SECTION 1

GENERAL INFORMATION

1.1 INTRODUCTION

The assembler translates a symbolic 8080 assembly language program ("source code") into the binary instructions ("object code") required by the computer to execute the program.

The assembler can operate on two different kinds of source code files, both of which can be generated by the editors supplied with PTDOS:

Text Files:	Each line of a normal text file consists of the characters of that line followed by a carriage return (0DH).
ALS-8 Text Files:	Each line of an ALS-8 format file consists of a one-byte character count, a four-byte line number field (which may be blank), the characters of the line, and a carriage return (0DH).

ALS-8 files need never be used unless programs are exchanged with an ALS-8 system.

When the assembler is invoked, it is loaded into memory starting at location 100H. It processes the source code file in two passes. On the first pass, it builds a symbol table containing all of the labels defined in the source program. (See Section 2.3.) The symbol table begins at the memory location immediately following the assembler; each entry in the table is 7 bytes long. Certain errors may be detected during the first pass, causing error messages to be output to an error file (usually the console). On the second pass, the object code is generated and usually output to an object code file. In addition, a formatted listing of both source and object code may be output to a listing file, and symbol and cross-reference table listings may be output to a possibly different file. Any errors detected during this pass cause messages to be output to the error file.

To abort the assembly process at any time, press the MODE key (or CTRL-@) on the keyboard.

If the assembly runs to completion and no errors are detected, the resulting object code file is an image file that you can execute by typing its name as a PTDOS command (if the source code contained an XEQ pseudo-operation - see Section 3). To load the file without executing it, type its name followed by a comma.

Example:

```
*SPELL   loads and executes a file called SPELL.
*SPELL,  loads SPELL but does not execute it.
```

1.2 ASSEMBLER COMMAND FORMAT

The assembler is invoked by a PTDOS command with the following format. The square brackets [] surround optional arguments; the angle brackets <> surround the generic name of an item to be typed, e.g., <name> means "type a name."

ASSM <source>[,<list>,<object>,<error>,<symbol>,<S=options>]

<source> The name of the source code input file. This parameter must be present; all others are optional.

<list> The name of the listing output file. If this argument is absent, no listing is generated. If the specified file does not exist, it is created with type '.' and block size 4CØH.

<object> The name of the object code output file. If this argument is absent, no object code is generated. If the specified file does not exist, it is created with type 'I.' and block size 1ØØH.

<error> The name of the file to which lines containing errors are written. (All lines, including those containing errors, are written to <list>, if that argument is present.) Default for <error> is the console file (file #1), unless <list> is #1, in which case there is no default <error> file.

<symbol> The name of the file to which symbol and cross reference tables are written. If this argument is absent, no symbol or cross reference table is generated. If an equals sign (=) is used instead of a file name, the table is written to the same file as the listing.

S=<options> Various assembler options may be controlled by following the S= with one or more of the following option specifiers. The list of options is terminated by a comma or carriage return. For those options that may be preceded by a + or -, the + is optional and will be assumed if absent.

```
+A   The source file is in ALS-8 format.
-A   The source file is a normal text file.
```

If neither of these is specified, the assembler attempts to determine the file type by examining the first few lines. If it fails, an error message is generated. The appropriate option then must be specified in order to assemble the program successfully.

+L The source file has line numbers in column 1-4 of each line.

-L The source file has no line numbers.

If neither of these is specified, the assembler will examine the first few lines to determine if the file has line numbers.

Instructs the assembler to generate its own line numbers in the listing in place of those in the source file (if any).

Ø,1,2 or 3 Specifies the spacing on the listing:

Ø = no additional spacing
1 = 72 column output
2 = 8Ø column output (default)
3 = 132 column output

P Instructs the assembler to paginate output to the listing file. The disk name and file name of the source code file will be printed on the top left-hand corner of each page. A page number and the date from the system global area will be printed on the top right-hand corner of each page. If a TITL pseudo-operation occurs in the source code, a one- or two-line title will be centered at the top of each page.

X Instructs the assembler to output a cross-reference table to the symbol file (if one was specified in the ASSM command). The source code must either contain line numbers or be assigned line numbers by the assembler (see # option, above).

The file name arguments are positional, e.g., the first is always <source>, the second always <list>, etc. Any except <source> may be omitted with the effect described above. If one of the intermediate arguments is to be omitted while a subsequent argument is present, an extra comma must be used to hold its place. The S=<options> argument may occur at any position in the list of arguments, since it is keyword delimited.

If the symbol table or cross reference listing is to be output to the listing file, an equals sign (=) must be used in place of the symbol file name in the ASSM command. If the same file name (other than the console output file #1) appears more than once in an ASSM command, a PTDOS error will result.

Example 1: ASSM TEST.S,TEST.L,TEST,TEST.E,TEST.SYM,S=P1

The source code file TEST.S will be assembled with the listing output to the file TEST.L, and the object code output to TEST. The listing will be paginated with the spacing format set to 72 columns. If any errors are found during the assembly, the line(s) containing the error(s) will be written to the file TEST.E. A symbol table for the program will be written to the file TEST.SYM.

Example 2: ASSM TEST.S,#1

The source code file TEST.S will be assembled and the listing will be output to the console output file #1. No object code file will be generated. The output listing will be unpaginated and formatted to 80 columns (by default).

Example 3: ASSM TEST.S,,TEST

The source code file TEST.S will be assembled and the object code file written to TEST. No listing file will be generated, but any lines containing errors will be output to the console (by default).

SECTION 2

STATEMENTS

2.1 INTRODUCTION

An assembly language program (source code) is a series of statements specifying the sequence of machine operations to be performed by the program.

Each statement resides on a single line and may contain up to four fields as well as an optional line number. These fields, label, operation, operand and comment, are scanned from left to right by the assembler, and are separated by spaces.

2.2 LINE NUMBERS

Line numbers in the range 0000-9999 may appear in columns 1-4. Line numbers need not be ordered and have no meaning to the assembler, except that they appear in a cross reference listing; if a source code file has no line numbers and a cross reference listing is desired, S=# must be specified in the ASSM command. Line numbers may also make it easier to locate lines in the source code file when it is being edited. The disk and memory space required for normal text files will be increased by five bytes per line if line numbers are used; this may become significant for large files. (ALS-8 format text files have space allocated for line numbers, whether or not line numbers are used.)

If line numbers are not used, the label field starts in column 1 and the operation field may not start before column 2. If line numbers are used, they must be followed by at least one space, so the label field starts in column 6 and the operand field may not start before column 7.

Once the starting column for the label has been established, the same format must be followed throughout the file: either all of the lines or none of the lines can have line numbers. Any other file(s) assembled along with the main file (using the COPY pseudo-operation) must conform to the format of the main file.

Example of source statements with line numbers:

```
column
1234567
-------
0000 LABEL ORA A   Label field must start at column 6.
0001  JNZ NEXT     Operation field starts at column 7 (minimum).
0002 LOOP MOV A,B  Operation field starts one space after label.
```

Example of source statements without line numbers:

```
column
1234567
-------
LABEL ORA A         Label field must start at column 1.
 JNZ NEXT           Operation field starts at column 2 (minimum).
LOOP MOV A,B        Operation field starts one space after label.
```

2.3 LABEL FIELD

The label field must start in column 1 of the line (column 6 if line numbers are used). A label gives the line a symbolic name that can be referenced by any statement in the program. Labels must start with an alphabetic character (A-Z,a-z), and may consist of any number of characters, though the assembler will ignore all characters beyond the fifth; e.g., the labels BRIDGE, BRIDG and BRIDGET cannot be distinguished by the assembler. A duplicate label error will occur if any two labels in a program begin with the same five letters.

A label may be separated from the operation field by a colon (:) instead of, or in addition to, a blank.

The labels A, B, C, D, E, H, L, M, PSW and SP are pre-defined by the assembler to serve as symbolic names for the 8080 registers (see Section 2.5.1). They must not appear in the label field.

An asterisk (*) or semi-colon (;) in place of a label in column 1 (column 6 if line numbers are used) will designate the entire line as a comment line; see Section 2.6.

2.4 OPERATION FIELD

The operation field contains either 8080 instruction mnemonics or assembler pseudo-operation mnemonics. Appendix 1 summarizes the standard instruction mnemonics recognized by the assembler, and Appendix 4 lists several references to consult if more information on the 8080 machine instructions is needed. Assembler pseudo-operations are directives that control various aspects of the assembly process, such as storage allocation, conditional assembly, file inclusion, and listing control. The pseudo-operations are described in Section 3.

An operation mnemonic may not start before column 2 (column 7 if line numbers are used) and must be separated from a label by at least one space (or a colon).

2.5 OPERAND FIELD

Most machine instructions and pseudo-operations require one or two operands, either register names, labels, constants, or arithmetic expressions involving labels and constants.

The operands must be separated from the operator by at least one space. If two operands are required, they must be separated by a comma. No spaces may occur within the operand field, since the first space following the operands delimits the comment field.

2.5.1 Register Names

Many 8080 machine instructions require one or two registers or a register pair to be designated in the operand field. The symbolic names for the general-purpose registers are A, B, C, D, E, H and L. SP stands for the stack pointer, while M refers to the memory location whose address is in the HL register pair. The register pairs BC, DE, and HL are designated by the symbolic names B, D, and H, respectively. The A register and condition flags, when operated upon as a register pair, are given the symbolic name PSW.

The values assigned to the register names A, B, C, D, E, H, L, M, PSW and SP are 7, 0, 1, 2, 3, 4, 5, 6, 6 and 6, respectively. These constants, or any label or expression whose value lies in the range 0 to 7, may be used in place of the pre-defined symbolic register names where a register name is required; such a substitution of a value for the pre-defined register name is not recommended, however,

2.5.2 Labels

Any label that is defined elsewhere in the program may be used as an operand. If a label is used where an 8-bit quantity is required (e.g., MVI C,LABEL), its value must lie in the range -256 to 255, or it will be flagged as a value error.

If a label is used as a register name, its value must lie in the range 0 to 7, or be 0, 2, 4, or 6 if it designates a register pair. Otherwise, it will be flagged as a register error.

During each pass, the assembler maintains an instruction location counter that keeps track of the next location at which an instruction may be stored; this is analogous to the program counter used by the processor during program execution to keep track of the location of the next instruction to be fetched.

The special label $ (dollar sign) stands for the current value of the assembler's instruction location counter. When $ appears within the operand field of a machine instruction, its value is the address of the first byte of the next instruction.

Example:

```
FIRST EQU $              The label FIRST is set to the address
TABLE DB ENTRY           of the first entry in a table and LAST
*                        points to the location immediately after
*                        the end of the table.  TABLN is then
*                        the length of the table and will remain
LAST  EQU $              correct, even if later additions or
TABLN EQU LAST-FIRST     deletions are made in the table.
```

2.5.3 Constants

Decimal, hexadecimal, octal, binary and ASCII constants may be used as operands.

The base for numeric constants is indicated by a single letter immediately following the number, as follows:

D = decimal
H = hexadecimal
O = octal
Q = octal
B = binary

If the letter is omitted, the number is assumed to be decimal. Q is usually preferred for octal constants, since O is so easily confused with Ø (zero). Numeric constants must begin with a numeric character (Ø-9) so that they can be distinguished from labels; a hexadecimal constant beginning with A-F must be preceded by a zero.

ASCII constants are one or two characters surrounded by single quotes ('). A single quote within an ASCII constant is represented by two single quotes in a row with no intervening spaces. For example, the ASCII value of a single quote mark (') is represented by the expression '''', where the two outer quote marks are the delimiters of the ASCII string, and the two inner quote marks represent the string itself, i.e., the single quote character. A single character ASCII constant has the numerical value of the corresponding ASCII code. (Appendix 2 contains a list of ASCII codes.) A double character ASCII constant has the 16-bit value whose high-order byte is the ASCII code of the first character and whose low-order byte is the ASCII code of the second character.

If a constant is used where an 8-bit quantity is required (e.g., MVI C,1ØH), its numeric value must lie in the range -256 to 255 or it will be flagged as a value error.

If a constant is used as a register name, its numeric value must lie in the range Ø to 7, or be Ø, 2, 4, or 6 if it designates a register pair. Otherwise, it will be flagged as a register error.

Examples:

```
MVI A,128       Move 128 decimal to register A.
MVI C,1ØD       Move 1Ø decimal to register C.
LXI H,2FH       Move 2F hexadecimal to register pair HL.
MVI B,3Ø3Q      Move 3Ø3 octal to register B.
MVI A,'Y'       Move the ASCII value for Y to register A.
MVI A,1Ø1B      Move 1Ø1 binary to register A.
JMP ØFFH        Jump to address FF hexadecimal.
```

2.5.4 Expressions

Operands may be arithmetic expressions constructed from labels, constants, and the following operators:

+	addition or unary plus
-	subtraction or unary minus
*	multiplication
/	division (remainder discarded)

Values are treated as 16-bit unsigned 2's complement numbers. Positive or negative overflow is allowed during expression evaluation, e.g., 32767+1=7FFFH+1=8000H=-32768 and -32768-1=8000H-1=7FFFH=32767. Expressions are evaluated from left to right; there is no operator precedence.

