DDT PROGRAMMING MANUAL

PDP-6

DIGITAL EQUIPMENT CORPORATION • MAYNARD, MASSACHUSETTS

DEC-6-0-UP-DDT-UM-FP-ACTOI

PDP-6 PROGRAMMING MANUAL
DDT-6
DYNAMIC DEBUGGING TECHNIQUE

DIGITAL EQUIPMENT CORPORATION • MAYNARD, MASSACHUSETTS

CONTENTS

CONTENTS (continued)

ILLUSTRATION

TABLE

CHAPTER 1

INTRODUCTION

Historically, when a program error occurred, the computer user sat down at the computer console and stepped through his program to determine what went wrong. Although producing results, this method of program check-out was an extremely inefficient use of both computer and programmer time.

An improvement in this method of operation occurred when programs were "batched" on a magnetic tape. Batched processing allows for each program to be run consecutively without intervention. This method therefore provides for more efficient use of the computer.

Batch processing, however, loses the distinct advantage of the previous method of program check-out. That is, there is no direct interaction between the machine and the user. Even though dynamic dumps or trace routines are available in batch processing systems for program check-out, information can only be obtained at prearranged points in the program and under prearranged conditions. Thus, a computer which has no ability to reason is allowed in the batch processing method to run blindly until an error situation occurs or the program halts.

At this point, a memory dump, or what is commonly called a post-mortem dump, is usually taken so that the programmer may have a snapshot of a static condition. With the use of the dump the programmer hopes the information is available to find the problem and in most cases he can find the error. The user, however, must wait several hours to get another computer run of his program even though the program error may have been a minor one (e.g., forgetting to reset an index register). In other cases he does not have the right information and therefore must place dumps at other points. Thus two or more runs are needed to get the necessary information. Since the waiting time between computer runs is approximately 3 to 24 hours, the time to check out a program by the batch processing method is too time consuming.

In contrast to the previous two situations, the PDP-6 time-sharing approach allows for the maximum number of programs to be run per hour and still allows for dynamic interaction during the check-out of programs. The name given to this on-line dynamic check-out system is DDT-6, the Dynamic Debugging Technique.

DDT-6 has all the dump capabilities of the batch processing debugging system. That is, the user of DDT-6 can set breakpoints any place in memory. When the breakpoints are reached, portions of memory are dumped conditionally or unconditionally as determined by the user. All memory can be dumped as well.

DDT-6 is a more extensive and flexible debugging system with significant improvements over conventional techniques. In DDT-6 it is possible, for instance, to make symbolic changes to the program at any time. These can be insertions or deletions. With this capability each of many users can sit down at remote consoles and check out programs in the assembly language without being concerned with the more obscure machine (binary or octal) language. Consequently both dynamic interaction plus maximum use of the computer system via many users is achieved. In addition, with many easily used commands the user can interrogate his program at will. If the user wishes to see the contents of some absolute or symbolic memory location, he merely types that memory location followed by a slash and the contents of that location are typed out on the console printer. The user also has the option of determining whether the information is typed out in octal, symbolic, decimal, floating point, etc.

All these commands and many more, for a total of 50, have been implemented in order to handle any debugging situation efficiently and easily by multiple users.

CHAPTER 2

REGISTER EXAMINATION AND CHANGE

To debug a program, the programmer must be able to examine and change memory locations. These memory locations are represented by a symbol and/or an absolute number.

The register examination pointer pinpoints the address of a register that can be changed or modified. In this regard the register examination pointer normally points to the address of the register which has been "opened." A register that is considered open by DDT may have a value typed into it. All underlined information in given examples represents information typed by DDT. The user of DDT-6 typed everything else.

/ The slash commands DDT to type out in the current mode (symbolic instruction, half word, constants) the contents of the address last typed by DDT or the programmer. The address is now open for modification.

Example:

ADR/ MOVE A,CC1

If an address immediately precedes the slash (no spaces separate the address from the slash), as in the above example, the register examination pointer . (period) is set equal to that address.

The register examination pointer (.) is now set equal to ADR. Thus with the use of the register examination pointer the user can reinterrogate the currently open address.

Example:

./ MOVE A,CC1

Since the register examination pointer (.) has been set equal to the address (ADR), in the preceding example, the typeout for both examples is the same.

Once an address has been opened, its contents may be altered by typing the new expression following the typeout by DDT.

Example:

ADR/ MOVE 3,ETA MOVE A,T

Thus ADR was opened by immediately following the address by a slash. DDT-6 typed out MOVE 3,ETA. At this point, the user decided to change the contents of the address (ADR), so he typed MOVE A,T.

It should be noted that the contents of any address can be examined by the forward slash (/) whether that address immediately precedes the slash or not. (The address, however, must immediately precede the slash in order to set the register examination pointer (.) to that address.)

Example:

ADR/ MOVE AC,IO / ADD 2,SUM

The second use of the slash types out the contents of the address (IO) preceding the slash. In this case the register examination pointer (.) remains set equal to ADR not IO (even though the contents of address IO have just been typed out).

It should also be noted that the spaces that occur after DDT completes the typing of the contents of ADR are automatically produced by DDT, not the user.

