.

E. DDT will not allow core outside a specified range to be modified by command[1]. The area that may be modified includes the auto-index registers, the core area between the symbols P#' and C# and the core area between TP# and the top of core. TP# is the address of the first location above the bootstrap. P#' is the address of the lowest register in the patch area. C# is an address within DDT which contains the two's complement value of the "proceed count". Immediately below C# are other special DDT addresses, such as, A# (the address of the saved AC). All the special DDT addresses preceding and including C# may be modified by command to DDT. The core between the patch area and these special DDT registers is where the user's programs are loaded. A breakpoint may be placed anywhere in a user program or in the patch area, but is may not be placed at an auto-index register nor within DDT. In systems with XVM mode enabled, uninitialized COMMON blocks are loaded starting at TP# and going up in memory, if there is room.

F. If a register is opened which is below the patch area, the current program will not change since no file name exists for registers below P#'. The user may, however, define a file name with a value less than P#' for the purpose of examining lower core.

G. The special symbol Q#, which is often used as a default argument to DDT commands, assumes the value of the contents of the most recently opened register or the value of the argument to the last intervening retype command.

[1] See section 4.7.7 for exceptions.

Example:

```
>LOC/ →| 123456 →| Q#= 123456 →| Q#+1↵
>./   →| 123457 →|
```

Register LOC is opened and contains 123456. Note that at this point the value of Q# is 123456. Then the contents of register LOC is increased by +1, utilizing Q# to represent its current contents.

4.3.3 Register Examination Commands

The basic register examination command is /. Typing the command K/, where K is an expression, will open register K and print its contents in the current modes. The default argument is "point" (i.e., / is equivalent to ./).

If / is replaced by [, printout is temporarily forced into octal mode. (If [is preceded by ALTMODE ($), then octal becomes the permanent instruction mode.)

If / is replaced by], printout is temporarily forced into symbolic mode. (If] is preceded by ALTMODE, then symbolic becomes the permanent mode.)

If / is replaced by <, printing of register contents is omitted until the next occurrence of a carriage return command. (This mode, called "type-in mode", allows for rapid code insertion.)

If any register examination command is followed by ?↵, an attempt was made to open a nonexistent register.

Examples:

```
>ADR-1/  →| JMP* SUBR →| ↵ (print in current modes)
>.[      →| 623115   →| ↵ (force octal)
>./      →| JMP* SUBR →| ↵ (modes haven't changed)
>.$[     →| 623115   →| ↵ (permanent mode change)
>./      →| 623115   →| ↵ (mode now octal)
>.]      →| JMP* SUBR →| ↵ (force symbolic)
>.<      →| ↓          (contents are not printed out until
  ADR    →| ↓          ↵ encountered)
  ADR+1  →| ↵
>
```

4.3.4 Expression Retype Commands

Alternative representations of the contents of a register or of an expression may be obtained by using retype commands (see Section 2.4).

Where K is an expression,

- K= means retype K in octal
- K← means retype K in symbolic
- K: means retype K as a transfer vector
- K? means retype the contents of "point" and the following register as 5/7 ASCII text (K unused).

In each case above, the value of Q# is set to the value of K. The default argument, if K is missing, is the current value of Q#.

Retype commands force a temporary mode change; they do not alter the setting of the type-out modes.

4.3.5 Register Modification Commands

The basic register closing and modifying command is carriage return (↵). If K↵ is typed when a register is open, then the value of K (expression) becomes the contents of the open register and the register is closed (K becomes the value of Q#). If K is omitted, the register is closed without modification. If no register is open, none is modified and K becomes the value of Q#. Upon completion, DDT will type >. The command ↵ also turns off type-in mode, the temporary mode which allows the user to insert code without having to wait for DDT to print the contents of each register.

In addition to ↵ there are "sequencing commands", which behave like ↵ in that, if given an argument K, K replaces the old contents of the open register (assuming one is open) and then that register is closed. Then a new register is opened. For every command listed below, with the exception of $Z, the newly opened register becomes the current register. The significance of this will be explained in the example.

- K↓ (Line Feed). Modify the open register, close it and then open the next register in sequence.
- K↑ Modify the open register, close it and then open the register preceding the one just closed.

K$Y — Modify the open register, close it and then open the register whose address is contained in the address part of K. K is assumed to be a memory reference instruction. The address part (12 or 13 bits, page or bank mode, respectively) is taken, and that address (in the same memory page or bank as the now closed register) is opened. If K is missing from the command, the address part of the contents of the closed register is taken. In other words, the argument to $Y is always Q#.

K$Z — This command is identical to $Y with the exception that the newly opened register is not made current (the current register and the value of . now differ). $Z is useful for examining literals without breaking program sequence; that is, subsequent use of ↓ or ↑ will refer to the current register, not the one opened by $Z.

K$J — This command is similar to $Y. It modifies the open register, closes it and then, treating K as a 15-bit transfer vector, opens that address.

If DDT types a question mark after any of these commands, either the command was not properly received (unlikely) or an attempt was made to alter a location outside the allowable range (autoindex registers and P#' through C# and TP# through the top of core)[1].
In either case, the command is ignored.