If an expression is used where an 8-bit quantity is required (e.g., MVI C,TEMP+10H), it must evaluate to a value in the range -256 to 255, or it will be flagged as a value error.

An expression used as a register name must evaluate to a value in the range 0 to 7, or to 0, 2, 4, or 6 if it designates a register pair. Otherwise, it will be flagged as a register error.

Examples:

```
MVI     A,255D/10H-5
LDA     POTTS/256*OFFSET
LXI     SP,30*2+STACK
```

2.5.5 High- and Low-Order Byte Extraction

If an operand is preceded by the symbol <, the high-order byte of the evaluated expression will be used as the value of the operand. If an operand is preceded by the symbol >, the low-order byte will be used.

Note that the symbols < and > are not operators that may be applied to labels or constants within an expression. If more than one < or > appears within an expression, the rightmost will be used to determine whether to use the high- or low-order byte of the evaluated expression as the value of the operand. That is, the rightmost < or > is treated as if it preceded the entire expression, and the others will be totally ignored.

Examples:

MVI A,>TEST	Loads register A with the least significant 8 bits of the value of the label TEST.
MVI B,<0CC00H	Loads register B with the most significant byte of the 16-bit value CC00H, i.e., CCH.
MVI C,<1234H	Loads register C with the value 12H.
MVI C,>1234H	Loads register C with the value 34H.

2.6 COMMENT FIELD

The comment field must be separated from the operand field (or operation field for instructions or pseudo-operations that require no operand) by at least one space. Comments are not processed by the assembler, but are solely for the benefit of the programmer. Good comments are essential if a program is to be understood very long after it is written or is to be maintained by someone other than its author.

An entire line will be treated as a comment if it starts with an asterisk (*) or semicolon (;) in column 1 (column 6 if line numbers are used).

Examples:

```
LOOP IN  STAT INPUT DEVICE STATUS
     ANI 1    TEST STATUS BIT
     JZ  LOOP WAIT FOR DATA
*DATA IS NOW AVAILABLE
```

If listing file formatting is specified in the ASSM command (S=<options> contains 1, 2, or 3), the comment field must be preceded by at least two spaces to ensure proper output formatting. Furthermore, instructions and pseudo-operations requiring no operand must be followed by a dummy operand (a period is recommended).

Examples:

```
MVI A,10  COMMENT
RZ .  COMMENT
```

SECTION 3

PSEUDO-OPERATIONS

Pseudo-operations appear in a source program as instructions to the assembler and do not always generate object code. This section describes the pseudo-operations recognized by the PTDOS assembler.

In the following pseudo-operation formats, <expression> stands for a constant, label, or arithmetic expression constructed from constants and labels. Optional elements are enclosed in square brackets [].

Equate

<label> EQU <expression>

This pseudo-operation sets a label name to the 16-bit value that is represented in the operand field. That value holds for the entire assembly and may not be changed by another EQU.

Any label that appears in the operand field of an EQU statement must be defined in a statement earlier in the program.

Examples:

```
BELL EQU 7        The value of the label BELL is set to 7.
BELL2 EQU BELL*2  Label BELL2 is set to 7*2.
```

Set Origin

[<label>] ORG <expression>

This pseudo-operation sets the assembler's instruction location counter to the 16-bit value specified in the operand field . In other words, the object code generated by the statements that follow must be loaded beginning at the specified address in order to execute properly. The label, if present, is given the specified 16-bit value.

Any label that appears in the operand field of an ORG statement must be defined in a statement earlier in the program.

If no origin is specified at the beginning of the source code, the assembler will set the origin to 100H. If no ORG pseudo-operation is used anywhere in the source program, successive bytes of object code will be stored at successive memory locations.

Examples:

```
      ORG 64      Determines that the object code generated by
                  subsequent statements must be loaded in locations
                  beginning at 64 (40H).
START ORG 100H    Determines that the object code generated by
                  subsequent statements must be loaded in locations
                  beginning at 100H.
```

Set Execution Address XEQ <expression>

This pseudo-operation specifies the entry point address for the program, i.e., the address at which it is to begin execution. If a program contains no XEQ pseudo-operation, the object code image file will contain no start address; if its name is typed as a PTDOS command, it will be loaded but not executed (exactly as if its name were followed by a comma). If more than one XEQ appears in a program, the last will be used.

An example of the difference between ORG and XEQ is that a program whose first 100 bytes are occupied by data will have an ORG address 100 bytes lower in memory than its XEQ address.

Example:

```
100H        The entry point address for the assembled program
                 is set to 100H.
```

Define Storage [<label>] DS <expression>
                [<label>] RES <expression>

Either of these pseudo-operations reserves the specified number of successive memory locations starting at the current address within the program. The contents of these locations are not defined and are not initialized at load time.

Any label that appears in the operand field of a DS or RES statement must be defined in a statement earlier in the program.

Examples:

```
SPEED DS 1      Reserve one byte.
 DS 400         Reserve 400 bytes.
 RES 177Q       Reserve 177 (octal) bytes.
```

Define Byte [<label>] DB <expression>[,<expression>,...]

This pseudo-operation sets a memory location to an 8-bit value. If the operand field contains multiple expressions separated by commas, the expressions will define successive bytes of memory beginning at the current address. Each expression must evaluate to a number that can be represented in 8 bits.

Examples:

```
 DB 1                                    One byte is defined.
 DB 0FFH,303Q,100D,11010011B,3*BELL,-10  Multiple bytes are defined.
TABLE DB 'A','B','C','D',0               Multiple bytes are defined.
```

Define Word [<label>] DW <expression>

This pseudo-operation sets two memory locations to a 16-bit quantity. The least significant (low-order) byte of the value is stored at the current address and the most significant byte (high-order) is stored at the current address + 1.

Examples:

```
SAVE DW 1234H   1234H is stored in memory, 34H in the low-order
                byte and 12H in the high-order byte.
YES DW 'OK'     The ASCII value for the letters 'O' and 'K' is
                stored with the 'K' at the lower memory address.
```

Define Double Byte [<label>] DDB <expression>

This pseudo-operation is almost the same as DW, except that the two bytes are stored in the opposite order: high-order byte first, followed by the low-order byte.

Example:

```
FIRST DDB 1234H  1234H is stored in memory, 12H in the low-order
                 byte and 34H in the high-order byte.
```

Define ASCII String [<label>] ASC #<ASCII string>#
[<label>] ASCZ #<ASCII string>#

The ASC pseudo-operation puts a string of characters into successive memory locations starting at the current location. The special symbols # in the format are "delimiters;" they define the beginning and end of the ASCII character string. The assembler uses the first non-blank character found as the delimiter. The string immediately follows this delimiter, and ends at the next occurrence of the same delimiter, or at a carriage return.

The ASCZ pseudo-operation is the same except that it appends a NUL (ØØH) to the end of the stored string.

Examples:

```
WORDS ASC "THIS IS AN ASCII STRING"
      ASCZ "THIS IS ANOTHER STRING"
```

Set ASCII List Flag ASCF Ø
ASCF 1

If the operand field contains a Ø, the listing of the assembled bytes of an ASCII string will be suppressed after the first line (four

bytes). Likewise, only the first four assembled bytes of a DB pseudo-operation with multiple arguments will be listed. If a program contains many long strings, its listing will be easier to read if the ASCF pseudo-operation is used.

If the operand field contains a 1, the assembled form of subsequent ASCII strings and DB pseudo-operations with multiple arguments will be listed in full. This is the default condition.

See Appendix 3 for an example of the listing format.

Conditional Assembly IF <expression>
.
source code
.
ENDF

The value of the expression in the operand field governs whether or not subsequent code up to the matching ENDF will be assembled. If the expression evaluates to a Ø (false), the code will not be assembled. If the expression evaluates to a non-zero value (true), the code will be assembled. Blocks of code delimited by IF and ENDF ("conditional code") may be nested within another block of conditional code.

Any label that appears in the operand field of an IF...ENDF pseudo-operation must be defined in a statement earlier in the program.

```
YES EQU 1       Sets the value of the label 'YES' to 1.
NO EQU Ø        Sets the value of the label 'NO' to Ø.
*
 IF YES          The expression here is true (1), so the
 MVI A,'Y'       code on this line will be assembled.
 IF NO           The expression here is false (Ø), so the code
 MVI A,'N'       on this line will not be assembled.
 ENDF            This terminates the NO conditional.
 ENDF            This terminates the YES conditional.
```

List Conditional Code IFLS

This pseudo-operation enables listing of conditional source code even though no object code is being generated because of a false IF condition. The assembler will not list such conditional source code if this pseudo-operation is not used.

Copy File COPY <file name>[/<unit>]

This pseudo-operation copies source code from a disk file into a program being assembled. The code from the copied file will be assembled starting at the current address. When the copied file is exhausted, the assembler will continue to assemble from the original

file. The resulting object code will be exactly like what would be generated if the copied source code were part of the original file, but the COPY pseudo-operation does not actually alter any source file.

A copied file may in turn copy another, as long as no more than six files are active at any given time. Note that one file must not copy another which in turn copies the original file--the assembly will generate duplicate label errors and assemble the same code over and over until the object code file overflows the diskette on which it is being written.

All files that are accessed by the COPY pseudo-operation must be of the same format as the main source file, i.e., either ALS-8 format or normal text files, and either having or not having line numbers.

Listing Control NLST
LST

The NLST pseudo-operation suppresses all output to the listing file. Object code will still be output to the object code file and lines containing errors will still be output to the error file. The LST pseudo-operation re-enables output to the listing file.

Listing Title TITL <first line>"<second line>

If the P option is specified in the ASSM command, the one- or two-line title specified by this pseudo-operation will be printed centered at the top of each page of the listing.

Page Eject PAGE

If the P option is specified in the ASSM command, this pseudo-operation causes a skip to the top of the next page of the listing.

End of Source File END

This pseudo-operation terminates each pass of the assembly. Only one END statement should be in the file or files to be assembled, and it should be the last statement encountered by the assembler. Since an end-of-file on the source code input file will also terminate each pass, the END statement is unnecessary in most cases.

SECTION 4

ERROR MESSAGES

4.1 ASSEMBLER COMMAND ERRORS

A number of console messages may be generated in response to errors in the ASSM command. When an error of this sort occurs, the assembly is aborted and control returns to PTDOS.

BAD ALS8 FILE STRUCTURE	The ALS-8 source code input file has a structure defect. The most likely cause of this error is trying to assemble a very short non-ALS-8 source code file without specifying S=-A.
EXPECTED NAME	The source code input file name is missing.
ILLEGAL FILE NAME	A file name contains illegal characters. This message can also be generated by an erroneous file name in a COPY pseudo-operation.
ILLEGAL FILE NUMBER	A file ID is too large (>255).
ILLEGAL OPTION SPECIFIER	An unrecognized option specifier follows S=.
SYNTAX ERROR	The ASSM command contains too many arguments or an argument of the form X=... where X is a letter other than S.

In addition to the above special messages, several standard error messages may be generated by the PTDOS file system.

4.2 ASSEMBLY ERRORS

If a statement contains one of the following errors, there will be a single letter error code in column 19 of the line output to the listing and/or error files. An error detected during both the first and the second pass of the assembler will be flagged twice in the listing(s). If the error is not an opcode error, NULs will be output as the second and, if appropriate, third bytes of object code for that instruction. If the error is an opcode error, the instruction will be assumed to be a three-byte instruction, and three NULs will be written to the listing and/or error files. The error codes are:

A	ARGUMENT ERROR	An illegal label or constant appears in the operand field. This might be 1) a number with a letter in it, e.g., 2L, 2) a label that starts with a number, e.g., 3STOP, or 3) an improper representation of a string, e.g., '''A''' in the operand field of a statement containing the ASCII pseudo-operation.
D	DUPLICATE LABEL	The source code contains multiple labels whose first five characters are identical
L	LABEL ERROR	The symbol in the label field contains illegal characters, e.g., it starts with a number.
M	MISSING LABEL	An EQU instruction does not have a symbol in the label field.
O	OPCODE ERROR	The symbol in the operation field is not a valid 8080 instruction mnemonic or an assembler pseudo-operation mnemonic.
R	REGISTER ERROR	An expression used as a register designator does not have a legal value.
S	SYNTAX ERROR	A statement is not in the format required by the assembler.
U	UNDEFINED SYMBOL	A label used in the operand field is not defined, i.e., does not appear in the label field anywhere in the program, or is not defined prior to its use as an operand in an EQU, ORG, DS, RES, or IF pseudo-operation.
V	VALUE ERROR	The value of the operand lies outside the allowed range.