[The left bracket has the same effect as the slash, (the address immediately preceding the [will be opened) except that it forces the typeout to be in numbers of the current radix (octal or decimal).

Example:

ADR[11 (octal)

ADR[9. (decimal)

(leading zeros are suppressed)

] The right bracket has the same effect as the slash except that it forces the typeout to be in symbolic instructions.

Example:

ADR] MOVE 1,105

The following commands close an open register; that is, they accomplish the modification of the open register if a modification has been typed in; otherwise, the original contents remain the same.

↓ The line-feed key causes a carriage return to be typed then adds one to the register examination pointer and types the new resulting address followed by the contents of that address. The register examination pointer is set to the resultant address, and that address is now open.

Example:

ADR/ MOVE A,CC1 MOVE A,C ↓
ADR+1/ ADD A,CC3

The symbolic or absolute address, represented by ADR, followed by a slash opens this address and types out MOVE A,CC1. Since the register or address is now open, the user types in the change, MOVE A,C, and hits the line-feed key. The line-feed key causes the contents of ADR to be changed to MOVE A,C, and the teletypewriter skips to the next print line.

Then the address, ADR, is incremented by one, and the contents of the resultant address (ADR+1) are automatically typed out by DDT-6.

↑ The vertical arrow key causes a carriage return, then subtracts one from the register examination pointer and types the resulting address followed by a slash and the contents of the resultant address. The register examination pointer is set to the resultant address, and that address is now open.

Example:

ADR+1/ MOVE AC,CC MOVE A,C↑
ADR/ SUB 2,Y ↓
ADR+1/ MOVE A,C

→| The horizontal tab causes a carriage return line-feed, then sets the register examination pointer to the address (the new address if a modification was made) of the instruction in the register just closed. Then DDT types this new address, followed by a slash, followed by the contents of that location.

Example:

```
ADR5/    JRST ADR1          JRST ADR  →|
ADR/     MOVEM B,CC2        →|
CC2/     666
```

\ The backslash opens the register addressed by the last location typed and types out the addressed register's contents. Backslash does not change the register examination pointer. Unlike the forward slash, the backslash causes the previously opened register to be modified if a new quantity has been typed in.

Example:

```
ADR/     MOVE A,CC2         JRST X\        MOVE AC,3
```

The use of the backslash accomplishes two things. First it changes ADR by replacing its contents with JRST X. Second, the backslash causes DDT to type out the contents of X, namely, MOVE AC,3. The register examination pointer continues to point to ADR.

If the line-feed control character and the vertical arrow were used in conjunction with the backslash, the results would be as follows:

Example:

```
ADR/     MOVEM B,CC2        MOVE A,CC1\  105776 ↓
ADR+1/   MOVE A,C           ↑
ADR/     MOVE A,CC1
```

Carriage Return The carriage return causes a line-feed to be typed. All temporary typeout modes (symbolic instructions, constants, etc.) are returned to permanent modes. The currently open register, if any, is modified if a new quantity has been typed, and then is closed.

!

The exclamation point works exactly like slash except that it suppresses type out of register contents until either /, [, or] is typed by the user.

Example:

ADR!	MOVE AC,555 ↓		(1)
ADR+1!		(Carriage return key hit here)	(2)
ADR/	MOVE AC,555		(3)

Thus, in the first part (1) of the example the contents of ADR are not typed out, but the address is open to modification and MOVE AC,555 has been typed in by the user.

Step two (2) of the example shows that the register examination pointer has been incremented by one and the contents of ADR+1 are not typed out. This is because the exclamation point is still in effect. The exclamation point will continue to take effect until slash (/), open bracket ([), or close bracket (]) is typed in by the user. In this case, hitting the slash terminates the effect of the exclamation point.

Step three (3) shows that the modification (MOVE AC,555) of ADR typed in step one (1) has been accomplished.

The following is a summary in table form of those special control characters and their corresponding functions. For example, the chart shows that the forward slash (/) will examine the contents of an address, type out in the current mode, open the address, change the examination pointer to the address just opened, but cannot be used to insert new quantities in that address.

TABLE 1 SPECIAL CONTROL CHARACTER FUNCTIONS

Control Character	Examine Contents	Mode	Address Opened	Change Register Examination Pointer	Insert New Qty. If New Qty. Has Been Typed
/ [ ] !	Yes Yes Yes No	Current Constant Symbolic -	Yes	Yes*	No
\	Yes**	Current	Yes	No	Yes
tab (→|)	Yes**	Current	Yes	Yes	Yes
↑	Yes**	Current	Yes	Yes (-1)	Yes
line-feed (↓)	Yes**	Current	Yes	Yes (+1)	Yes
carriage return (↵)	No	None	No (closes)	No	Yes

*If a quantity immediately preceded.

**If ! has not suppressed typeout.

CHAPTER 3

MODE CONTROL

REGISTER MODE CONTROLS

Mode control allows a user to present or be presented with information in a temporary or permanent radix (base) by the use of a single or double dollar sign. The single dollar sign preceding a control character will temporarily set the information being typed to the radix corresponding to that control character (as explained below). The double dollar sign will set the typed information permanently (until another $$) to the radix desired by the user.