Example

```
>LOC/     → LAW - 1     → LAC LIT12$Z
          → 12          → ↓
 LOC+1    → DAC TMP     → ↓
 LOC+2    → JMS* TV     → $Y
          → DAC ARG-77  → :   SUBR1 → $J
          → 0           → ↓
 SUBR1+1  → CLL!CLA     → ↓
 SUBR1+2  → SAD XYZ-2   → ↑
>SUBR1+1  → CLL!CLA     → STL!CLA )
```

Note that the addresses opened by the commands $Y, $Z and $J are not printed by DDT. Note that $Y after JMS* TV opened register TV and printed its contents as if it were an instruction. This did not look right, so : was typed to request reprint as a transfer vector.

The use of $Y or $Z following indexed and indirect instructions, such as,

```
LAC* TAB,X$Y
```

[1] See Section 4.7.7.

will not perform the indexing nor indirection to determine the address of the next register to be examined. In this example, register TAB would be opened.

If the user wishes to effectively insert code in his program, he must, in general, modify two registers in his program: one to JMP* through a transfer vector and the other to the transfer vector point to patch space in free core (which might not be in the same core bank or page as the user's program). Patch space begins at the register P#' and goes as high as the address stored in register 103 (.SCOM+3). The user should be aware that making patches in locations higher than the value in 103 will overlay the user's program.

Besides the method of opening and modifying registers there are special instructions available for initializing memory between limits to a constant and for updating the values of special DDT registers. These commands will be discussed under later headings.

4.4 DEFINING A SYMBOL

In addition to the symbols passed on to DDT by the Linking Loader, the user has the option of defining additional symbols at run time. These "DDT-time" symbols are added to the end of DDT's symbol table (see Appendix C).

E(S) — The symbol S, which must be a unique symbol, is given the value of the expression E. (S) is equivalent to 0(S).

S) — The symbol S is given the value of "point".

Only these "DDT-time" symbols may be redefined. Symbols may be deleted as follows:

$K — (no argument) will Kill (delete) all DDT-time symbols.

0$K — will Kill all the user's load-time symbols. This would only be done, presumably, to increase the available patch space (P#' will be suitably redefined).

If the user attempts to define any symbol which is not a DDT-time symbol, DDT will ignore the command and type an X. If he attempts to define a symbol when the DDT-time symbol table is full, DDT will ignore the command and type an O (Overflow).

4.5 SEARCH OPERATIONS

There are three search commands, each of which searches inclusively between a lower core limit and an upper core limit for words or parts of words which have or do not have a specified value. The search commands use as arguments the values in three special DDT registers:

M#	search mask
LO#	lower limit search address
HI#	upper limit search address

Each of these registers may be individually modified, e.g.,

```
>M#/  →|  777777  →|  770000↓
 LO#  →|  012252  →|  14000 ↓
 R#   →|  025611  →|  14400 ↵
>
```

Notice that these registers appear sequentially in memory. (The symbol R# is equal to HI# and will be printed instead of HI#.)

The contents of LO# and HI# may be modified directly by command:

A;B$L

where A and B are expressions. This sets the value in LO# to A and the value in HI# to B. The default arguments are P#' (beginning of free core patch area) and L# (the address of the saved value of the Link within DDT). A# (the saved AC) is the first register in DDT and L# is the second. LO# is initially set to P#' and HI# to L#. The initial contents of the mask, M#, is 777777 (all ones).

The first command is the Word Search:

M;K$W

searches through every core register between the limits set in registers LO# and HI#, inclusively, for words whose values match the value of K (an expression) in those bit positions specified by 1's in the corresponding bits in the mask argument M.

For example, to search for all words between locations 100 and 200, equal in value to JMP BEG, type:

```
100;200$L
777777;JMP BEG$W
```

Of course, if LO# already contains 100, HI# contains 200, and M# contains 777777, the user need only type:

```
JMP BEG$W
```

Specifying a mask in the $W command does not cause the contents of M# to be modified.

To search for all words between locations 100 and 200 whose Op code parts contain JMP, type:

```
100;200$L
740000;JMP$W
```

Note that the mask is set so that only the high order 4 bits (the Op-code bits) are tested for a match.

To print out the contents of registers 100 through 120, type:

```
100;120$L
0;$W
```

Zero is used as a mask so that there is no possibility of a mismatch. The second argument is immaterial and was, therefore, not given.

The default arguments for M;K$W are the contents of M# (if M missing) and 0 (if K missing).

The second command is the Not Word Search:

```
M;K$N
```

It is identical to $W except that the search is for words which differ from rather than match K in the masked bit position. For example, to print all registers between locations 100 and 200 whose contents are non-0, type:

```
100;200$L
$N
```

(Note that 0 is the default value of K. It is assumed, in this example, that M# contains 777777).

The third command is the Effective Address Search:

```
M;K$E
```

The default arguments for M and K are, as above, the contents of M# and 0, respectively. $E searches core between the limits specified by LO# and HI# for all memory reference instructions which directly or indirectly reference the address K (an expression), testing for a match only in those bit positions specified by 1's in the mask. (Bits 0-2 are always disregarded.)

The following example gives a good indication of the usefulness of this command. It is desired to know all the registers between locations 1000 and 2000 which reference auto-index registers:

```
1000;2000$L
777770;10$E
```

To search for all references to ADR+1, provided that the mask and limits contain the desired values, one need only type:

```
ADR+1$E
```

The contents of the hardware index register are saved in and restored from XR#, and an effective address search which encounters indexed instructions will use the value in XR# as the index.