APPENDIX 2
TABLE OF ASCII CODES (Zero Parity)

Paper tape 1 2 3 . 4 5 6 7 P	Upper Octal	Octal	Decimal	Hex	Character		
.	0000	000	0	00	ctrl @	NUL	
• .	0004	001	1	01	ctrl A	SOH	Start of Heading
• .	0010	002	2	02	ctrl B	STX	Start of Text
•• .	0014	003	3	03	ctrl C	ETX	End of Text
• .	0020	004	4	04	ctrl D	EOT	End of Xmit
• • .	0024	005	5	05	ctrl E	ENQ	Enquiry
•• .	0030	006	6	06	ctrl F	ACK	Acknowledge
••• .	0034	007	7	07	ctrl G	BEL	Audible Signal
. •	0040	010	8	08	ctrl H	BS	Back Space
• . •	0044	011	9	09	ctrl I	HT	Horizontal Tab
• . •	0050	012	10	0A	ctrl J	LF	Line Feed
•• . •	0054	013	11	0B	ctrl K	VT	Vertical Tab
• . •	0060	014	12	0C	ctrl L	FF	Form Feed
• • . •	0064	015	13	0D	ctrl M	CR	Carriage Return
•• . •	0070	016	14	0E	ctrl N	SO	Shift Out
••• . •	0074	017	15	0F	ctrl O	SI	Shift In
. •	0100	020	16	10	ctrl P	DLE	Data Line Escape
• . •	0104	021	17	11	ctrl Q	DC1	X On
• . •	0110	022	18	12	ctrl R	DC2	Aux On
•• . •	0114	023	19	13	ctrl S	DC3	X Off
• . •	0120	024	20	14	ctrl T	DC4	Aux Off
• • . •	0124	025	21	15	ctrl U	NAK	Negative Acknowledge
•• . •	0130	026	22	16	ctrl V	SYN	Synchronous File
••• . •	0134	027	23	17	ctrl W	ETB	End of Xmit Block
. ••	0140	030	24	18	ctrl X	CAN	Cancel
• . ••	0144	031	25	19	ctrl Y	EM	End of Medium
• . ••	0150	032	26	1A	ctrl Z	SUB	Substitute
•• . ••	0154	033	27	1B	ctrl [	ESC	Escape
• . ••	0160	034	28	1C	ctrl \	FS	File Separator
• • . ••	0164	035	29	1D	ctrl]	GS	Group Separator
•• . ••	0170	036	30	1E	ctrl ^	RS	Record Separator
••• . ••	0174	037	31	1F	ctrl _	US	Unit Separator
. •	0200	040	32	20	Space		
• . •	0204	041	33	21	!		
• . •	0210	042	34	22	"		
•• . •	0214	043	35	23	#		
• . •	0220	044	36	24	$		
• • . •	0224	045	37	25	%		
•• . •	0230	046	38	26	&		
••• . •	0234	047	39	27	'		
. • •	0240	050	40	28	(		
• . • •	0244	051	41	29	)		
• . • •	0250	052	42	2A	*		
•• . • •	0254	053	43	2B	+		
• . • •	0260	054	44	2C	,		
• • . • •	0264	055	45	2D	-		
•• . • •	0270	056	46	2E	.		
••• . • •	0274	057	47	2F	/		
. ••	0300	060	48	30	0		
• . ••	0304	061	49	31	1		
• . ••	0310	062	50	32	2		
•• . ••	0314	063	51	33	3		
• . ••	0320	064	52	34	4		
• • . ••	0324	065	53	35	5		
•• . ••	0330	066	54	36	6		
••• . ••	0334	067	55	37	7		
. •••	0340	070	56	38	8		
• . •••	0344	071	57	39	9		
• . •••	0350	072	58	3A	:		
•• . •••	0354	073	59	3B	;		
• . •••	0360	074	60	3C	<		
• • . •••	0364	075	61	3D	=		
•• . •••	0370	076	62	3E	>		
••• . •••	0374	077	63	3F	?		

APPENDIX 2
TABLE OF ASCII CODES *(Cont'd)* (Zero Parity)

Paper tape 1 2 3 . 4 5 6 7 P	Upper Octal	Octal	Decimal	Hex	Character	
. . . . ●	0400	100	64	40	@	
● . . . . . . ●	0404	101	65	41	A	
. ● . . . . . ●	0410	102	66	42	B	
● ● . . . . . ●	0414	103	67	43	C	
. . ● . . . . ●	0420	104	68	44	D	
● . ● . . . . ●	0424	105	69	45	E	
. ● ● . . . . ●	0430	106	70	46	F	
● ● ● . . . . ●	0434	107	71	47	G	
. . . ● . . . ●	0440	110	72	48	H	
● . . ● . . . ●	0444	111	73	49	I	
. ● . ● . . . ●	0450	112	74	4A	J	
● ● . ● . . . ●	0454	113	75	4B	K	
. . ● ● . . . ●	0460	114	76	4C	L	
● . ● ● . . . ●	0464	115	77	4D	M	
. ● ● ● . . . ●	0470	116	78	4E	N	
● ● ● ● . . . ●	0474	117	79	4F	O	
. . . . ● . . ●	0500	120	80	50	P	
● . . . ● . . ●	0504	121	81	51	Q	
. ● . . ● . . ●	0510	122	82	52	R	
● ● . . ● . . ●	0514	123	83	53	S	
. . ● . ● . . ●	0520	124	84	54	T	
● . ● . ● . . ●	0524	125	85	55	U	
. ● ● . ● . . ●	0530	126	86	56	V	
● ● ● . ● . . ●	0534	127	87	57	W	
. . . ● ● . . ●	0540	130	88	58	X	
● . . ● ● . . ●	0544	131	89	59	Y	
. ● . ● ● . . ●	0550	132	90	5A	Z	
● ● . ● ● . . ●	0554	133	91	5B	[	shift K
. . ● ● ● . . ●	0560	134	92	5C	\	shift L
● . ● ● ● . . ●	0564	135	93	5D	]	shift M
. ● ● ● ● . . ●	0570	136	94	5E	^	shift N
● ● ● ● ● . . ●	0574	137	95	5F	_	shift O
. . . . . ● ●	0600	140	96	60	`	
● . . . . ● ●	0604	141	97	61	a	
. ● . . . ● ●	0610	142	98	62	b	
● ● . . . ● ●	0614	143	99	63	c	
. . ● . . ● ●	0620	144	100	64	d	
● . ● . . ● ●	0624	145	101	65	e	
. ● ● . . ● ●	0630	146	102	66	f	
● ● ● . . ● ●	0634	147	103	67	g	
. . . ● . ● ●	0640	150	104	68	h	
● . . ● . ● ●	0644	151	105	69	i	
. ● . ● . ● ●	0650	152	106	6A	j	
● ● . ● . ● ●	0654	153	107	6B	k	
. . ● ● . ● ●	0660	154	108	6C	l	
● . ● ● . ● ●	0664	155	109	6D	m	
. ● ● ● . ● ●	0670	156	110	6E	n	
● ● ● ● . ● ●	0674	157	111	6F	o	
. . . . ● ● ●	0700	160	112	70	p	
● . . . ● ● ●	0704	161	113	71	q	
. ● . . ● ● ●	0710	162	114	72	r	
● ● . . ● ● ●	0714	163	115	73	s	
. . ● . ● ● ●	0720	164	116	74	t	
● . ● . ● ● ●	0724	165	117	75	u	
. ● ● . ● ● ●	0730	166	118	76	v	
● ● ● . ● ● ●	0734	167	119	77	w	
. . . . ● ● ●	0740	170	120	78	x	
● . . . ● ● ●	0744	171	121	79	y	
. ● . . ● ● ●	0750	172	122	7A	z	
● ● . . ● ● ●	0754	173	123	7B	{	
. . ● . ● ● ●	0760	174	124	7C	|	
● . ● . ● ● ●	0764	175	125	7D	}	Alt Mode
. ● ● . ● ● ●	0770	176	126	7E	~	Prefix
● ● ● . ● ● ●	0774	177	127	7F	DEL	Rubout

APPENDIX 3

ASSEMBLER LISTING

```
ADDRESS  ASSEMBLED CODE     ERROR FLAG  LINE NO.  LABEL   OPERATION  OPERAND     COMMENT

                            0000 *
                            0001 *SEARCH TABLE FOR MATCH TO STRING
                            0002 *EACH TABLE ENTRY IS FOLLOWED BY A TWO-BYTE DISPATCH ADDRESS.
                            0003 *TABLE MUST HAVE AT LEAST ONE ENTRY AND IS TERMINATED BY A
                            0004 *ZERO BYTE.
                            0005 *ON ENTRY:  HL POINTS TO STRING
                            0006 *           DE POINTS TO TABLE
                            0007 *           C  IS NUMBER OF CHARACTERS IN TABLE ENTRIES
                            0008 *ON RETURN: ZERO FLAG SET IF NO MATCH, ELSE DE POINTS TO
                            0009 *           DISPATCH ADDRESS
                            0010 *
0100 E5                     0011 TSRCH  PUSH   H      SAVE STRING ADDRESS
0101 41                     0012        MOV    B,C    INITIALIZE CHARACTER COUNT
0102 1A                     0013 TS1    LDAX   D      COMPARE CHARACTERS
0103 BE                     0014        CMP    M
0104 C2 11 01               0015        JNZ    TS3
0107 23                     0016        INX    H      CHARACTERS MATCH, GO ON TO NEXT
0108 13                     0017        INX    D
0109 05                     0018        DCR    B
010A C2 02 01               0019        JNZ    TS1
010D F6 01                  0020        ORI    1      MATCHING ENTRY FOUND
010F E1                     0021 TS2    POP    H
0110 C9                     0022        RET
0111 B7                     0023 TS3    ORA    A      TEST FOR END OF TABLE
0112 CA 0F 01               0024        JZ     TS2
0115 13                     0025 TS4    INX    D      SKIP TO NEXT ENTRY
0116 05                     0026        DCR    B
0117 C2 15 01               0027        JNZ    TS4
011A 13                     0028        INX    D
011B 13                     0029        INX    D
011C E1                     0030        POP    H
011D C3 00 01               0031        JMP    TSRCH
                            0032 *
                            0033 *EXAMPLE OF TSRCH USE:
                            0034 *
                            0035 *(ASSUME HL POINTS TO A FOUR-CHARACTER COMMAND STRING)
0120 11 35 01               0036        LXI    D,CTABL  DE POINTS TO COMMAND TABLE
0123 0E 04                  0037        MVI    C,4    TABLE ENTRIES ARE FOUR CHARACTERS LONG
0125 CD 00 01               0038        CALL   TSRCH
0128 CA 00 00      U        0039        JZ     ERROR  COMMAND NOT IN TABLE
012B EB                     0040        XCHG   .      SET UP STACK FOR RETURN TO MAIN ROUTINE
012C 11 00 00      U        0041        LXI    D,COMMAND
012F D5                     0042        PUSH   D
0130 7E                     0043        MOV    A,M    DISPATCH TO APPROPRIATE COMMAND ROUTINE
0131 23                     0044        INX    H
0132 66                     0045        MOV    H,M
0133 6F                     0046        MOV    L,A
0134 E9                     0047        PCHL
                            0048 *
                            0049 *COMMAND TABLE
                            0050 *
0135 43 4F 4D 31            0051 CTABL  ASC    'COM1'  FIRST ENTRY
0139 00 00         U        0052        DW     SUB1   ADDRESS OF SUB1
013B 43 4F 4D 32            0053        ASC    'COM2'  SECOND ENTRY
013F 00 00         U        0054        DW     SUB2   ADDRESS OF SUB2
0141 00                     0055        DB     0      END OF TABLE MARK
```

SYMBOL TABLE LISTING

Label	Addr.	Label	Addr.	Label	Addr.	Label	Addr.
CTABL	0135	TS1	0102	TS2	010F	TS3	0111
TS4	0115	TSRCH	0100				

CROSS REFERENCE LISTING

(Printed in place of Symbol Table Listing if X option is specified.)

Label	Addr.	References
CTABL	0135	0092
TS1	0102	0131
TS2	010F	0192
TS3	0111	0239
TS4	0115	0307
TSRCH	0100	0367 0374

APPENDIX 4

REFERENCES

1. 8080/8085 Assembly Language Programming Manual (Intel Corporation, Santa Clara, CA., 1977), Order Number 9800301B

2. Leventhal, Lance A., 8080A/8085 Assembly Language Programming (Adam Osborne & Associates, Berkeley, CA., 1978)

DEBUG

A DEBUGGER FOR PTDOS 1.5

TABLE OF CONTENTS

SECTION 1

INTRODUCTION

1.1 GENERAL INFORMATION

This program is an aid for debugging a machine language program developed and assembled on a Helios-based microcomputer system using PTDOS. With DEBUG, the user is permitted to set as many as fifteen "breakpoints" in memory. (In general, these breakpoints will be set within a program that requires debugging, although the debugger may also be used to examine a program about whose operation the user is simply curious.) When the program containing the breakpoints is executed under the control of DEBUG, it will stop at each of these addresses so that CPU registers, flags, and specified memory locations may be examined and modified. It is possible to resume execution at a breakpoint (or at another specified memory location) after modifications have been made. At the conclusion of debugging, the modified version of the user program may be saved from memory with the PTDOS IMAGE command or the SOLOS/CUTER SAVE command. (The IMAGE command will write to a disk file; the SAVE command, to a tape file.)