A temporary mode change is terminated when one of the methods that "closes" all registers is initiated. To close a register means that the register now is not open for modification. That is, the carriage return closes all registers and the temporary mode is terminated. The tab (→|), however, closes one register and opens another, and therefore the temporary mode remains in effect.

There are four types of register mode controls:

$S(or $$S)* Changes the typeout mode to symbolic instructions temporarily or permanently depending on the number of dollar signs.

Example:

X/ 200100003716 $SX/ MOVE 2,M

$T ($$T) This command types out registers in the ASCII character set. Left justified characters are assumed unless the leftmost character is null. Then right justified characters are assumed. This is to make it possible to type out ASCII text in its normal form and also as a single right justified character.

*Hitting the ALT MODE key produces the same result as hitting the $ key. The ALT MODE key is a lower case character on the teletypewriter and is therefore more easily used than the $ key.

$H($$H) Changes the typeout mode to half words in the current radix.

Example:

100/ (CAR)CDR

where CAR is the address in the left half of the word, and CDR is the address in the right half of the word. The typeout depends on the current radix. (The parentheses around the left half of the word will always be typed out.)

$C($$C) Changes the typeout mode to constants, numbers of the current radix.

Example:

$$S X/ MOVE 2,M $$C X/ 200100003716

$F($$F) Changes the typeout mode to floating point.

ADDRESS MODE CONTROL

This command is used for typing out relative addresses.

$R($$R) Addresses will be typed out with a symbol plus a constant if the constant is less than 64 and if the address is greater than 48; otherwise, addresses will be constants in the prevailing radix.

Example:

$$R X/ MOVE AC,ABLE+50

The address ABLE+50 has been typed out with a symbolic address (ABLE) plus a constant (50_8).

$A($$A) All addresses will be constants in the prevailing radix.

N$R (N$$R) Any radix (≥ 2) may be set by typing N$R for temporary change of radix or N$$R for permanent change of the output radix, where N stands for the radix desired by the user.

Example:

12$R - This will temporarily change the radix to decimal because the normal input radix is octal ($12_8 = 10_{10}$).

TYPEOUT MODE CONTROLS

These are three special characters which cause DDT-6 to type out information in a certain format regardless of the mode set by one of the previously defined methods.

← The left arrow forces DDT to retype the last quantity in the symbolic instruction or half word format depending on the mode.

Example:

402002002733←SETZM M+3(2)

= The equal sign forces DDT to retype the last quantity as a constant of the current radix.

Example:

X/ SETZM M+3(2) =402002002733

& The ampersand forces DDT to retype the last quantity as a radix 50 symbol (see description of the Linking Loader).

TYPE IN

" The double quote (") is used to insert ASCII characters into memory registers.

To insert a single right justified character, type double quote, the desired character, and then hit the ALT MODE key.

Example:

"A ALT will insert the corresponding ASCII bit configuration for A (101 in octal).

To insert left justified ASCII characters type double quote, then some "terminating" character, the desired characters to be inserted, and finally the terminating character that preceded the desired characters. Up to five ASCII characters can be inserted.

Example:

X! "/ABCD/ (carriage return)

Thus ABCD will be inserted left justified in X.

The user may type two addresses into the same word in similar format to the half word mode (see page 10).

Example:

ADR! (,-1) 5000 (carriage return)

This command results in -1 (truncated to 18 bits by the comma) going into the left half, and 5000 going into the right half.

Anything typed out by DDT can be typed in with the identical binary result.

CHAPTER 4

PROGRAM INTERROGATION

SEARCHES

There are three types of searches: The word search, the not-word search, and the effective address search.

The searches can be done between limits.

	W	Word search
a < b > c $	N	Not-word search
	E	Effective address/search

a — Is the lower limit of the search; 0 is assumed if this argument and its delimiter are not present.

b — Is the upper limit of the search. The lower numbered end of DDT's symbol table is assumed if this argument and its delimiter are not present.

c — Is the quantity searched for.

Hitting any Teletype key terminates a search.

The effective address search will find and type out all locations where the effective address, following all indirect and index-register chains to a depth of 64_{10} levels, equals the address being searched for.

Examples:

```
4517<5000>X$E
INPUT<5000>700$E
```

Examples of DDT output when searching for X in the above example:

```
4517/    SETZM X
4721/    MOVE 2,X
5000/    MOVE 3,@4721 (Indirectly addresses X through address 472
```

The word search and the not-word search compare each storage word with the word being searched for in those bit positions where the mask has ones. The mask contains all ones unless otherwise set by the user. If the comparison shows an equality, the word search types out the address and the contents of the register; if the comparison results in an inequality, the word search would type out nothing. The not-word search types nothing if an equality is reached it types the contents of the register when the comparison is an inequality.

Examples:

```
INPT<INPT+10>NUM$W
INPT<INPT+10>5000$N
```

$M/ This command types out the contents of the mask register, which is now open. The contents of the mask register are ordinarily all ones unless changed by the user.

NUM$M NUM$M inserts NUM into the mask register.

BREAKPOINTS

The ability to automatically stop and examine the program at specific strategic points is an integral part of DDT. This ability to stop the program can occur every time the program executes a particular instruction, every hundredth or so time, or only when a particular condition occurs. DDT allows up to eight such breakpoints.