During any of the three search operations whenever a condition is met (match or no match), DDT prints the octal address of the memory word which satisfies the search constraints, prints a tab, prints the contents of that memory location in the prevailing type-out modes and types a carriage return. The search then continues until the upper limit is exceeded. At the end of the search, the value of "point" is set to the address of the last register printed in the search map (the current register remains unchanged). "Point" remains unchanged if no printout occurred.

4.6 BREAKPOINTS

4.6.1 Definition

A "breakpoint" is a pre-selected point in a program where the flow of the program is broken to allow the user to perform DDT functions.

Whenever it is about to give control to the user's program, DDT saves the instruction at each breakpoint and replaces it with a JMS* 17 instruction.[1] DDT also stores a return pointer to itself in auto-index register 17. Thus, whenever a breakpoint is reached, control is transferred to DDT to allow the user to examine and alter registers, search, etc. When the user signals DDT to continue execution of his program ($P), the instruction that was originally at the breakpoint location is simulated and then DDT transfers control to the register following the breakpoint.

The user's program must not modify the auto-index register that DDT uses for breakpoint returns.

Up to four breakpoints may be set to facilitate debugging when there is uncertainty as to which path a program will follow.

4.6.2 Setting Breakpoints

Initially, all four breakpoints are cleared (unassigned). The general form of the breakpoint command is:

N;A$B

where N is a breakpoint number (1 to 4) and A is an expression evaluated to a 15-bit address. This causes DDT to assign (set) breakpoint N at location A (provided that the value of A is non-0; see below). For example, to set breakpoint 2 at location ADR, type:

2;ADR$B

It is possible to reset a breakpoint to some other address without first deassigning that breakpoint, e.g., if breakpoint 2 had been set at location XYZ, the preceding command (2;ADR$B) would supersede the earlier (XYZ) assignment.

---

[1] A command exists to tell DDT to use some other auto-index register for breakpoints (discussed later).

If the argument A is missing, the default argument is "point". For example:

ADR/ →| LAC TMP →| 3;$B

This sets breakpoint 3 at location ADR because "point" has the value of register ADR. (3;$B is equivalent to 3;.$B.)

If the user does not care which breakpoint number is used when assigning a breakpoint, he can, simply by leaving out the first argument N, request DDT to assign an unused breakpoint number. For instance, assuming all four breakpoints are available, to set a breakpoint at location X, type:

X$B

DDT, finding breakpoint 1 unused, then types a 1 to indicate which number it selected, i.e.,

X$B1

If all four breakpoints are already in use, DDT types a question mark:

X$B?

and ignores the command.

If register X is outside the legal range of registers which may have breakpoints (P#' to C#),[1] DDT types:

X$B1?

indicating breakpoint 1 is free but X is illegal, and then ignores the command.

In addition to assigning and reassigning breakpoints, the user may remove (clear) them. This is signified by a value of 0 for the argument A.

The input

N;0$B

removes breakpoint N (1 to 4). If N is absent, the input

0$B

removes all breakpoints.

---

[1] See Section 4.7.7.

When a breakpoint is set at a given location, the contents of that location are not changed, e.g.,

```
>X/ →| LAC  TMP →| $B
>X/ →| LAC  TMP
```

The swapping of the contents of X with a JMS* 17 occurs only when DDT gives control to the user program.

4.6.3 Breakpoint Restrictions

It has already been mentioned that DDT will not allow breakpoints to be set at locations outside the range P#' to C#.[1]

The user must not set a breakpoint at locations containing:

a) instructions which are program modified

b) instructions which are used as constants (operands of other instructions).

This is because the actual instruction in a breakpoint location is changed by DDT, prior to program execution, to a JMS* 17 (or some other autoindex register).

If a breakpoint is set on a CAL or XCT instruction and a break at such a location occurs, DDT remembers this fact. If the command is then given to proceed with program execution where it left off, the breakpoint is removed and replaced by the original CAL or XCT. This is done because DDT cannot simulate the CAL (which has a variable number of arguments following it) out of place. The XCT cannot be simulated out of place since XCT can execute a CAL instruction. However, should control return to DDT via some other breakpoint, the breakpoint on the CAL or XCT will then be reinstated.

Should the user wish to place a breakpoint on a CAL instruction, the following practice will always ensure that the breakpoint remains set on the CAL.

Example:

```
CAL 3          (set breakpoint 1)
12
LAC BUFF       (set breakpoint 2)
```

[1] See Section 4.7.7.

The CAL used above is the macro expansion of the Monitor call:

```
.WAIT 3
```

Placing a second breakpoint on the register immediately following the CAL and its arguments (the return point) ensures that the breakpoint at the CAL will always be reinstated. An additional feature in the breakpoint logic, which is explained later, allows the user to specify breakpoint 2 (above) so that it never causes a break (breaks and then immediately continues). This makes breakpoint 2 "transparent" or "invisible".