There are two versions of the DEBUG program on the PTDOS system disk.

DEBUG3 is loaded and executed at 3000H.
DEBUG is loaded and executed at 5000H.

The two versions are identical, except that they run at different memory locations. The purpose of having two versions is to allow the debugging of a program that uses memory space required by one or the other of the DEBUG programs. DEBUG and DEBUG3 both use a little more than 4K of memory and may be restarted at the addresses at which they are loaded. (Hereafter, DEBUG or "the debugger" will be used to designate either version of the program.)

The DEBUG program contains its own VDM output driver. When execution begins, however, all output is sent to the current PTDOS CONOUT driver; a command is provided (the V command, described in Subsection 2.3) to direct output to the internal VDM driver, instead of to CONOUT.

1.2 LOADING AND INITIALIZATION OF THE DEBUGGER

To enter the debugger, type the name of the desired version of the program after the PTDOS prompt (*). If the program to be debugged is a PTDOS image file, type a command having the format

*file,DEBUG {parameters}

where "file" is the name of the program to be debugged, and {parameters} represents the list of parameters required by that

program. (The brackets are not literal; they indicate that a parameter list is optional, since not all programs require parameters.) Remember that typing a filename followed by a comma causes the named file to be loaded but not executed: a command of the form given above will execute only the DEBUG program. (The first file will be loaded into memory so that it can be executed from within the debugger.)

When DEBUG begins to run, it will display the question

RST?

on the output device (probably the video display). Your response, a number between Ø and 7, inclusive, will determine which 8080 "restart" location will be used by the debugger to implement breakpoints. (Subsection 1.3 discusses breakpoints and restarts.)

When the > prompt appears, the debugger is ready to accept a command from the keyboard.

1.3 BREAKPOINTS AND RESTARTS

A BREAKPOINT is a location at which the operation of a program stops to permit some kind of external intervention; in the case of DEBUG, the user program (or the program being examined) stops to permit the user to examine and modify registers and memory. A breakpoint can be set at any address; when the debugger is in operation and a breakpoint has been implemented, it is possible to proceed from that breakpoint, or even to determine that execution will no longer stop there.

When a breakpoint is encountered by the debugger, the value of each register is immediately displayed as a hexadecimal number following the letter symbol for the register and an equals sign (=). For example, B=3E means that the number 3E is in register B. The symbols for the registers are:

A for the Accumulator
B for register B
C for register C
D for register D
E for register E
F for CPU Flags
H for register H
L for register L
M for the content of the memory location to which H and L point
P for Program Counter
S for Stack Pointer

The flags that were set at the time of the breakpoint are indicated by letter symbols following the letter "F" and an equals sign. The symbols for the flags are:

S for the Sign flag
Z for the Zero flag
A for the Auxiliary Carry flag
P for the Parity flag
C for the Carry flag
N for no flag

Thus, F=ZAP means that the Zero, Auxiliary Carry, and Parity flags were set. (Obviously, the letter N will only appear if no other symbol follows the equals sign.)

The X command (discussed in Subsection 2.3, below) may be used to modify any of the values existing in the registers or flags at the time of the breakpoint. There are also commands that make it possible to examine and modify the contents of memory at any named location.

The 8080 microcomputer allows for eight possible RESTART locations, numbered 0 through 7 and corresponding to memory addresses 0, 8, 16, 24, 32, 40, 48, and 56 Decimal. It is quite common to give a much-used subroutine an origin at one of the restart locations, because a call to such a location requires only the one-byte RST instruction, rather than the three-byte CALL instruction. In the debugger a subroutine for dealing with breakpoints has its origin at whatever restart location is specified in answer to the RST? question. The program provides for a choice of restart locations, in order to allow the other restart addresses to remain available for access by the user. (For example, the program being debugged may use restart addresses as origins for some of its subroutines.) If all eight of the restart locations are available for use by the debugger, then the answer to RST? can be any number between 0 and 7.

SECTION 2

COMMANDS

2.1 CONVENTIONS

On the next page is a list of the commands accepted by the DEBUG program. In this list and for the remainder of the manual, the following conventions are used:

The symbol <cr> denotes the RETURN key.

Upper case letters are literal: the $Bexpr<cr> command actually contains the upper case letter B. (Note that it also ends with a carriage return.)

Lower case letters are not literal: the $Bexpr<cr> command contains a four character hexadecimal address or an expression that evaluates to a four character hexadecimal address. (The rules governing expressions are given below.) If a number occupying more than four hexadecimal places is entered as an address, only the rightmost four characters are significant. Similarly, if an expression evaluates to a number occupying more than four hexadecimal places, only the rightmost four places are significant.

Brackets {} indicate that a parameter is optional. The command $P{n}<cr> contains an optional parameter represented by the letter n.

Other punctuation is literal, except that the dollar sign ($) signifies the ESCape key, rather than the shift-4. (The ESCape key is actually echoed on the screen as a dollar sign.)

EXPRESSIONS

The letters expr denote an expression that points to an address in memory, i.e., an expression that evaluates to a number between Ø and 65535, inclusive. The characteristics of an expression are as follows:

1) An expression may involve any of the operators + (add), - (subtract), * (multiply), and % (divide). Expressions are evaluated from left to right, with no operator precedence. Parentheses are not allowed.

2) An operand is assumed to be a hexadecimal number, unless it is preceded by an exclamation point (!), in which case it is assumed to be a decimal number.

100 is 100 Hexadecimal, or 256 Decimal.
!100 is 100 Decimal, or 64 Hexadecimal.

3) Multiplication and division operate on two 16-bit unsigned numbers. The result of division is truncated to its integer part, and the remainder is lost.

3D%7 evaluates to 0008.

4) There is no check for overflow or for division by zero.

It may be useful to imagine the acceptable range of numerical values (corresponding exactly to the range of addressable memory) on a circular number "clock," with 0 at the twelve o'clock position and values increasing in a clockwise direction. Thus the largest number in the system (65535) is next to the smallest (0), just counter-clockwise of twelve o'clock. If we follow the rule, "Move clockwise to increment a value, counter-clockwise to decrement a value," it becomes clear that in this system 3-5 will be 65534, and 65534+5 will be 3. Although it is possible to utilize this arrangement to advantage, it is probably less confusing to use expressions that actually evaluate to a number neither less than zero, nor greater than 65535.

Division by zero will always give the result 65535 Decimal.

5) A period (.) in an expression represents the address of the last memory location examined. Thus, a memory location offset by 100 Hexadecimal from the last location examined could be represented as 100+. or as .+100. If no memory location has yet been examined, the value of . is 0000.

6) BLANKS ARE NOT ACCEPTABLE WITHIN EXPRESSIONS.

2.2 DEBUGGER COMMAND LIST

All of these commands will be described in the next subsection. A command may be entered at any time that the > prompt appears on the video display.

SYNTAX	FUNCTION
$A<cr>	Set breakpoint mode to Static.
$Bexpr<cr>	Set breakpoint at address expr. Up to fifteen breakpoints may be set.
$C<cr>	Set output mode to Character.
$Daddr<cr>	Delete the breakpoint at address expr.

$E<cr>	Exit DEBUG; return to PTDOS.
expr/	Display the content of the location designated by expr. Allow modification of the value at that location.
expr=	Print the value of expr in Hexadecimal.
expr#	Print the value of expr in Decimal.
$Fexpr1,expr2,bb<cr>	Fill memory from address expr1 to address expr2 with byte bb.
$H<cr>	Set output mode to Hexadecimal.
$I<cr>	Set output mode to Instruction.
$K<cr>	Delete all currently set breakpoints.
$Lfile<cr>	Send subsequent output to the named PTDOS file. The filename (denoted by "file") must meet the requirements of PTDOS. The L command without a file name stops output to an open log file, or causes such output to resume.
$P{n}<cr>	Proceed from a breakpoint; continue execution, skipping this breakpoint until it is met again for the nth time. Default for n is 1.
$Rexpr<cr>	Begin execution (of the program being debugged or examined) at address expr.
$Sexpr1,expr2,bb,mm<cr>	Search memory from address expr1 to address expr2 for byte bb using mask mm.
$T<cr>	Display a list of current breakpoint addresses.
$V<cr>	Change output driver (CONOUT to VDM, or vice-versa).
$Wexpr1{,expr2}<cr>	Dump contents of memory from address expr1 to address expr2.
$Xr<cr>	Display the content of CPU register r. Allow modification of that value.
$Z<cr>	Set breakpoint mode to Remove.

2.3 DESCRIPTION OF COMMANDS

This subsection describes all of the commands in the debugger and provides short examples of their use. (There are no examples in cases in which the operation of a command is not evident on the display, i.e., in which the DEBUG program simply issues a carriage return and a prompt after the command is executed.) For the purpose of this discussion, it is convenient to group the commands as follows:

GROUP 1 DEBUGGER CONTROL

These are commands not directly related to the process of debugging a program. They determine where output will be sent from the debugger (V,L), and whether the contents of memory will be represented as hexadecimal numbers (H), characters (C), or 8080 instructions (I). Also included in this group is the command that terminates execution of the debugger (E).

GROUP 2 CONTROLLING EXECUTION OF THE USER PROGRAM

These commands set and delete breakpoints (B,D,T,K), start and restart the program being debugged (R,P).

GROUP 3 EXAMINING AND MODIFYING MEMORY

These commands are related to the examination and modification of particular memory locations (expr/,W,F,S), CPU registers and flags (X). These commands are generally used after a breakpoint has been encountered, although it is possible to examine memory without setting any breakpoints.

Subsection 2.4, below, illustrates a typical sequence of steps followed while debugging a program.

GROUP 1: V and L set the output file(s).
H sets output mode to Hexadecimal.
I sets output mode to Instruction.
C sets output mode to Character.
E exits the program.

CHANGE OUTPUT DRIVER $V<cr>

DEBUG can send output either to the internal VDM driver or to the PTDOS CONOUT driver; the V command changes the output driver from CONOUT TO VDM, or vice-versa. When the program is first executed, output is sent to the CONOUT driver. (Normally, this is also the video display.)

The internal VDM driver has a variable speed option: while output is being displayed, it is possible to alter the speed of the display by striking a key representing one of the digits (Ø is fastest, 9 is slowest). Output can be suspended temporarily by the space bar and reactivated by any other key. The default display speed is 2.

SET LOG FILE $Lfile<cr>

The L command opens a named PTDOS file to receive output from the debugger. (All output will also go to the CONOUT driver or the internal VDM driver.) When DEBUG begins execution, output is set to go only to CONOUT; the first time that the L command is entered, the named file is opened.

If the L command is entered without a filename, the debugger will either start or stop sending output to an open log file, whichever action would change the condition that existed when the command was given. Once open, a log file is closed only by an exit from the debugger.

EXAMPLE:

```
>$LNOTES/1<cr>   (command to open a file NOTES on unit 1)
>                (NOTES is opened)
....
....             (other commands entered)
....
>$L<cr>          (output will no longer be sent to NOTES)
>                (NOTES is closed; NOTES2 is opened)  ....
....             (other commands entered)
....
>$L<cr>          (output will be sent to NOTES again)
```

SET OUTPUT MODE TO HEXADECIMAL $H<cr>

This command determines that when the content of a memory location is examined (expr/ command), it will be displayed as a hexadecimal number. (The commands to examine memory are in Group 3.) The default mode for output is Hexadecimal; it is therefore unnecessary to specify this mode unless another mode is in force.

SET OUTPUT MODE TO INSTRUCTION FORMAT $I<cr>

This command determines that when the content of a memory location is examined (expr/ command) or dumped (W command), it will be decoded into the corresponding 8080 instruction mnemonic. (The twelve undefined operation codes are output in Hexadecimal.) In Instruction mode, DEBUG will assume that the location given by the expression in the expr/ command is the first byte of an instruction. If the location specified in the command is, in fact, the second or third byte of a multiple-byte instruction, DEBUG will still decode the byte as an assembly language instruction mnemonic, and the result will not reflect what is actually happening in the object code.

There are two exceptions to the rule that every byte displayed in Instruction mode will be displayed as an 8080 instruction mnemonic. If the W command is entered while Instruction mode is set, or if expr/ specifies the first byte of a multiple-byte instruction and the linefeed key is used to examine the next location(s), the DEBUG program will display the second and third bytes of instructions in Hexadecimal format.

SET OUTPUT MODE TO CHARACTER $C<cr>

This command determines that when the contents of memory are examined (expr/ command), any value that corresponds to the code for a printable ASCII character will be displayed as that ASCII character. Any value that does not correspond to a printable ASCII character will be printed as a hexadecimal number.