To Insert Breakpoints

WORD $nB The expression WORD consists of the address at which the user wants the "break" (program transfer to DDT control) in the program to occur.

WORD can consist of symbolic and/or absolute information. n stands for one of eight possible breakpoints ($1 \leq n \leq 8$).

Example:

```
4002$2B
```

Break occurs at 4002.

WORD may combine the breakpoint address with the name of a register, in parentheses, for which the contents will be typed at the time the break occurs.

Example:

4000(X)$8B

Break occurs in program at address 4000 and types out the contents of X.

The break in the program occurs before the instruction in that particular address is executed.

WORD$B — This command to DDT places the expression WORD in the next available breakpoint in DDT (maximum of eight breakpoints). This allows the user to forget which breakpoints have been used previously. If no free breakpoints remain, DDT will type a question mark.

To Remove Breakpoints

$B — This command removes all breakpoints.

0$NB — This command removes the Nth breakpoint (where 0 is a zero).

Example:

0$2B

This removes the second breakpoint.

Restrictions for Breakpoints

Breakpoints may not be used with instructions that are:

1. Modified by the program.
2. Used as data or literals.
3. Used as part of an indirect addressing chain.
4. User mode programmed operators that call the time sharing monitor. (Those programmed operators that result in control returning to user location 41 may be used with breakpoints.)

Restarting After a Break Occurs

The program is usually restarted after a break by using the breakpoint proceed command. N$P commands DDT to put N (if N is not present, 1 is assumed) into the proceed counter for the breakpoint last encountered, execute the instruction at that breakpoint, and return control to the user's program.

Example:

100$P

This command causes DDT to return to the user's program and start executing at the point left off before the break occurred. In addition this break will not occur again until the instruction at this breakpoint is executed 100 times.

When a breakpoint is inserted with a double dollar sign ($$)

Example:

adr(AC)$$B

or when $$P is used after a breakpoint breaks, DDT will do an automatic proceed (as opposed to the manual proceed where the user types $P) after the breakpoint information is typed out.

Breakpoint Registers Internal to DDT

For each breakpoint, specified by $nB where n is a number (1-8), internal to DDT are three registers which may be examined and changed and which contain information about the breakpoint.

$nB/	Address of breakpoint in right half. Address of register to examine (or 0) in left half. If both halves equal 0, the breakpoint is not in use.
$nB+1/	Conditional break instruction, or 0.
$nB+2/	Proceed counter. If the proceed counter is less than or equal to 0, a break occurs.

When control is transferred from DDT to a program, each breakpoint register ($1B - $8B) is examined and, if it has a breakpoint specified, the instruction at the address of the breakpoint is saved and is replaced by a JSR to DDT's breakpoint logic.

When a breakpoint is encountered, control is transferred to DDT. At that point the conditional break instruction is executed if there is one. Any non-zero value in $nB+1 is considered a conditional break instruction. The conditional break instruction may be a single instruction or a call to a closed subroutine.

If the net result of executing the conditional break instruction is a skip in the sequence of instructions executed (instruction after breakpoint is skipped), a break occurs.

Example:

If address 6700 was reached and DDT's fourth breakpoint registers were as follows:

```
$4B/        6700 (AC1)
$4B+1/      CAIE AC1, 100
$4B+2/      200
```

and AC1 contained 100, DDT would type

```
$4B >6700  AC1/ 100
```

If AC1 did not contain 100, no break would occur; $4B+2 would be decremented by one and the users program would continue running (pick up at the point where the PC for the users' program is set).

If the conditional break instruction transfers to a subroutine which, after the subroutine is executed, causes the next two instructions to be skipped in the user's program (PC in user's program is increased by two during the execution of the subroutine), a break will never occur regardless of the proceed counter.

Example:

If the internal DDT breakpoint registers ($2B and $2B+1) have the following contents, a break would not occur unless accumulator 3 contains 100.

```
$2B/        ADR
$2B+1/      JSR TEST
```

```
TEST/      0              (contains PC when jump to subroutine
                          TEST is made)
TEST+1/    AOS TEST
TEST+2/    CAIE 3, 100
TEST+3/    AOS TEST
TEST+4/    JRST @TEST
```

The subroutine test causes a double skip (the return is to the third instruction after the call) in the user's program if accumulator 3 does not equal 100_8. A break would never occur at address ADR (regardless of the proceed counter) unless accumulator 3 contained 100_8.

If the conditional break instruction does not cause a skip, or if the instruction equals zero, the proceed counter is decremented by one. (If the user wishes a break to occur based only on the conditional instruction, he should set the proceed counter to some very large number so that the proceed counter will never reach zero.) If the result is less than or equal to zero, a break occurs. Otherwise the programmer's instruction at the address of the breakpoint is executed, and control leaves DDT and goes to the program.

When a break occurs, the state of the user's program is saved (see Entering DDT), the JSR breakpoint instructions are removed, and the programmer's instructions are restored.

DDT types out the number of the breakpoint and a symbol indicating the reason for the break, > for the conditional break instruction, >> for the proceed counter and the address in the user's program where the break occurred.