4.6.4 Flow of Control at Breakpoints

When execution of the user's program reaches a breakpoint, control is returned to DDT. At this point, DDT executes what is called a "conditional break instruction." If this instruction does not cause a skip (e.g., NOP), which is the usual case, then DDT decrements the "proceed count". If this causes the proceed count to be equal to zero, a break occurs. If not, DDT will simulate the instruction which was replaced at the breakpoint (by a JMS* 17) and then return control to the user's program at the address following the breakpoint register. In other words, the break does not take place.

There is only one proceed count for all four breakpoints. As the name suggests, the proceed count is a value specified by the user indicating how many times breakpoints are to be reached before a break is to occur. It is particularly suited for specifying the number of times a loop should be executed. The proceed count will be discussed later on.

If the conditional break instruction causes one register to be skipped (e.g., SKP or SPA with the AC positive) the program continues without a break and without decrementing the proceed count. This is the "invisible" breakpoint discussed in section 4.6.3.

If the conditional break instruction causes two registers to be skipped (explained below) DDT will always break, without altering the proceed count.

Each breakpoint N has its own conditional break instruction, which the user may examine and modify directly by opening register I#+N (N = 1 to 4). Initially, when a breakpoint is set with the $B com-

mand, the contents of the associated conditional break instruction register, I#+N, is set to NOP. NOP does not skip, so the break will be determined by decrementing the proceed count. After the $B command, one may modify I#+N to contain any instruction one likes (e.g., CLA!SNL or LAC* 10 or JMS* V#+N). The first of these will decrement the proceed count only when the Link is zero and continue otherwise. The second always decrements the proceed count but, in addition, effectively inserts one instruction prior to the breakpoint.

The third allows the user to call one of his own subroutines when the breakpoint is reached. The subroutine, when it returns, can decide whether to skip two, skip one, or not skip, simply by incrementing its entry point twice, once, or not at all. V#+N, shown in the last of the three examples, is a register for breakpoint N set aside by DDT specifically to be used to store the transfer vector for a conditional break instruction which needs to make an indrect memory reference. So, for example, if the user wants to call subroutine TEST, when breakpoint 2 is reached, he types:

```
I#+2<  →| JMS*  V#+2$Z↵
       →| TEST↵
```

(Recall that when opening a register using < as a command, DDT does not type out the contents of that register.) The user is warned not to insert JMP .+3 in a conditional break register, expecting a double skip. "Point" will not have the correct value to be able to do this.

One may set the conditional break instruction directly, without having to open I#+N, by using the command:

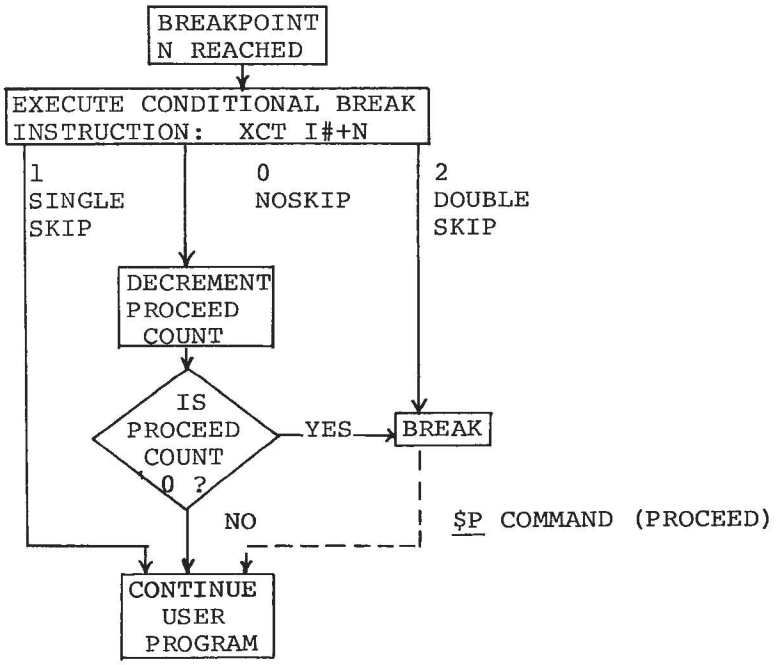
```
X;$N
```

where N = 1, 2, 3 or 4 to indicate the breakpoint number. This command also takes a second argument, but it is immaterial to this discussion.

$N	(with argument X missing) sets I#+N to NOP (no skip)
0;$N	sets I#+N to SKP (skip 1)
1;$N	sets I#+N to skip 2 locations

Only the last bit in argument X is used to determine what is placed in I#+N.

The effective flow chart for the break/no break decision is as follows:



Flow Chart, Break/No Break Decision

4.6.5 What Happens on a Break

When a break occurs, DDT makes three checks:

a) If the break comes from a location not known to be a breakpoint (if the user's program accidentally executes a JMS* 17), DDT reinitializes itself and types:

```
DDT XVM Vnxnnn
>
```

It is not possible to proceed from such a breakpoint.

b) If the break comes from a known breakpoint but PI (program interrupt) is disabled, the same action as above will take place. Therefore, breakpoints should not be placed in routines which operate with PI off.

c) Also, if API is active when the break occurs, DDT will type the contents of the API status register:

```
API   →| 4XXXXX
```

One may not proceed from such a breakpoint. Any other command is valid, including a command to delete this breakpoint.