EXIT TO PTDOS $E<cr>

This command terminates execution of the debugger and returns to PTDOS. At this point the altered program may be saved from memory, or its source may be altered by one of the PTDOS editors. If the program is going to be saved from memory, all current breakpoints must be removed before the E command is entered.

GROUP 2: B sets a breakpoint; D deletes a breakpoint.
T displays all breakpoints; K deletes all breakpoints.
A and Z set breakpoint mode.
R and P begin and restart program execution.

SET BREAKPOINT $Bexpr<cr>

This command sets a breakpoint at the location specified by the expression expr. A breakpoint causes program execution to stop immediately BEFORE the execution of the instruction at the specified address; for this reason it is not permissible to set a breakpoint on the second or third byte of a multiple-byte instruction.

There may be as many as fifteen breakpoints set at any given time.

DELETE BREAKPOINT $Dexpr<cr>

This command deletes the breakpoint currently set at the location specified by the expression expr. If there is no breakpoint at the specified address, a question mark will be printed.

DISPLAY ALL CURRENT BREAKPOINTS $T<cr>

This command causes the addresses of all current breakpoints to be displayed; thus it becomes evident how many breakpoints have been set and whether there are any that can be deleted.

EXAMPLE:

```
>$T<cr>          (command to type out current breakpoints)
4Ø75             (addresses at which breakpoints have been
4Ø89             set with the B command)
41Ø2
```

KILL ALL CURRENT BREAKPOINTS $K<cr>

This command deletes all of the breakpoints that have been set. Once a program has been debugged, it can be executed normally from within the debugger if all breakpoints have been removed. If the altered version of a program is going to be saved following a return to PTDOS, it is necessary to delete all breakpoints before entering the E command.

SET BREAKPOINT MODE TO STATIC $A<cr>

This command determines that breakpoints will NOT be deleted after they are encountered, that is, that execution will stop again every time a breakpoint address is reached. Static mode is set when the debugger is entered.

SET BREAKPOINT MODE TO REMOVE $Z<cr>

This command determines that breakpoints WILL be deleted after they are encountered. Execution will stop only the FIRST time that the breakpoint address is reached.

PROCEED FROM A BREAKPOINT $P{n}<cr>

This command causes program execution to resume after a breakpoint has been encountered and related examination or modification of the code has been completed. Execution will continue, beginning at the instruction that caused the break, and will proceed until the next breakpoint is encountered. All registers will be loaded with values that reflect the modifications that have been made; a register or flag whose value has not been modified will retain the value that it contained when the breakpoint was encountered.

If a number is given after the letter P, the command is taken to mean: proceed with execution and do not stop again for this breakpoint until it is encountered for the nth time. For example, the command $P5<cr> will cause the breakpoint just implemented to be bypassed four times; all other breakpoints will be implemented normally. The default for n is 1; that is, normally execution will proceed, and any breakpoint that has not been deleted or removed will be implemented normally.

BEGIN EXECUTION $Rexpr<cr>

This command will start execution of a program at the location specified by expression expr. The R command is used to execute a program at its starting address; it should not be used to proceed from a breakpoint, because the values of registers and flags will not be restored! (The P command, by contrast, restores the values of registers and flags.)

GROUP 3: X displays CPU registers and flags.
W dumps a series of memory locations.
F fills a series of locations with a given value.
S searches a series of locations for a given value.
expr/ displays the contents of location expr.
expr= displays the value of expr in Hexadecimal.
expr# displays the value of expr in Decimal.

DISPLAY CPU REGISTERS AND FLAGS $Xr<cr>

This command is used to examine and modify the values of CPU registers and flags after a breakpoint has been encountered. The r in the command format represents a symbol for the register that is to be examined or modified. If no value is specified for r, the values of ALL registers and flags are displayed. The carriage return is NOT required if a value is specified for r.

SYMBOLS DESIGNATING REGISTERS AND FLAGS

Here is a list of the symbols for registers and memory. Any of these symbols may be used as the r element in the X command.

A for the Accumulator
B for register B
C for register C
D for register D
E for register E
F for CPU Flags
H for register H
L for register L
M for the content of the memory location
to which H and L point
P for Program Counter
S for Stack Pointer

These are the symbols for the flags. The X command will not display the value of an individual flag; rather, the flags are displayed as a group when F is specified in the X command.

S for the Sign flag
Z for the Zero flag
A for the Auxiliary Carry flag
P for the Parity flag
C for the Carry flag
N for no flag

MODIFYING A REGISTER OR FLAG

In order to modify a register or flag, enter the X command, following the letter "X" with the symbol that designates the register. To modify one of the flags, type $XF<cr>, NOT the symbol that stands for the particular flag! If the letter "X" is followed simply by a carriage return, the values of all registers and flags will be displayed again.

When the X command is entered, the value of the named register will be displayed.

EXAMPLE:

```
>$XB            (command to display register B)
B=52            (hexadecimal value of register B)
```

If you do not want to modify the register, after all, type a carriage return to re-enter command mode. If you DO want to modify the register, enter a new value at the cursor position, without inserting additional punctuation or spaces. The value that you enter will replace the value currently in the register. Follow the entry with a space or a carriage return; a space dictates that the next register be displayed (on the current line), while a carriage return effects a return to command mode. To modify the value of a flag, enter the symbols of all flags that are to be set, whether or not they are set already.

```
>$XF<cr>          (examine flags)
F=SZC SZP<cr>     (Sign, Zero, and Carry flags already set;
                  user sets Parity, alters carry so no
                  longer set)
>                 (back in command mode)
```

DUMP MEMORY $Wexpr1{,expr2}<cr>

This command causes the contents of a specified section of memory to be displayed in the current output mode (see Group 1). If the mode is not Instruction format, memory will be dumped with fourteen bytes represented on each line: first all bytes are displayed in Hexadecimal, and then all are displayed as characters. (A period will be printed if the value of the byte does not correspond to a printable ASCII character.) In Instruction format, memory will be dumped in decoded format, with one instruction per line and the second and third bytes of multiple-byte instructions represented in Hexadecimal.

Memory will be dumped starting at the location specified by expr1 and continuing to that specified by expr2. If no value is specified for expr2, the value expr1 will be used; the dump will continue as though memory were circular, starting at expr1 and continuing past 65535 to 0, finally stopping when the byte before expr1 is reached. To terminate a dump before it reaches expr2, type MODE SELECT or CTRL-@.

```
>$W100,110<cr>

0100 01 07 00 21 50 00 11 65 00 78 B1 C8 0B 7E  ...!P..e.x...~
010E 12 24 14                                   .$.
```

FILL MEMORY WITH A GIVEN BYTE $Fexpr1,expr2,bb<cr>

This command fills memory from expr1 to expr2 with byte bb. If expr1 and expr2 are quite far apart in memory, a few moments may pass before the prompt (>) reappears on the screen.

SEARCH MEMORY FOR A GIVEN BYTE $Sexpr1,expr2,bb,{mm}<cr>

This command searches memory from the location specified by expr1 to that specified by expr2 for byte bb, using mm as a mask.

As each byte is examined, it is ANDed with mask mm and then checked for equality to byte bb. If the quantities are equal, then the memory address and the byte at the address are printed. If the mask is not specified, it will be assumed to have the value ØFF Hexadecimal, i.e., all bits will be compared.

EXAMPLE:

```
>$S234Ø,2375,4C<cr>    (search for 4CH, using default mask)
2357 4C                (4C is found at location 2357)
>$S5261,5269,Ø,1<cr>   (search for even numbers)
5261 C2                (even numbers found at five locations)
5263 52
5264 3E
5265 ØE
5266 9Ø
```

EVALUATE EXPRESSION OR EXAMINE MEMORY expr=, expr#, expr/

If an expression is entered and followed immediately by an equals sign (=), the expression is evaluated and the result is displayed as a Hexadecimal number.

```
>5*6=ØØ1E        (result displayed in Hexadecimal)
```

If an expression is entered and followed immediately by a pound sign (#), the expression is evaluated and the result is displayed as a Decimal number.

```
>5*6#ØØØ3Ø       (result displayed in Decimal)
```

If an expression is entered and followed by a slash (/), the expression is evaluated and the content of the memory location denoted by the expression is displayed in the current output mode (see Group 1).

```
>$H<cr>          (output mode set at Hexadecimal)
>!34%2/ 4Ø       (value of location 17 Decimal is
                  displayed in Hexadecimal output mode)
```

Note that none of these commands requires a carriage return.

In the rules for expressions (see Subsection 2.1), an expression was defined as POINTING to an address in memory. Actually, in the case of an expression followed by an equals sign or a pound sign, the expression need not denote a location that is to be examined; any arithmetic problem whose result will lie between Ø and 65535, inclusive, can be entered, even if the computer being used does not have any memory at the designated location. If the expr/ command is entered and there is no memory at the specified location, the result will be FF in Hexadecimal mode, RST Ø7 in Instruction mode.

Once expr/ has been entered and the location has been displayed, several different entries are possible.

A CARRIAGE RETURN effects a return to command mode.

A SINGLE QUOTE MARK (') causes the value of the location to be displayed in Character mode (without changing the current mode setting for the debugger).

```
>$H<cr>                (set output mode to Hexadecimal)
>5002/ 50 ' P<cr>      (value displayed in Hex, then as
>                      Character; return to command mode)
```

A SEMI-COLON (;) causes the byte to be displayed in Instruction format; the location is assumed to be the first byte of an 8080 instruction. The current mode setting for the debugger is not altered.

```
>$H<cr>                        (set output mode to Hexadecimal)
>1234/ 39 ; DAD SP<cr>         (value displayed in Hex, then as
>                              Instruction; return to command mode)
```

An EQUALS SIGN (=) causes the contents of the present memory location to be displayed in Hexadecimal, without changing the current mode setting for the debugger.

```
>$I<cr>                  (set output mode to Instruction)
>025F/XCHG =EB<cr>       (value displayed as Instruction, then
>                        in Hex; return to command mode)
```

A DOUBLE QUOTE MARK (") followed by a character specifies that character as a replacement for the current value of the location. Replacement input must be terminated by a carriage return, linefeed, or up arrow; each of these delimiters will also perform the function ascribed to it elsewhere in this list, e.g., a linefeed will delimit replacement input and then cause the next location to be displayed. If an attempt is made to enter more than one character following a double quote mark, DEBUG will respond with a question mark and will not accept either character entered.

```
>$C<cr>              (set output mode to Character)
>47D8/ @ " B<cr>     (value displayed as Character, replaced
>./ B<cr>           with letter B; new value displayed)
>                    (return to command mode)
```

(In this example, the period is used to designate the last location displayed; see the rules for expressions in subsection 2.1, above.)

A COLON indicates that subsequent input is an instruction. It is possible to enter a multiple byte instruction in place of a single byte instruction; input will be placed in successive memory locations and the previous contents of those locations will be overwritten. Such a disturbance of the previous contents of memory will seldom be desirable.

To enter a replacement in Instruction format, type the mnemonic for the instruction, rather than the corresponding machine code. The standard Intel instruction mnemonics have been implemented. (See the appendix of 8080 Operation Codes.) Most of the standard symbolic names for registers may be used in the operand field; the two exceptions are that "P" should be used to denote the Program Status Word (PSW) and "S" should be used to denote the Stack Pointer (SP). The instruction must be entered immediately after the colon. Use a single blank to separate operands from the operation code, and a comma to separate two operands. Terminate the input with a carriage return, linefeed, or up arrow; any of these delimiters will first delimit the input and then perform the function ascribed to it elsewhere in this list, e.g., a carriage return will delimit the input and than cause a return to command mode.

```
>$H<cr>                          (set output mode to Hexadecimal)
>2113/ 1C ;INR E : INR C<cr>     (value displayed in Hex, then as
>                                Instruction; Instruction input
                                 and return to command mode)
```

A LINEFEED causes the contents of the next location to be displayed. If the current output mode is Instruction mode and the last location examined was interpreted as the first byte of a multiple-byte instruction, the next one or two locations, if examined by means of the linefeed, will be displayed in Hexadecimal.

```
>$I<cr>                          (set output mode to Instruction)
>5002/MOV D,B <linefeed>         (value displayed; display next
5003 LXI SP, <linefeed>          location, and next...)
5004 3E <linefeed>               (second and third bytes of multi-byte
5005 F5 <cr>                     instruction are displayed in Hex)
>                                (return to command mode)
```

An UP ARROW (^) causes the content of the previous location to be examined. If the current mode is Instruction mode, the location will be assumed to be the first byte of an instruction (whether or not this is actually the case) and will be decoded into an 8080 instruction mnemonic. (Consider that whereas it is possible to determine from an operation code how many subsequent bytes are part of the instruction, it is not always possible to tell whether or not PRECEDING bytes are operation codes.)

```
>$C<cr>          (set output mode to Character)
>2300/ G ^       (value displayed as Character)
2299 U<cr>       (value of previous location displayed)
>                (return to command mode)
```