Example:

If address ADR was reached in the user's program and DDT's breakpoint registers contained:

```
$2B/       ADR
$2B+1/     0
$2B+2/     0              (proceed counter)
```

DDT-6 would type

```
$2B >>ADR
```

Starting a Program

ADR$G — This command causes the program to start executing instructions at the address specified by ADR.

Example:

```
4000$G
```

The program begins to execute instructions starting at 4000. This command can also be used after a breakpoint brings the program to a halt and the user is given the information required. He may then wish to interrogate certain memory locations and start again. If he wanted to start again at address 6000, he would type 6000$G.

INSTRUC$X — This command causes INSTRUC to be executed.

Example:

```
ADD AC,Y$X
```

CHAPTER 5

FUNDAMENTAL DEFINITIONS

SYMBOLS

Certain symbols can be referenced in one program from another. These symbols are called "global." Those which can only be referenced from within the same program are called "local." Any symbol which has been defined as global (internal or entry) by the assembler (MACRO-6) will be considered as global by DDT-6 when it is referenced. All DDT defined symbols are considered global.

The user may want to reference a local symbol in a particular program. This is possible by stating the program name followed by $: . Thus if a user wishes to use a symbol(s) local to program MIN, the command MIN$: accomplishes this for him.

Defining Symbols

There are two ways to assign a value to a symbol: numeric value < symbol: and TAG: .

NUMERIC VALUE < SYMBOL: This command puts SYMBOL into DDT-6's symbol table with a value equal to the specified NUMERIC VALUE. SYMBOL is any legal symbol defined or undefined.

Example:

305<XVAR:

XVAR has now been defined to have the value 305.

TAG: This command puts TAG into DDT-6's symbol table with a value equal to the address of the last register opened.

Example:

400/ ADD 2,12012 X:

This puts the symbolic tag X into DDT-6's symbol table and sets X equal to 400, the address of the last register opened.

Deleting Symbols

There are times when the user will want to restrict or eliminate the use of a certain few or even all defined symbols. The following three ways give the user of DDT-6 these capabilities.

SYMBOL $$K — SYMBOL is killed (removed) in DDT-6's symbol table. SYMBOL can no longer be used for input or output.

Example:

X$$K

This command would remove X from DDT's symbol table.

$$K — This command kills all symbols in the DDT-6 symbol table which were previously defined by the user. If the symbols DELTA, ETA, SUM, XI, and YI, were previously defined by the user, $$K eliminates them from further use. If they are used without being defined again, DDT types out a U after the symbol.

SYMBOL$K — This command prevents DDT from using this symbol for typeout; it still can be used for typein.

As an example, the user may have set the same numeric value to several different symbols. The user, however, does not wish certain symbol(s) to be typed out as addresses or accumulators. The command SYMBOL$K restricts certain typeouts.

X/ MOVE J,SAV J$K ← MOVE N,SAV N$K ← MOVE AC,SAV

Since the user does not wish J to be typed out as an accumulator, he types in J$K; he follows this with a left arrow to type out the contents of X again and MOVE N, SAV is typed out. He then repeats the above process until the desired result, namely AC, is typed out. Any further reference to the contents of X results in the latter typeout (MOVE AC,SAV).

Zero Core

There are a number of circumstances when the user will want to zero out certain memory location(s). The following command will provide this capability:

FIRST<LAST $$Z This command will zero out the memory locations between the indicated FIRST address and LAST address inclusively. If the FIRST address is not present, the zero location is assumed. If the LAST address is not present, the location before the low end of the symbol table is assumed. In no case will locations 20-37, or in time sharing 20-137, nor any part of DDT be zeroed.

DDT Assembly

When improvising a program on line to the PDP-6 on a teletypewriter, the user will want to use symbols in his instructions in making up the program. In this and in other situations, undefined symbols may be used by following the symbol with the # sign. The symbol will be remembered by DDT from then on. Until the symbol is specifically defined by the use of a colon or semicolon, the value of the symbol is taken to be zero. Successive uses of the symbol cause DDT to type out the # sign. Appending the # sign to all subsequent uses of the symbol enables the user to readily identify undefined (not yet defined by a colon or semicolon) symbols.

Example:

```
MOVE 2,VALUE#
```

VALUE is now remembered by DDT-6 and may be used further without the user appending the # sign. If subsequent instructions are given involving VALUE, DDT-6 appends a # sign automatically to that symbol. Thus VALUE will always appear as VALUE followed by the # symbol (until VALUE is defined).

Example:

```
START!      MOVE 2,VALUE#↓     (user types the # sign)
START+1!    ADDI 2,50↓
START+2!    MOVEM 2,VALUE ↓#   (DDT-6 types #)
```

START+3! JRST VALUE+#1↓ (DDT-6 types # after the plus sign because at only that point DDT realizes the symbol VALUE is complete.)
START+4!

Undefined symbols can only be used in operations involving addition or subtraction. The undefined symbols may only be used in the address field.

Example:

```
MOVEI 2,3*UNDEF#
```

This is an illegal operation multiplication with a symbolic tag (UNDEF) which has not previously been defined.

? The question mark lists all undefined symbols that have been used up to that point in the program.