If a valid break occurs, DDT will type the breakpoint number N followed by a tab and the contents of a register (specified by the address in R#+N) in the prevailing typeout modes, e.g.,

```
1      →| 776403
```

R#+N (N = 1 to 4) is a special register associated with breakpoint N (as are I#+N and V#+N). Initially, when a breakpoint is set with the $B command, R#+N contains the address of A# (the stored AC). Therefore, if the user does not alter R#+N, the contents of the AC will be printed when a break occurs.

The user may change the address in R#+N in either of two ways: one is to open and modify the register,e.g.,

```
>R#+N<  →| ADR
```

the second is by using the command:

A;B$N

The first argument A, changes the contents of I#+N (N = 1 to 4) as discussed in section 4.6.4. Argument B specifies the address whose contents are to be printed when a break occurs at breakpoint N. This address is stored in R#+N.

For example,

0;4004$1

will set the conditional break instruction in I#+1 to skip a location when executed. Since this means a break will never occur, the second argument is unimportant.

1;4004$1

will set I#+1 so that a break will always occur and print:

1 →| contents of 4004

The default argument for B is A#. If 0 is stored in R#+N, when the break takes place only the breakpoint number will be typed.

When breaks occur, DDT saves the contents of the AC, Link, MQ, Index Register, and Limit Register in the special DDT locations addressed by the symbols A#, L#, MQ#, X#, and LR#, respectively.

Once DDT has stopped typing after a break, the user is free to type commands. Typically he would examine certain program registers, perform searches, set and reset breakpoints and make program corrections. Then he may continue program execution at the point where the break occurred by typing:

M$P

This command places the value of the expression M in the proceed count (the default value is 1) and proceeds by simulating the instruction at the breakpoint (the one saved and replaced by JMS* 17) and

gives control to the user program at the location immediately following the breakpoint. If control returns to DDT because the user types ↑T (CTRL T), $P may be used to continue program execution from where it was interrupted. Normally, when no breakpoint has been reached, the $P command will not be accepted since there is no address from which to proceed. However, after DDT has initially been loaded or restarted and has printed:

```
DDT XVM Vnxnnn
>
```

the command $P (as $G) starts the user program at the location specified in SA# (the start address). Recall that the $G command always sets the proceed count to 1.

For convenience, the symbols B1#, B2#, B3#, and B4# are defined as the addresses of the breakpoints. (If breakpoint N is not set, BN# will be defined as 0). DDT never uses these symbols on output, but the user may use them on input to DDT. If breakpoint 1 is set at register 4000, which contains instruction CLA, then the following commands yield the indicated results:

```
B1#/ →|CLA
B1#=004000
```

As already mentioned, DDT initially assumes the use of auto-index register 17 for breakpoint returns. The command:

```
A$0
```

where A is an expression which must evaluate to 10 thru 17, commands DDT to use autoindex register A for its breakpoint return. For example:

```
11$0
```

will change the breakpoint instruction to JMS* 11. The default value of the argument A is 17 ($0 is equivalent to 17$0).

4.6.6 The Execute Command

Built into the DDT breakpoint processor is the facility to allow execution of single instructions. By giving the command:

```
I$X
```

the user may execute the instruction I (an expression). For example,

```
CLA!CLL$X
LAC␣SUM$X
```

The first example will clear registers A# and L# (the saved AC and Link in DDT). DDT will type ? and ignore the command if an attempt is made to execute a CAL instruction, an XCT instruction or an EAE instruction which requires a memory reference to pick up an argument (e.g., MUL). The default instruction, in the absence of I, is the instruction at location "point" executed as if it were located at "point". The reason for this is that if it is a memory reference instruction, it must be executed in the proper memory bank.

If the executed instruction causes a skip, DDT indicates this by typing an extra carriage return. However, it does not alter the address to return to after a break. For example, assume that breakpoint 1 was set as indicated in the following code and that a break at that location has occurred:

```
      LAC (1
A     DAC TMP (breakpoint 1 set here)
B     TAD ADR
```

The break occurs after LAC (1 has been executed and before DAC TMP is executed. If the user now types:

```
SKP$X
  $P
```

program execution returns to location B after the DAC TMP is simulated. The SKP does not cause a skip.

One may execute a call to a subroutine:

```
JMS SUBR$X
```

provided that SUBR does not expect to pick up arguments following the JMS nor skip more than one location on return.

4.7 MISCELLANEOUS FEATURES

4.7.1 Operate Link and AC

The command:

M;N$U

will set the Link (L#) from bit 17 of the value of the expression M and the AC (A#) from the value of N. The default values of both arguments are Ø, so that $U will clear the Link and AC (equivalent to CLA!CLL$X).

4.7.2 Make Subprogram Current (Header Command)

The header command is used primarily to make a particular subprogram current, thus giving preference to its local symbols when DDT performs a symbol table search. The form is:

A$H

The argument A is usually a filename (e.g., PRG1'), but it may be any unique address symbol. This command makes address A the current location, sets the value of "point" equal to A and makes the program containing that address the current program.

4.7.3 Initialize Memory

The command

N$M

where N is an expression, will change the contents of all memory words between the limits in LO# and HI# to the value of N. The default argument is Ø.