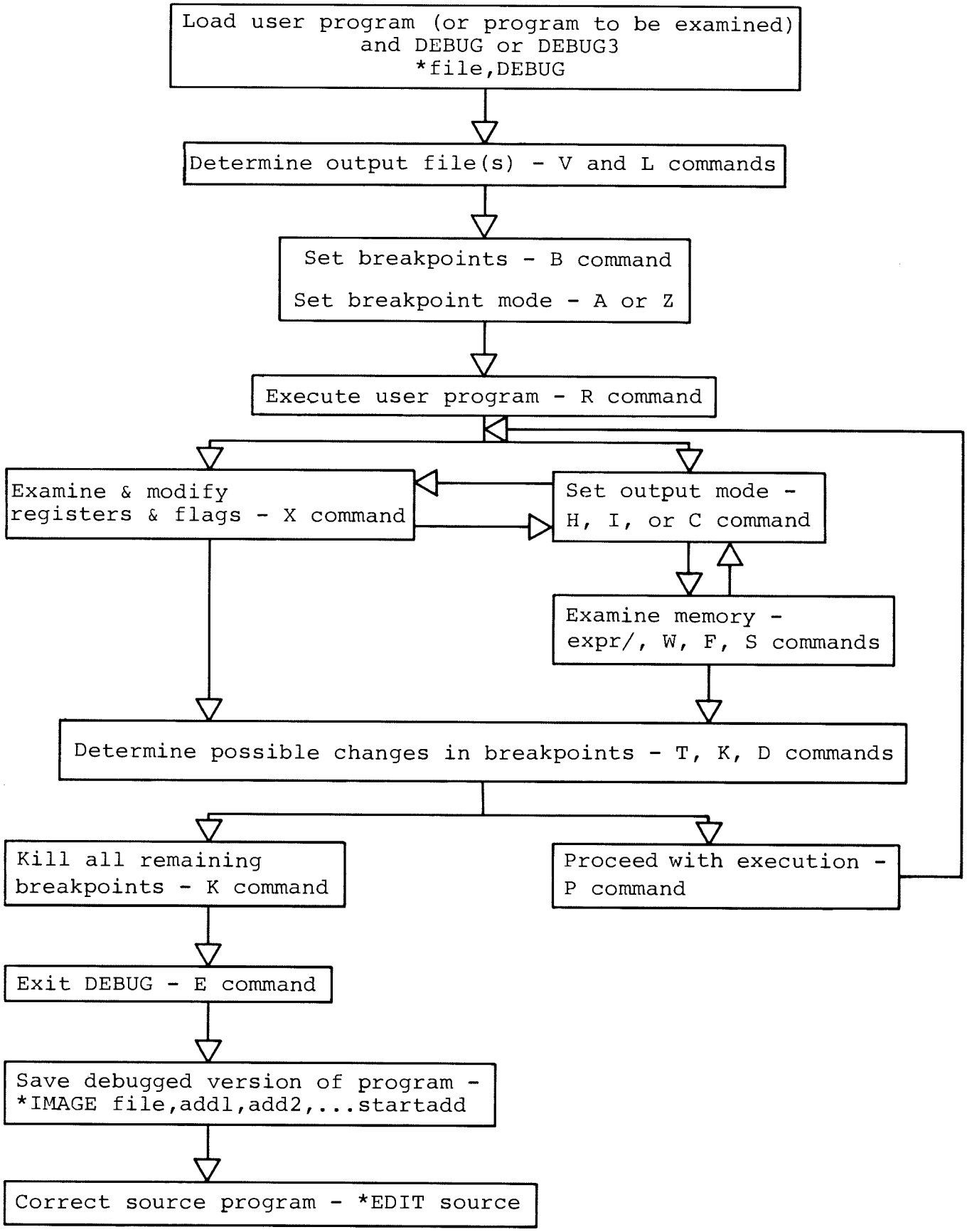

Fig. 2-1 Typical Procedure for Using DEBUG to Debug A Program

2.4 A WALK THROUGH THE DEBUGGER

The example below illustrates the use of DEBUG to locate and correct the errors in an assembly language program. The next few pages consist entirely of tutorial material; they do not contain any additional information about the features of the debugger. The figure on the facing page is a generalized diagram of the process of using DEBUG to debug a program. This figure and the command summary in subsection 2.2 are intended to serve as your quick reference materials after you have read the detailed descriptions in the rest of the manual.

SAMPLE PROBLEM

Below is the assembler listing of a routine just added to a large program called TEST. The purpose of this routine is to move BC bytes of information from one area of memory to another. When the routine is first called, the H and L registers point to the first of BC consecutive locations occupied by the information to be moved; the D and E registers point to the first of BC consecutive locations to be occupied by the same information when control returns to the calling routine. The calling routine prints out the BC bytes beginning at the location to which the first byte was moved.

```
0100 78       BMOVE    MOV    A,B
0101 B1                ORA    C
0102 C8                RZ     .
0103 0B                DCX    B
0104 7E                MOV    A,M
0105 12                STAX   D
0106 24                INR    H
0107 14                INR    D
0108 C3 00 01          JMP    BMOVE
```

Let us assume that we have run TEST in PTDOS, and that where we would expect 7 bytes, the characters F, I, D, D, L, E, and S, to be printed out as the ASCII string FIDDLES, we see a good first character 'F' followed by a great deal of suspicious screen activity. (Write and execute a program that calls BMOVE and then prints out about 100 bytes, beginning at the location indexed by D and E at the time of the call.) Let us also assume that we do not immediately recognize the bugs in the program, and that we decide to use the debugger to take a closer look at the BMOVE routine during its execution.

Remember that the symbol > is the prompt character and should not be typed. Also, $ signifies the ESCape key, not the dollar sign.

1) LOAD DEBUG or DEBUG3 from PTDOS

```
*TEST,DEBUG<cr>
```

TEST is loaded but not executed. DEBUG is loaded and executed.

```
RST? 3
```

DEBUG asks user to assign restart location. User enters 3.

2) DETERMINE OUTPUT FILE(S) with L and/or V commands

```
>$LSAVE<cr>
```

Output will be sent to the PTDOS file called SAVE, as well as to the default output file, i.e., the CONOUT driver. (The V command could be used to direct output to the internal VDM driver, rather than to CONOUT.)

3) SET BREAKPOINT(S) with B command

```
>$BØ1Ø2<cr>
```

Because only the first byte of information appears to have been moved properly (see description above), it is worth checking whether the BMOVE loop is executed only once, that is, whether the zero flag is set and causes a return the second time the RZ instruction is reached. By setting a breakpoint at Ø1Ø2H, we can examine the condition of the flags at the time of the RZ instruction. (More breakpoints could, of course, be set; to simplify this example, we set only one at a time.)

4) SET BREAKPOINT MODE with A or Z command

```
>$A<cr>
```

Breakpoints will not be deleted automatically after they have been encountered once. The distinction between the modes is relevant here, because the breakpoint at Ø1Ø2H will give useful information only the second time it is encountered. (It is actually unnecessary to use the A command, unless the Z command has been used previously; Static breakpoint mode is set when the DEBUG program begins to run.)

5) EXECUTE USER PROGRAM (or program to be examined) with R command

```
>$R12ØØ<cr>
```

The address specified in this command is the starting address of the program called TEST. Execution will proceed until it reaches the breakpoint address; then that address and the contents of all registers and flags will be displayed.

```
*Ø1Ø2
A=Ø7 B=ØØ C=Ø7 D=ØØ E=65 F=N H=ØØ L=5Ø M=46 S=625C P=Ø1Ø2
```

Assuming that we intended to move 7 bytes of information beginning at address 5Ø to consecutive addresses beginning at 65, all is well so far. The STAX operation will put the value 46 (or ASCII 'F') in location ØØ65H. (M represents the value of the location addressed by H and L.)

SET OUTPUT MODE with H, I, or C commands
EXAMINE MEMORY with expr, W, F, or S command

For good measure, we can EXAMINE MEMORY to make sure that the characters F, I, D, D, L, E, and S are actually stored at consecutive

locations beginning at 50. The W command can be used to display the locations; output mode does not need to be set to Character, because in the default Hexadecimal mode the dump will appear both in Hexadecimal and in Characters.

```
>$W50,56<cr>

0050 46 49 50 44 4C 45 53                          FIPDLES
```

The dump shows that the third character is incorrect. To insert the correct character at address 52, we can enter

```
>$C<cr>
```

to set the output mode to Character, and then

```
>52/
```

to display the contents of location 52. The contents of the location will be displayed right after the slash. To enter the correct character, we type a double quote ("), the character, and a carriage return. Now the line looks like this:

```
>52/ P " D<cr>
```

Of course, the fact that there was a P instead of a D at location 52 does not account for the fact that the program does not run properly.

6) DETERMINE POSSIBLE CHANGES IN BREAKPOINTS with T,K,D,B

In this instance, we have no real reason to type out or delete our one breakpoint, but we might want to add a breakpoint at address 0108H. By looking at the registers at that point, we can see whether the locations addressed by HL and DE are what we would expect them to be, i.e., whether the value of each of these register pairs has been incremented by 1.

```
>$B0108<cr>
```

7) PROCEED FROM BREAKPOINT with P command

```
>$P<cr>
```

Execution will continue until the next breakpoint is encountered.

```
*0108
A=46 B=00 C=06 D=01 E=65 F=N H=01 L=50 M=2E S=625C P=0108
```

From this display of the values of registers and flags, it becomes clear that the register pairs that address memory locations have actually been incremented not by 1, but by 100H (256 Decimal). In order to verify that data is actually being stored at every hundredth (or 256th) address, we can proceed with execution until the next time

Ø108H is reached. Then we can use other commands to examine memory locations in the areas from which and to which we want to move our data. To proceed with execution, we enter the command

```
>$P<cr>
```

(We will not delete the breakpoint at Ø102, because we might want to look at it again; the next time that breakpoint is encountered, however, we can ignore it and Proceed with execution.)

When we reach Ø108, the values of registers and flags are:

```
*Ø108
A=2E B=ØØ C=Ø5 D=Ø2 E=65 F=N H=Ø2 L=5Ø M=Ø6 S=625C P=Ø1Ø8
```

8) EXAMINE MEMORY with the expr/ command

If we enter the command to examine locations 15Ø and 165, we can indeed see that the second byte to be moved was taken from location 15Ø and moved to 165, instead of being taken from location 51 and moved to 66.

```
>15Ø/ @<cr>
>165/ @<cr>
```

If we look at location 66, we find whatever value was at that location when the debugger began its operation:

```
>66/ .<cr>
```

If our sample routine were not so short, we might want to use the MEMORY SEARCH (S) command to locate the part of the program containing the INX instructions. The table of 8Ø8Ø Operation Mnemonics in the ASSM subsystem manual indicates that INX H, which should be one of our instructions, corresponds to the Hexadecimal value of 23. To search a section of memory above our most recent breakpoint, we can enter

```
>$S1ØØ,1Ø8,23<cr>
```

only to find that the BMOVE routine does not contain an INX H instruction at all!

Using the expr command to EXAMINE MEMORY, we can look at the code in the same area that we just searched for INX. First we shall change the output mode to Instruction, so that we will see the contents of memory as a series of assembly language instructions, rather than as Hexadecimal numbers.

```
>$I<cr>
>Ø1ØØ/ MOV A,B<linefeed>  (you enter the linefeeds)
Ø1Ø1/  ORA C<linefeed>
Ø1Ø2/  RZ .<linefeed>
       etc.
Ø1Ø6/  INR H<linefeed>
Ø1Ø7/  INR D<cr>
```

Now the cause of our troubles is clear: instead of adding 1 to each of the register pairs HL and DE, we have added 1 to each of the single registers H and L. By examining locations 0106 and 0107 again, we can change the two INR instructions to INX instructions and solve our problem. The colon indicates instruction input.

```
>0106/ INR H :INX H<linefeed>
0107   INR D :INX D<cr>
```

9) EXAMINE AND MODIFY REGISTERS AND FLAGS with X command

We have found the bug in BMOVE and want to continue running TEST, rather than reinitiate execution of that calling program. We can use the X command to modify several registers, and so backtrack in our execution of the program to a point before BMOVE first put an incorrect byte in an incorrect location.

Remember,we are still at a breakpoint. The X command without a register specification will cause the contents of all registers and flags to be displayed:

```
          >$X<cr>
A=2E B=00 C=05 D=02 E=65 F=P H=02 L=50 M=06 S=625C P=0108
```

The bug in BMOVE caused all bytes but the first to be moved to incorrect locations in memory. To backtrack to the point from which we want to reinitiate execution, we must alter the following registers:

C, so that BC indicates that the last 6 of the 7 bytes that compose the (English) word FIDDLE must still be moved:

```
>$XC
C=05 06
```

D and E, so that they point to the next location to which information should be moved:

```
>$XD
D=02 00
>$XE
E=65 66
```

and

H and L, so that they point to the next location from which information should be taken.

```
>$XH
H=02 00
>$XL
L=50 51
```

10) KILL ALL BREAKPOINTS using K command

Before either proceeding with execution or exiting to PTDOS to save the file, we kill all current breakpoints. In order to have used almost all possible commands in this example, we may as well type out the breakpoints first with the T command.

```
>$T<cr>
0102
0108
>$K<cr>
```

11) EXIT DEBUG with the E command

```
>$E<cr>
```

This is the command to return to PTDOS. Once in PTDOS, we can use the PTDOS IMAGE command to save the object file, and either EDIT or EDT3 to alter the source code file.