Example:

```
?
VALUE
UNDEF
```

BASIC DEFINITIONS

Symbols

A symbol is a string of up to six letters and numbers including the special characters . and %. Characters after the sixth are ignored. A symbol must contain at least one letter. If a symbol contains numerals and only one letter, that letter must not be a B, D, or an E. These letters are reserved for binary shifted and floating point numbers.

Numbers

A number may be octal, decimal, or floating point. An octal number consists of a string of digits (0-7).

A decimal number consists of a string of digits (0-9) terminated by a decimal point. The number may be no larger than 34,359,738,367.

A floating point number consists of the integer digits followed by a decimal point followed by the fraction digits. If the integer part is zero, it may be eliminated entirely; the fraction part may not be eliminated. A floating number may be followed by the letter E and a decimal exponent. Some examples are:

```
6.0
6.02E+4        (=60200.0)
 .12E-4        (=.000012)
20.0E7         (=200000000.0)
```

Arithmetic Operators

+	The plus sign means 2's complement addition.
-	The minus sign means 2's complement subtraction.
*	The asterisk means integer multiplication.
'	The single quote means integer division.

Symbols, numbers, and/or quantities inside parentheses (which appear preceded by +, -, *, ') are combined by +, -, *, ' to form expressions.

Examples:

```
6+2
S
2*(SYM+17)
2*3+1
```

Expressions are combined with space, comma, and/or quantities inside parentheses (which are not preceded by +, -, *, ') to form storage words.

Examples:

```
MOVE 6+2,S
(ADR1)ADR2
SKIPA A+1,4*3(IX2)
```

Field Separators

The storage word is considered by DDT to consist of three fields: the 36-bit whole word field; the accumulator or I/O device field; and the address field. Expressions are combined into these three fields by two operators:

Space — The space adds the expression immediately preceding it into the storage word being formed. It also sets a flag so that the expression going into the address field is truncated to the rightmost 18 bits.

Comma — The comma does three things: the left half of the expression is added into the storage word; the right half is shifted left 23 bits (into the accumulator field) and added into the storage word. If the leftmost three bits of the storage word are ones, the comma shifts its expression left one more place (I/O instructions thus shift device numbers into the device field). The comma also sets the flag to truncate addresses to 18 bits.

The comma is for placing information into the accumulator field.

The address field expression is terminated by any word termination command or character.

Expression Component

Parenthesis — A left parenthesis stores the status of the storage word assembler on the push down list and reinitializes the assembler to form a new storage word. A right parenthesis terminates the storage word and swaps its two halves to form the expression inside the parentheses. This expression is treated in one of two ways:

1. If +, -, ', or * immediately preceded the left parenthesis, the expression is treated as a term in the expression being assembled and therefore may be truncated to 18 bits if part of the address field.

2. If +, -, ', or * did not immediately precede the left parenthesis, the expression/quantity is added into the storage word.

Parentheses may be nested.

Special Symbols

@ The @ sign will set the indirect bit into the word specified.

Example:

```
MOVE AC,@X
```

CHAPTER 6

PAPER TAPE

PAPER TAPE CONTROL

$L — This command causes DDT to punch a checksummed loader, in DDT paper tape format. (For DDT paper tape format, see page 29.) Thus if the user wishes to punch out a program on paper tape, he would give a $L command first in order to get a loader punched on the same tape as the program. Later when the user wishes to read in the program from the paper tape, the RIM loader will load the checksum loader into the 30 locations just below the contents of location 37 and then the program will be loaded by the checksum loader.

FIRST<LAST TAPE — This command allows the user to punch out checksummed blocks in DDT format on paper tape from consecutive locations between FIRST and LAST address inclusively. This commandwill, for example, punch out a program existing in core memory in its present state of check-out for later use.

Example:

4000<20000 TAPE

ADR$J — This command punches a 1-word block that causes a transfer to address ADR after the preceding program has been loaded from paper tape. If ADR is not present, a JRST 4,DDT is punched.

The following succession of steps would punch a program on paper tape ready to be used as an independent entity.

1. $L
2. 5000<20000 TAPE
3. 6000$J (Transfer to address 6000 after program is loaded.)