4.7.4 Loading DDT without a Program

DDT can be used to create programs on-line and it is not necessary to load any user programs when DDT is called in. This is done by typing ↑T (CTRL T) when the Linking Loader has typed:

```
LOADER XVM Vnxnnn        or        BLOADER XVM Vnxnnn
>                                  >
```

When DDT is started, core has been cleared between P#' and C#, and no load-time nor DDT-time symbols exist.

4.7.5 Restarting DDT

The use of ↑T (CTRL T) to interrupt a user program and return control to DDT has been previously explained. ↑T may also be used to abort a search operation which is in progress.

4.7.6 Typing Mistakes

If the user discovers that he has made a typing error while inputting a command, he may type ↑U (CTRL U) or rubout, both of which are echoed as @, to delete the entire command.

4.7.7 Protect Mode Commands

In order to avoid serious errors, a subroutine in DDT is used to validate addresses before certain commands are executed. The following commands are protected:

a. Proceed ($P) following a ↑T
b. Go ($G)
c. All register modifying commands
d. Breakpoint ($B)

Except for references to the autoindex registers in register modification commands and addresses within the range TP# and the top of core, addresses used by these protected commands must fall within the range P#' to C#. This restriction prevents the user from altering the Monitor, DDT's symbol table, DDT itself, or the bootstrap.

The following two commands affect the protect mode:

$@ Disables the protection feature, thereby allowing the user to modify and transfer to any location in core. DDT will not check for nonexistent memory references.

@ Reenables the protection feature.

4.8 ERROR RECOVERY

Sections 2.11, 4.7.5 and 4.7.6 explain how to correct typing errors and how to stop a runaway program.

The following DDT-generated error messages are not fatal:

U	(undefined symbol)
X	(illegal symbol definition)
O	(DDT-time symbol table overflow)
?	(general error message)
API	(breakpoint reached with API level active)

Commands which cause these errors are ignored. The "API" error message signifies that one may not proceed from such a break.

Errors which are caught by the Monitor (IOPSXX) may or may not be fatal. After unrecoverable IOPS errors, control can then return to DDT if the user types CTRL T (↑T).

If an error is trapped by the FORTRAN Object Time System (OTS), which prints .OTS XX, it will exit to the Monitor. No recovery is possible at that point.

APPENDIX A
SUMMARY OF DDT COMMANDS

A. EXPRESSION OPERATORS

!	logical inclusive or.
&	Logical and.
\	Logical exclusive or.
+	Two's complement sum.
-	Two's complement (unary minus) or subtract.
␣	(Space) Two's complement sum, but prohibit change of top 6 bits in this and remaining operations in the expression.
*	Two's complement sum and exclusive or indirect bit (2ØØØØ) to value.

B. INSTRUCTION MODE COMMANDS

$S	(Symbolic)	Print symbolic instructions.
$O	(Octal)	Print instruction as six unsigned octal digits.
$V	(Vector)	Print bits 3-17 as a transfer vector in the current address mode. If bits Ø-2 are non-zero, precede the address printout with these bits as a 6-digit octal number whose rightmost 5 digits are zero.
$T	(Text)	Consider registers "point" and "point-plus-one" as 5/7 packed ASCII and print the corresponding five characters.
$$R	(Text)	Consider the register "point" to be RAD50 and display the 3 characters.
$$S	(Text)	Consider the register "point" to be SIXBIT and display the 3 characters.

C. ADDRESS MODE COMMANDS

(Relevant to instruction modes $S and $V and also to tag printouts).

$R	(Relative)	Print addresses relative to the nearest tagged (labelled) location.
$A	(Absolute)	Print addresses as absolute 18-bit octal numbers.
$F	(Floating)	Print addresses relative to #. # is the re-location constant for the current program.

D. SPECIAL ADDRESSES IN AND ABOVE DDT

A# Where the AC is stored on breaks.

L# Where the link is stored (bit 17) on breaks.

MQ# Where the MQ is stored on breaks.

LR# Where the limit register is stored on breaks.

X# Where the index register is stored on breaks.

M# Where the default mask for searches is stored.

LO# Where the lower limit address of search operations is stored.

HI# Where the upper limit address of search operations is stored.

R#+N Where the register to be printed on breaking at breakpoint N is stored.

I#+N Contains the instruction to be executed on reacing breakpoint N.

V#+N Reserved for the transfer vector possibly used by the instruction in I#+N.

C# Contains the 2's complement of the proceed count.

TP# Address of the first word of core memory above the bootstrap. Contents of TP# are undefined.

E. SPECIAL ADDRESSES IN USER'S PROGRAM (NOT SEARCHED ON OUTPUT)

Address of the first word in the current program (same as P#' when patch area is current).

. Address of the last opened register.

P#' Address of the first word in the patch area (Pseudo file name)

SA# Address of the starting location of the first program loaded (originally).

B1# Address of breakpoint 1 instruction.

B2# Address of breakpoint 2 instruction.

B3# Address of breakpoint 3 instruction.

B4# Address of breakpoint 4 instruction.

F. CONTENTS OF SPECIAL USER LOCATIONS

Q# Represents the contents of the most recently opened storage word (never used on output) or value of last intervening retype command argument.

G. DEFINING A SYMBOL

E(S) Symbol S given value E (expression).

S) Symbol S defined as .; same as .(S).