EDIT*

A TEXT EDITOR FOR PTDOS 1.5

TABLE OF CONTENTS

EDIT

*This manual describes EDIT, Release 1.1.

SECTION 1

INTRODUCTION

The EDIT command initiates execution of a text editor program intended for use with the Sol video display or the VDM-1. When the EDIT command is given, the named file is read into memory and the first sixteen lines are displayed on the screen. The user may then either edit the file using single-letter control character commands, or, in the case of a new file, simply begin entering text as though on a typewriter, with all typed material appearing on the screen and all editor commands available for making corrections.

The single-letter control character commands make it possible to move the cursor anywhere on the screen, or to move through the file so that any desired 16 lines of text are visible. Characters or entire lines may be inserted, deleted, or moved from one part of the file to another. Text may be scrolled up and down, i.e., it is possible to move either backwards or forwards in a file and to stop this process when the desired part of the file is visible. There are also provisions for searching a file for particular strings of characters, replacing every occurrence of a given string with another string, and inserting another whole file into the text being edited.

At the conclusion of editing, a file may be written back to itself as an update, or it may be written to another named file, if several slightly different versions of one file are needed, or if a back-up of the same file is desired. Editor commands may also be used to divide a file into two or more files, if the original file becomes too long. The entire edited file is in memory at once, so it is even possible to choose NOT to save the file after editing. (This feature might be used if if you made extensive corrections and suddenly realized they were not appropriate.)

This editor accepts lines containing no more than 64 characters. There is no minimum line length. (Unlike the earlier version of EDIT, this version does not add blanks to lines containing fewer than 4 characters.)

SECTION 2

OPERATION OF EDIT

This section provides a general overview of the operation of EDIT.

2.1 COMMAND FORMAT

EDIT <from file> {,<to file>}{,<top of memory>}

where:

file = <filename {/unit}{<A>}>

{ } = optional parameters

/unit = disk drive unit number. If this parameter is not included, the default unit number is used.

<A> = ALS-8 format file. In an ALS-8 format file, the first byte of each line is a count of the number of bytes in that line, including itself and the carriage return. If this parameter is not included in the file name, the file is assumed NOT to be in ALS-8 format.

Note: the characters '<' and '>' are part of the <A> file format specification:

TESTFILE<A>	is an ALS-8 format file on the default unit.
FILE.ONE/1<A>	is an ALS-8 format file on unit 1.
NAME/1A	is an illegal file name.

<from file> = the file that contains the text to be edited. This file will be loaded into memory when the EDIT command is entered.

<to file> = the file to which the edited file will be written at the conclusion of editing. If no <to file> is specified, the file is written back to the <from file>.

<top of memory> = the highest memory address to be used by the edit operation. This address defines the end of the text area. (The text area starts right after the EDIT program code.) A file that exceeds the size of the text area will not be loaded, and any operation on a file in memory will not be executed if it would make the file too large for the available area. The default for this parameter is either the last good memory location or the lowest PTDOS system address (GLLOW).

2.2 STARTUP

The <from file> is read from the disk into memory for editing. If the named file does not exist, the editor asks whether a new file by that name should be created:

file is non-existent. Create?

If the character 'Y' is typed, the file will be created with type = '.' (period) and the block size = 4CØH, and the new (empty) file will be loaded into memory. (This procedure is one of the ways of creating a new file in PTDOS.) If a character other than 'Y' is typed, control will return to PTDOS.

If the <to file> is open or write protected, the <from file> cannot be edited. If either of these conditions applies to the named (or default) <to file>, the editor will indicate that

<file name> is open

or

<file name> is protected

Control will then be returned to PTDOS.

When a file is loaded successfully, the following information is presented:

```
Last load addr:  1EBA     ... last memory address of the file
Load count:  ØEF8         ... byte count of the file
End of file at:  1EBA     ... end of file mark (Ø1) address
C/R to continue
```

As soon as the carriage return (C/R) is entered, the first sixteen lines of the file are displayed on the screen, with the cursor at line 1 and character position 1 (column Ø). If the file contains fewer than sixteen lines, the remainder of the screen will be filled with # signs. The file is now ready for editing.

2.3 MEMORY USAGE

The EDIT program is read into memory starting at location 1ØØH. The <from file> is then read into a file area beginning just after the EDIT program code, and ending at the top-of-memory specified in the EDIT command. If a top-of-memory address was not specified, the

editor tests the memory of the computer and assigns a top-of-memory address; this address can never exceed GLLOW, the lowest address occupied by PTDOS. (The value of GLLOW can be altered with the SET BU= command described in the Commands chapter of the PTDOS manual.)

The entire <from file> must fit into available contiguous memory. If an edit operation would increase that size of a file so that the text would no longer fit in the text area, a message appears on the cursor line:

FULL - TYPE CONTROL Q

The offending operation will not be performed. After typing the CTRL and Q keys simultaneously, the user should either shorten the file by removing text, or leave the editor by typing the CTRL and F keys. If the second alternative is chosen, the file will be written back onto the disk; the editor can then be used to create a continuation file, or to split the large file into two smaller files, so that further additions can be made to each. (The discussion of EN and ST, below, contains a step-by-step procedure for splitting a file.)

2.4 TERMINATION

There are two ways to terminate the editing of a file:

1) The ESCAPE key aborts the EDIT program.

If the user does not wish to save the file in memory by writing it to the <to file>, the ESCAPE key should be used to abort the EDIT program. When this command is given, the question

Abort?

appears on the screen. If the character 'Y' is typed in answer, control is passed back to PTDOS and no file update occurs. The <from file> still resides on the disk, just as it did before the edit. If any character other than 'Y' is typed, the editor will re-display the file with the cursor and file position just as it was before the abort command was given.

2) CTRL and F keys terminate EDIT and update the file.

When the CTRL and F keys are typed simultaneously, the following information appears on the screen:

```
ØFC3  File start address     ... first memory address of file
2B3B  File end address       ... last memory address of file
1B79  File count             ... byte count of file

OK to write to "<file name>"?
```

Type 'Y' to write to the <to file> and return to PTDOS. Type any other character to return to the editor. If any character other than 'Y' is typed, the editor will re-display the file with the cursor and file position just as it was before the exit command was given.

If a <to file> was specified in the EDIT command, the contents of the text area is written to that file at the conclusion of editing. If the named <to file> does not exist on the disk, it is created with a type of '.' and a block size of 4CØH. If the <to file> parameter was omitted from the EDIT command, the <from file> is updated with the edited material. (Remember that when you write to a file, you write over the previous contents of that file.)

The editor may be used to convert an ALS-8 file to non-ALS-8 format, or vice-versa:

```
EDIT name<A>,name
```

loads an ALS-8 format file; when that file is written back onto the disk, it will no longer be in ALS-8 format.

SECTION 3

EDITOR COMMANDS

In the introduction to this manual, it was explained that the editor can be used for two purposes. By entering the EDIT command and the name of a file that does not yet exist, one can create that file and use the editor to enter text (e.g., a program, a letter) for the first time. In this context the control character commands permit corrections to be made conveniently; it is possible to retype words, restructure paragraphs or other segments of text, without having to retype the surrounding material. The other use of the editor is to make alterations in existing text files (probably created in the same editor on an earlier occasion).

Below is a list of the control keys used by the editor. A more complete description of each command is given after the list. To enter a control character, type the CTRL key and the named letter key at the same time.

CONTROL KEYS:

CTRL/ W - move cursor up one line
CTRL/ Z - move cursor down one line
CTRL/ A - move cursor left one character
CTRL/ S - move cursor right one character
CTRL/ I - move cursor to next tab stop
CTRL/ E - scroll file up one line
CTRL/ X - scroll file down one line
CTRL/ R - scroll file up 16 lines
CTRL/ C - scroll file down 16 lines
CTRL/ Q - home cursor to line 7; also escape from FULL
CTRL/ T - toggle insert character mode; ON/OFF
CTRL/ Y - repeat next command n times
CTRL/ H - delete character under cursor
CTRL/ B - insert line above cursor
CTRL/ P - delete line
CTRL/ O - initiate string search mode
CTRL/ L - continue search for string
CTRL/ V - set TAB/BLOCK MOVE/NULL/LL/FL/ST/EN/PA/IF
CTRL/ U - execute TAB / BLOCK MOVE
CTRL/ F - exit editor with file update
CTRL/ M - same as RETURN (below)
CTRL/ J - same as LINEFEED (below)

OTHER KEYS:

TAB	-	same as CTRL/ I
LOAD	-	same as CTRL/ L
ESCAPE	-	abort editor with no file update
RETURN	-	insert line below cursor
LINE FEED	-	delete all text to the right from the cursor & position cursor one line down

Sol KEYS:

CURSOR UP	-	move cursor up one line (same as CTRL/ W)
CURSOR DOWN	-	move cursor down one line CTRL/ Z)
CURSOR LEFT	-	move cursor left one character (same as CTRL/ A)
CURSOR RIGHT	-	move cursor right one character (same as CTRL S)

NOTE: The cursor keys on a Sol are on either side of the space bar.

3.1 CURSOR AND FILE POSITIONING

The cursor marks the position at which the next typed character will appear on the screen (and so, also, in the file in memory). In general, any character entered from the keyboard, except a control character, will replace the character at the cursor position. (There is a special command that makes it possible to insert text at the cursor position without "typing over" any other characters. That command, the CTRL/ T command, will be described later.)

The cursor positioning commands, themselves, do not affect the contents of the file in memory; their purpose is to move the cursor to the desired position on the screen, so that insertions or changes may be made there.

CTRL/ W	move cursor up one line
CTRL/ Z	move cursor down one line
CTRL/ A	move cursor left one character
CTRL/ S	move cursor right one character

Note that the keys A, S, W, Z form a diamond on the keyboard.

The Sol keyboard also contains cursor control keys to the left and right of the space bar. These keys may be used with or without the control key to move the cursor up, down, left or right, as above.

CTRL/ Q cursor home command - file FULL excape

This command has two unrelated functions. Because these functions are useful in two completely different contexts, there is never any ambiguity as to the user's intentions.

1) Unless the file area is full and a warning message has been displayed, the command causes the cursor to be moved to line 7 and character position 1.

DO NOT USE THIS COMMAND TO MOVE THE CURSOR UNLESS THERE ARE AT LEAST SEVEN LINES OF TEXT ON THE SCREEN!!

2) The file area full condition is described above, in the discussion of how the editor uses memory. CTRL/ Q is the escape from this condition.

CTRL/ E	scroll up one line
CTRL/ X	scroll down one line
CTRL/ R	scroll up sixteen lines
CTRL/ C	scroll down sixteen lines

Screen scroll commands are provided to allow the file to be "rolled" through the screen area until the desired file line is reached. Scrolling UP moves the file upward across the screen, so that a LATER part of the file becomes visible; scrolling DOWN moves the file downward across the screen, so that an EARLIER part of the file becomes visible.

CTRL/ I or TAB move cursor to next tab stop

This command moves the cursor to the first tab stop to the right of the present cursor position, or to the first character position in the line, if there are no tab stops to the right of the cursor. Tab stops are set and cleared by means of the TS command (one of the "extended commands" described in Section 3.5, below).

3.2 STRING SEARCH

These commands permit the user to enter a string of characters and then search the file quickly for occurrences of the string.

CTRL/ O string search

When this command is given, the screen is cleared and a colon (:) appears on line 16. At this point the editor expects to receive an input line consisting of one to thirty-nine characters and a carriage return. The input line becomes the search string. Any occurrence of the string, regardless what characters precede or follow it, will be recognized as a match. Therefore, it is unnecessary to enter more characters than are required to define the text uniquely, i.e, "the qu" can be used locate a line containing "the quick brown fox."

After accepting the input string, the editor searches the file, beginning one line below the current cursor line and ending at a matching string or the end of the file, whichever is encountered first. Upon finding a matching string, the editor positions the line containing the match at the first line on the screen. If no match is found before the end of the file, the the first sixteen lines of the file are displayed. If a match is found, the search may be continued to the next match of the same string; see the CTRL/ L command below. To search the entire file, type the CTRL/ and V keys, then the command FL (see discussion of these commands, below). Next type CTRL/ and O, and enter the search string. If the CTRL/ L command is used to search from each string match to the next, all occurrences of the input string will eventually be found.

CTRL/ L or LOAD continue string search

After a string has been located with the CTRL/ O command, the CTRL/ L command may be used to search for the next occurrence of the same string. This command causes the editor to start searching with the first line below the current cursor line and continue until a matching string is found or until the end of file is reached. The CTRL/ L command may be given as often as is desired.