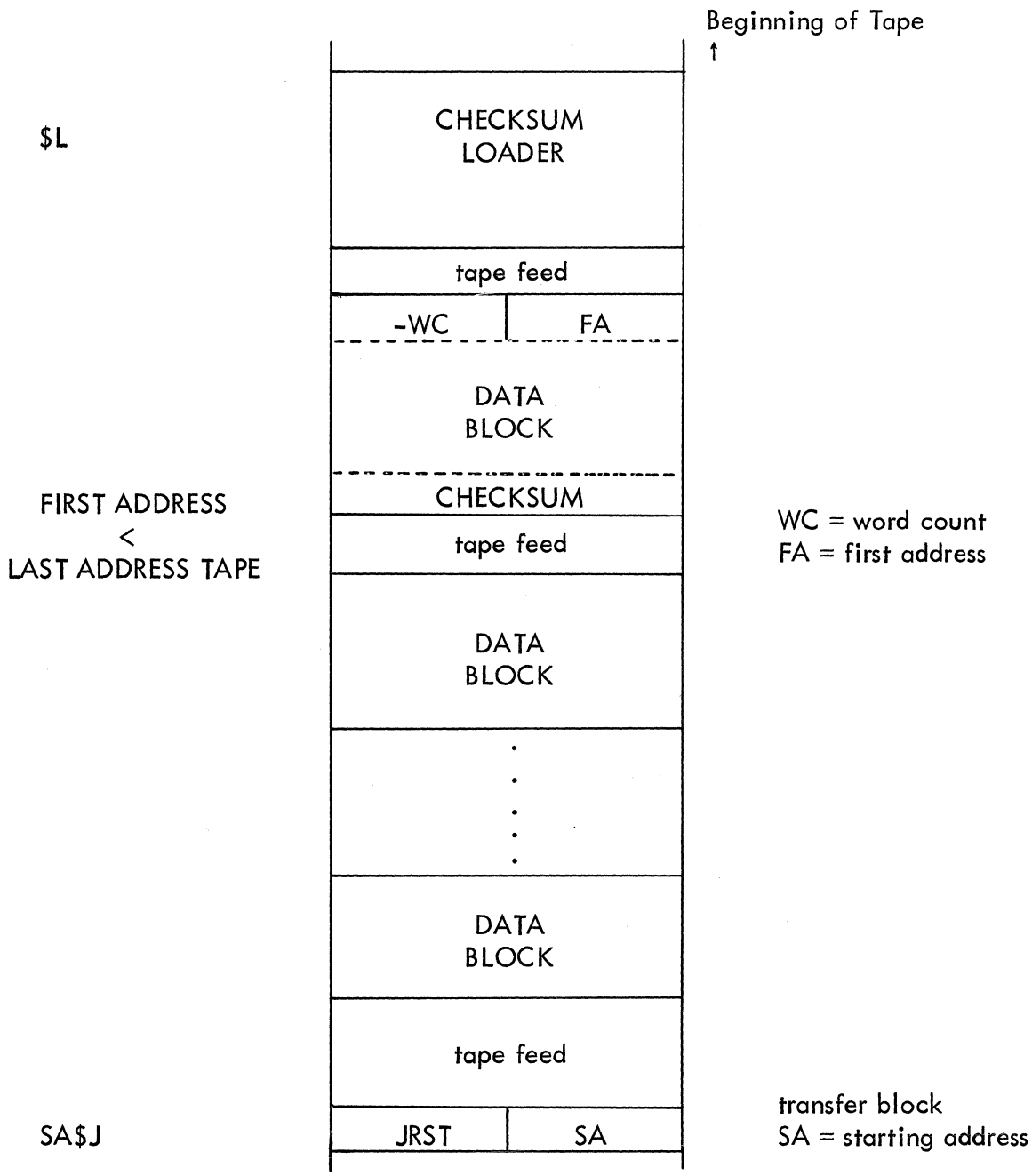

Figure 1 DDT Checksummed Block Format

ADR/MOVEIT,6 ~~TAPE~~ This punches out a checksummed block containing the contents (the new contents if a change has been made) of the register (ADR).

TAPE and ~~TAPE~~ are single control keys on the Teletype.

Yank (Read) and Verify

FIRST<LAST$Y — Read (yank) a tape into core starting at FIRST and up to LAST address.

FIRST<LAST$V — VERIFY a tape with core. That is, check to see if the paper tape just punched matches the information in core memory in those bit positions where the rack has ones. VERIFY the information on paper tape with the information from FIRST to LAST address inclusive.

Teletype Control

A half duplex Teletype should not have the computer echo each character. Two commands are provided to control the echo:

CONTROL N means No, echo
CONTROL Y means Yes, echo

Error Messages

If any undefined symbol that cannot be assembled is typed, U will be typed back. If an illegal control command is given, ? will be typed back. The RUBOUT key will "erase" the last word or part of a command and type XXX. After DDT has finished typing, the correct data word or command may be retyped.

Miscellaneous

$Q — Q represents the value of the last quantity typed.

Entering and Leaving DDT

When control is transferred to DDT, the state of the machine is saved inside DDT:

1. The accumulators are saved.

*2. The status of the priority interrupt system (the result of a CONI PI, $I) is stored in the right half of register $I.

---

*Any commands that are preceded by an asterisk are not available in the time sharing user mode.

3. The central processor flags are saved in the left half of register $I.

*4. The PI channels (Teletype control register) are turned off (by a CONO PI, @ $I+1) if they have a bit in register $I+1.

*5. The Teletype PI channel is saved in the right half of register $I+2. The Teletype buffer is saved in the left half of $I+2 but can never be restored. The character in the output buffer will have been typed on the Teletype.

When the execution of a program is restarted, the following happens:

1. The accumulators are restored.

*2. Those PI channels which were on (when DDT was entered) and which have a bit equal to 1 in register $I+1 are turned on.

$C(\$I)_R \wedge C(\$I+1)_R \vee 2000 \rightarrow$ PI SYSTEM

(logical AND ($\wedge$), logical OR ($\vee$))

*3. The Teletype PI channel is restored.

0 → TTI DONE → TTI BUSY → TTO BUSY

TTO done is set to 1 if either TTO busy or TTO done was on when DDT was entered. Otherwise 0 → TTO done.