$K Kill DDT-time user defined symbols.

O$K Kill load-time user symbols.

H. STARTING A PROGRAM

AD$G	Start at address AD (expression).
$G	Start at address SA#.
;AD$G	Set SA#=AD and start there.

I. BREAKPOINT COMMANDS

K$B	Put lowest available breakpoint at address K (must be non-zero; default: .) DDT types breakpoint number after the B.
O$B	Remove all breakpoints.
N;K$B	Set breakpoint N (default: 4) at address K (must be non-zero; default: .).
N;O$B	Remove breakpoint N (Default: 4).
M;K$N	Type out register K (default: A#) when breaking at breakpoint N (1 to 4) (N is part of command and may not be omitted). If M; is missing, normal procedure will be followed at breakpoint. If M bit 17 is Ø, breakpoint will never break. If M bit 17 is 1, breakpoint will always break.
E$O	Use auto-index register E (10-17; default: 17) for breakpoint instructions.
M$P	Proceed from breakpoint and put M (default: 1) in proceed count.

J. REGISTER EXAMINATION AND MODIFICATION

↵	Close any open register, depositing its argument in that register. If no argument is given, the register is unchanged.
/	Close any open register and open another. If an argument is given, it is taken as the 18-bit address of the next register to be opened and made current. If not, the register "point" is opened.
LINE FEED	Behaves like ↵ except that it opens the register following the current register and makes the new register current.
↑	Behaves like ↵ except that it opens the register preceding the current register and makes the new register current.
$Y	Behaves like ↵ except that Q# is taken as the 12 (page mode) or 13 (bank mode) bit address of a location which is opened and made current.
$J	Behaves like ↵ except that Q# is taken as the 15 bit address of a location which is opened and made current.

$Z	Behaves like $Y except that the new open register is not made current.
[	Behaves like /, but forces the printout to be in numeric mode.
]	Behaves like /, but forces the printout to be in symbolic mode.
<	Behaves like / except that the printing of register contents is omitted until / is used to close a register (type-in mode).
=	Print argument (default: Q#) as an octal number.
←	Behaves like = but causes symbolic printouts.
:	Behaves like = but retypes expressions as transfer vectors.
?	Prints out contents of . and .+1 interpreted as 5/7 packed ASCII text.
$?	Prints out contents of . as SIXBIT ASCII.
$$?	Prints out contents of . as RAD5Ø.

K. WORD SEARCHES

M;N$L	Set the contents of LO# to M (default: P#') and HI# to N (default: L#).
M;D$W	Search from address in LO# through address in HI# (? if lower than address in LO#) for words which are the same as D (default: Ø) in all bits that are 1 in M (default: contents of M#).
M;D$N	Search as in $W, but print unequal words.
M;D$E	Search as in $W and $N for memory reference instructions effectively addressing (directly or indirectly) an address which is identical to D (default; Ø) in every bit which is 1 in M (default: contents of M#).

L. OTHER DDT COMMANDS

I$X	Execute instructions I (default: contents of "point" as if located at address "point").
↑U	Wipe out current command.
RUBOUT	Same as ↑U.
C$H	Make the address C (e.g., FILE') current, but do not open it. (Default: SA#).
M;N$U	Update contents of L# to M (default: Ø) and the contents of A# to N (default: Ø).
N$M	Initialize memory from address in LO# through address in HI# to N (default: Ø).
↑T	Interrupt -- transfer control to DDT. Used to stop a runaway program or to abort a long search operation.

@	Enable Protect mode.
$@	Disable Protect mode. Allow modifications of and transfer to any location in core.

M. ERROR MESSAGES

?	General error indication. Command ignored.
X	Symbol definition command attempted to redefine a non-DDT-time symbol. Command ignored.
U	Symbol just used is undefined. Command ignored.
API	A breakpoint has been reached with API active (cannot proceed).
O	Symbol table capacity has been exceeded by symbol definition command. Command ignored.

N. SPECIAL DDT SYMBOLS

;	First argument delimiter for commands with two arguments.
(	Same as ; when used with symbol definition commands.
$	Indicates that the following symbol is a command (used primarily with letters and numbers, which ordinarily comprise expression syllables).
ALTMODE	Same as $ and echoes as $.
↗>	Indicates that the command has been performed by DDT.
→| >	Same as ↗>.
'	Indicates that the preceding symbol is a file name (must be used with file names for correct operation).
"	Indicates that the preceding symbol is an address (omit searching the instruction mnemonic symbol table).
*	Sets the indirect bit of an argument (20000).
,X	Sets the index bit of an argument (10000).
,	Same as ,X.

APPENDIX B
MNEMONIC INSTRUCTION TABLE

MEMORY REFERENCE

CAL	000000
DAC	040000
JMS	100000
DZM	140000
LAC	200000
XOR	240000
ADD	300000
TAD	340000
XCT	400000
ISZ	440000
AND	500000
SAD	540000
JMP	600000

EAE GROUP

EAE	640000

INPUT/OUTPUT

IOT	700000

INDEX CLASS

PAX	721000
PAL	722000
AAC	723000
PXA	724000
AXS	725000
PXL	726000
PLA	730000
PLX	731000
CLX	735000
CLLR	736000
AXR	737000

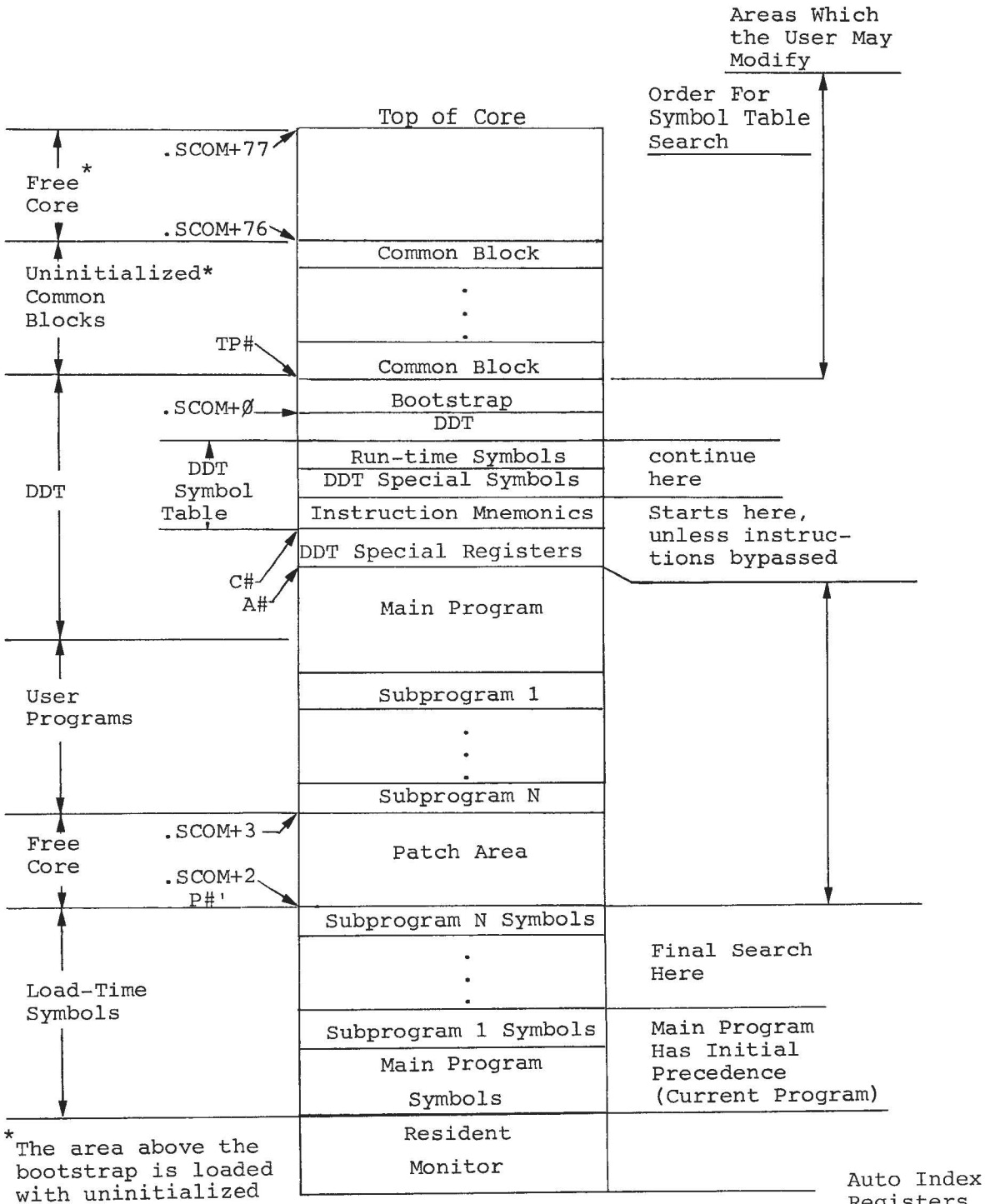
OPERATE CLASS

NOP	740000*
OPR	740000
CMA	740001
CML	740002
RAL	740010
RAR	740020
IAC	740030
SMA	740100
SZA	740200
SNL	740400
SKP	741000
SPA	741100
SNA	741200
SZL	741010
RTL	742010
RTR	742020
SWHA	742030
CLL	744000
STL	744002
RCL	744010
RCR	744020
CLA	750000
CLC	750001
GLK	750010
LAW	760000

* DDT interprets 740000 as NOP.

APPENDIX C

DDT MEMORY LOAD MAPS



*The area above the bootstrap is loaded with uninitialized COMMON blocks if XVM mode is ON. Free core above the bootstrap should not be used as a patch area, although it can be used to store additional data. The area above the bootstrap should not be used for any purpose if XVM mode is OFF.

Auto Index Registers

INDEX

READER'S COMMENTS

NOTE: This form is for document comments only. Problems with software should be reported on a Software Problem Report (SPR) form.

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

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

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

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

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

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code or Country _____

If you require a written reply, please check here. ☐

Please cut along this line.

---------- Fold Here ----------

---------- Do Not Tear - Fold Here and Staple ----------

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

Postage will be paid by:

digital

Software Communications
P. O. Box F
Maynard, Massachusetts 01754

digital

digital equipment corporation

PRINTED IN U.S.A.