4. The processor flags are restored from the left half of register $I. The PC change flag will be set.

---

*Any commands that are preceded by an asterisk are not available in the time sharing user mode.

APPENDIX 1

SUMMARY OF COMMANDS

Command	Action
/	When immediately following the address of a register, slash causes the register to be opened and its contents typed. The register examination pointer (REP) is set equal to the address of the register. Following a register printout, slash will cause the contents of the preceding addressed register to be typed.
!	Inhibits the typeout of the contents of registers until [,], or / is typed. Otherwise, it works the same as the forward slash (/).
[	Same effect as slash (/) above except that it forces the typeout to be in numbers of the current radix.
]	The right bracket has the same effect as the slash except that it forces the typeout to be in symbolic instructions.
\	Backslash opens the address last typed and types out the contents of the addressed register. It causes the contents of the opened address to be modified.
line-feed	Causes a carriage return, then adds one to the current address and types out the new address and its contents. REP is set equal to the new address.
↑	The vertical arrow key causes a carriage return-line-feed, subtracts one from the current address, and types the resultant address and its contents. REP is set equal to the new address.
horizontal tab (→|)	Causes a carriage return and sets the REP to the address of the instruction in the register just closed. DDT-6 types this new address and its contents.

Command	Action
Carriage Return	Causes all temporary typeout modes to return to permanent mode and makes the modification, if any, to the currently opened address.
$S($$S)	Sets the mode in which DDT types out words in symbolic instructions temporarily or permanently depending on the number of dollar signs.
$H($$H)	Changes the typeout mode to half words in the current radix.
$C($$C)	Changes the typeout mode to constants, numbers of the current radix.
$T($$T)	Changes the typeout mode to the ASCII character set.
$F($$F)	Changes the typeout mode to floating point.
$R($$R)	Addresses will be typed out with a symbol plus a constant if the constant is less than 64 and if the address is greater than 48; otherwise addresses will be constants in the prevailing radix.
$A($$A)	Addresses will be constants in the prevailing radix.
N$R(M$$R)	Any output radix (≥ 2) may be set by typing N as the radix desired.
←	Left arrow forces DDT to retype the last quantity in the symbolic instruction or half word format depending on the mode.
=	Equal sign forces DDT to retype the last quantity as a constant of the current radix.
&	Ampersand retypes the last quantity as a radix 50 symbol.
"	Double quote is used to insert ASCII characters into memory registers. 1. Insert single right justified character, type double quote, the desired character, and then hit the ALT MODE key.

Command	Action
" (cont)	2. Left justified characters are inserted by typing double quote, some terminating character, the desired characters to be inserted and finally the terminating character (up to five characters can be inserted).
a<b>c$	a is the lower limit of a search; b is the upper limit, and c is the quantity searched for. After specifying these parameters, either W, N, or E is appended to the dollar sign.
W	Word search compares each storage word with the word being searched for in those bit positions where the mask has ones. If the comparison shows an equality, the word search types out the address and the contents of the register.
N	Not-word search is the same as above except that if the comparison shows an inequality, the register and its contents would be typed out.
E	The effective address search will find and type out all locations where the effective address, following all indirect and index-register chains to a depth of 64 levels, equals the address being searched for.
NUM$M	NUM will be inserted into the mask register.
$M/	Types out the contents of the mask register and is now open.
ADR(EXAM)$nB	Inserts breakpoint n at address ADR and examines address EXAM when the break occurs.
WORD$B	Places a breakpoint in the next available (one of eight) DDT breakpoint register.
$B	Removes all breakpoints.
0$nB	Removes the nth breakpoint.

Command	Action
N$P	The user's program will continue from the point where the break occurred and set the proceed counter of DDT's breakpoint registers to N.
$nB/	Address of breakpoint in right half. Address of register to examine (or zero) in left half.
$nB+1/	Conditional break instruction, or 0.
$nB+2/	Proceed counter.
ADR$G	Starts executing instructions at address ADR.
INSTRUC$X	Executes the expression INSTRUC.
NUM<SYMBOL;	Defines SYMBOL to have a value NUM.
TAG:	Defines TAG to have a value equal to the address of the last opened register.
SYMBOL$$K	Symbol will be killed in DDT symbol table.
$$K	Kills all symbols in DDT symbol table.
SYMBOL$K	Prevents DDT from using this symbol for typeout.
$$Z	Zeros core except DDT and registers 20-37 (20-137 in time sharing user mode) which have special usage.
a<b $$Z	Zeros core between a and b registers inclusively.
UNDEF#	The # symbol is attached to all currently undefined symbols.
?	Types out all undefined symbols.
$L	Punches a check summing loader on paper tape.

Command	Action
a<b TAPE	Punches a paper tape with information from memory between a and b inclusively.
ADR$J	Punches on paper tape a 1-word checksummed block that is a transfer instruction to address ADR after the preceding program on the paper tape is loaded.
a<b $Y	Reads a checksummed paper tape into memory between a and b addresses inclusively.
a<b $V	Verifies a checksummed paper tape with memory between a and b inclusively.

EQUIPMENT
CORPORATION
MAYNARD, MASSACHUSETTS

PRINTED IN U.S.A.

5-8/65