3.3 FILE MODIFICATION

CTRL/ T character insert mode switch (on-off-on....)

Any input from the keyboard, other than an editor command, is placed in the file in one of two modes. These modes, normal and insert, are determined by the CTRL/ T command. Each entry of CTRL/ T changes the input mode, i.e., if insert mode is on when command is given, the command will turn it off.

When insert mode is OFF, as it is when the editor is loaded and whenever the cursor is move to a new line, each input character is placed at the cursor location, and the cursor moves to the right one place. When insert mode is on, however, each character is inserted into the file at the cursor position, moving the character at that location, and any any characters to the right of the cursor, one position to the the right. If the line resulting from such an insertion is more than sixty-four characters long, all characters after the sixty-fourth are lost.

CTRL/ H delete character mode

The delete character command removes the character at the current cursor position and moves each character to the right of the cursor one position to the left.

CTRL/ B insert line command

The line insert control moves the file down one line from the cursor, and inserts a new blank line at the old cursor line position. The cursor is moved to the first character position of the new line. Use this command to insert a new line 'above' the current line.

CTRL/ M or RETURN scroll up & insert line

Carriage return scrolls up one line and inserts a blank line in the file. The cursor is moved to the first character position of the new line. Use RETURN to insert a line 'BELOW' the current cursor position. No characters on the current line are deleted.

CTRL/ P delete line command

This control removes the current cursor line from the file.

CTRL/ J or LINEFEED blank remaining line

Linefeed deletes the character at the cursor position and all characters in that line to the right of the cursor. After the characters have been deleted, the cursor is placed at the first character position of the following line, and the file is scrolled up one line.

3.4 EDITOR CONTROL

CTRL/ Y repeat command

This command has three parts: 1) the CTRL and Y keys typed together, 2) the command to be repeated (probably a combination of CTRL and some other key), and 3) a character representing the number of times the command should be repeated. The three parts of the command should be entered in order, without blanks between them. Enter a carriage return only after the third item in the command.

During the execution of a CTRL/ Y command, the cursor disappears from the screen. If you are editing a file and the cursor unexpectedly disappears, assume that you have given the command by accident, and enter a repeat command that will not affect the contents of your file, e.g., repeat the CTRL/ S command once.

The editor determines how many times a command will be repeated by considering the third item of the command with an ASCII bias of 30 Hexadecimal. Thus, the characters 1 through 9 are taken to represent the corresponding numerical values; to designate a numerical value greater than 9, type the character whose hexadecimal ASCII value is 30 greater than the number you want to represent. (30 Hexadecimal is 48 Decimal.)

EXAMPLE: To delete 9 characters from the current cursor position, type

CTRL/ Y CTRL/ H 9 <cr>

LEAVING OUT THE BLANKS.

CTRL/ F

This command is the normal means of exit from the editor. After giving this command from anywhere in the text area, the user is asked to confirm the command by typing the letter Y. If the command was entered accidentally, the user can type another letter and return to PTDOS. (See the section about LEAVING THE EDITOR.)

ESCape

This command is used to abort an EDIT and return to PTDOS. After giving this command from anywhere in the text area, the user is asked to confirm the command by typing the letter Y. If the command was entered accidentally, the user can type another letter and return to the text area. (See the section about LEAVING THE EDITOR.)

3.5 EXTENDED COMMANDS

CTRL/ V invoke extended commands

The function of the CTRL/ V command is really to make ten other commands available. Three of these commands determine the function

to be associated with CTRL/ U (until that function is altered by another CTRL/ V), one command determines the tab stops to be considered by CTRL-I, and the other six are direct commands. When the CTRL/ and V keys have been entered, the screen is cleared and a colon (:) appears on line 16. The user is expected to type the code for the desired function immediately after the colon.

TA <cr> Set CTRL/ U for tab function

When CTRL/ U is set for the TAB function, the CTRL/ U command will expect one more keystroke to be typed in by the user. This value will be used to tab the cursor to the corresponding character position in the cursor line. To indicate a given character position, type the character whose ASCII value is 3Ø Hex greater than the value you want to represent. (The numbering of character positions starts at Ø, so type Ø to place the cursor over the leftmost character in a line.

When the EDIT program is loaded, the CTRL/ U command is set to perform the tab function. If either block move or null mode has been set since the program was loaded, type TA or TAB (and a carriage return) after the colon. The file area will immediately be displayed again, with file and cursor positions just as they were when the CTRL/ and V keys were typed.

BM <cr> Set CTRL/ U for block move function

After CTRL/ V is typed and the colon (:) prompt is printed on the screen, type BM and a carriage return to set the CTRL/ U command to move blocks of text from one position to another within the file. The procedure for moving blocks is as follows:

1) If you have not turned on the block move CTRL/ U option, do so now (as above).

2) Insert a new blank line above the first line of text to be moved, and type the letter 'F' (first) in the FIRST character position of the new line.

3) Next insert a new blank line below the last line of text to be moved, and type the letter 'L' (last) text to be moved, and type the letter 'L' (last) in the FIRST character position of the new line.

4) Now, go to the line location of the file where the block of text lines is to be moved, and insert a new blank line there. Type the letter 'I' (insert) in the FIRST character position of the new line.

5) Now type CTRL/ U

The block of lines inclosed within the 'F' and 'L' will be removed from its old position in the file, and inserted into the file at the 'I.' The file will then be displayed with the first line of the moved text as the first (top) line of the screen. The 'F', 'L', & 'I' lines will be removed from the file.

NU <cr> Set CTRL/ U for no function

Typing this command in response to the CTRL/ V prompt will turn the CTRL/ U functions off, i.e., CTRL/ U will have no effect if typed.

FL <cr> Position file to first line

The command causes the file to be displayed on the screen with the first line at the top of the screen.

LL <cr> Position file to last line

This command causes the file to be displayed on the screen with the last line of the file at the bottom of the screen.

ST <cr> Set start of file

This command deletes all text above the cursor line, so that the cursor line becomes the first line of the file. Follow these instructions carefully to use ST:

1) Position the cursor to the desired line.

2) Type CTRL/ V (colon prompt will appear).

3) Type ST and a carriage return. The file will be displayed with the new first line at the top of the screen.

All information above the cursor line will be lost if the file is written back onto the disk after this operation! If you have already given the ST command and decide you really do not want to have lost everything above the cursor, type the ESCAPE key to abort the edit operation, and re-load the file with a new EDIT command. (If you abort EDIT in this manner, you will lose ALL alterations you have made in the file since you last loaded the EDIT program.)

EN <cr> Set end of file

This command deletes the cursor line and all text below it.

1) Position the cursor to a line ONE LINE BELOW the line that you want at the end of the file.

2) Type CTRL/ V (colon prompt will appear).

3) Type EN and a carriage return. The last sixteen lines of the file will be displayed on the screen.

Just as with the ST command, the material that you delete with EN will no longer exist in the file when you write it back out to the disk.

NOTE: how to divide a large file into smaller files

1) Enter the EDIT command, specifying the large file as the <from file> and creating a new <to file>.

2) Set the cursor in the middle of the file, enter the CTRL/ V command, and use EN so that only the first half of the previous file remains in the file area.

3) Enter the CTRL/ F command and write the contents of the file area to the new <to file>.

4) Enter the EDIT command, specifying the large file as the <from file> and creating a second new <to file>.

5) Set the cursor to the same position as in step 2, enter the CTRL/ V command, and use ST so that only the second half of the previous file remains in the file area.

6) Enter the CTRL/ F command and write the contents of the file area to the second new <to file>.

7) Kill the large file using the PTDOS KILL command.

PA R{S} <cr> replace pattern

This command allows the user to search for and replace or delete text patterns within a file.

There are variants of the pattern replacement command:

```
PA RS  -  Selective Pattern Replacement
PA R   -  Pattern Replacement
PA DS  -  Selective Pattern Deletion
PA  D  -  Pattern Deletion
```

After either of the first two variants is typed, the pattern program will display:

Pattern:

Type the pattern for which to search the file, and then type the RETURN key. The program will display:

Replace:

Type the replacement pattern, and they type the RETURN key.

The DELete key may be used to correct an error in the typing of a pattern.

Upon finding a pattern match in the file, the first form of the pattern command will display the line in which the match was made and position the cursor directly below the first character of the matched text. The user now has the option of allowing the replacement to occur on this line (type any character other than 'N') or leaving the

line as it is (type the letter 'N'). Use this form if you are not sure whether to replace all occurrences of the pattern in the file. If you were to replace all occurrences of 'POP' with 'PUSH', and a line of the text contained the label 'POP1', the replacement of the characters 'POP' would produce 'PUSH1'.

If the form PA R is used, all replacements will be made with no display of matched lines.

It is possible to replace a text string with a new string of a different character count. In this case, the file will be expanded or contracted as needed. This may take some time, for example; replacing all of the single spaces on this page with a single # sign would only take a second. But replacing all of the spaces with say, four # signs (####) would take somewhat longer, since each replacement would require moving the following file down three spaces for each replacement.

It is possible to overflow a line of text by replacing text patterns with new patterns of greater length. In this case, the pattern program displays the offending line and a message to the effect that overflow would occur if the replacement were made. The replacement is then not made. Type any key except the ESCape key to continue the pattern replacement procedure. Type the ESCAPE key to abort the pattern replacement procedure; then enter a carriage return to return to the text area.

At the completion of pattern replacement a message will be printed giving the number of times the pattern was replaced. Enter a carriage return to return to the text area.

The other two forms of the command result in the deletion of the indicated pattern. PA D and PA DS are similar in operation to PA R and PA RS, respectively, except that the user is not asked to supply a replacement pattern, because no patterns are to be replaced.

At any time during the pattern replacement procedure the ESCAPE key may be typed to abort the operation. All replacements made up to this point will remain, but no further replacements will be made.

IF <file name> <cr> insert file

If this command is typed after the colon prompt (:), the named file (an existing PTDOS file with a legal filename) will be inserted into the file in memory. The insertion begins just BEFORE the current cursor position. The first line to be inserted pushes all characters on the right side of the cursor RIGHTWARD in the text area (as though that line had been inserted with the CTRL/ T command). All subsequent lines are inserted as a block, pushing the existing text downward; the file will expand to accommodate new lines (just as though they were entered from the keyboard) ,unless the insert operation would overflow the file area.

When the insertion is completed, the cursor is positioned on a blank line following the last line that was inserted.

TS<cr> set or clear tab stop(s)

If this command is typed after the colon prompt (:), a row of digits appears on the next line, with the cursor flashing in the first character position, and inverse video indicating current tab stops. "Inverse video" means that on a screen that represents characters as black on a white background, a character at a tab stop will be white on a black background (and vice versa for the white-on-black setting of the screen).

To set or clear a tab, use cursor control keys or the corresponding control characters to move the cursor to the character position whose status is to be changed; then strike CTRL-I or the TAB key. If that character position was previously a tab stop, it will no longer be; if it was not a tab stop, it will be set. Use a carriage return to exit the tab setting mode.

To move the cursor to the next tab stop, give the CTRL-I or TAB command in normal editing mode. If there are no tab stops set, or if no tab stops are set to the right of the cursor, the cursor will be moved to the first column of the current line.

EDT3*

A TEXT EDITOR FOR PTDOS 1.5

TABLE OF CONTENTS

EDT3

*This manual describes EDT3 1.1. EDT3 was derived from EDIT3, a product of LSM Engineering. Portions of this manual are subject to copyright by LSM Engineering and are used here with permission of the author.

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

SECTION 1

INTRODUCTION

1.1 CAPABILITIES

EDT3 is a text editor program that allows for the creation or modification of ASCII files such as source files coded in FORTRAN, BASIC, or Assembly Language. This program allows editing on character, string, line and page levels; at any of these levels additions, insertions, substitutions and deletions of text may be made. Additionally, EDT3 offers the option to retain a command string as a macro and execute it repeatedly. A special feature of EDT3 makes it possible to insert an unprintable character into a text file.

The EDT3 program resides in low memory and requires approximately 4K. EDT3 uses the remaining portion of memory as the text buffer area, reserving two areas: one for use as a command buffer, another for use as a macro buffer.

EDT3 receives its text from two sources: it reads data from disk files or allows creation of new text on-line from the keyboard. An input file is regarded as a single character string, usually organized into pages. Upon command, EDT3 will read one page of text from a disk file; that is, reading will progress until a page terminator (formfeed character) is encountered in the file, or until the buffer is full. After the text has been edited, it may be stored with or without the page terminator. (A text without terminators will be regarded by EDT3 as a single page.)

It is possible to use this diskette version of EDT3 to write a cassette file. While in PTDOS, assemble the CTAP:S file on your system diskette, and name the object file CTAP1.

```
*ASSM CTAP:S,,CTAP1
```

After you have entered EDT3, use the > command (described in Section 5, below) to open CTAP1 for output. The resulting tape file will have the name PTFIL and a block size of 1024.

1.2 CONVENTIONS

The following conventions are used in the examples throughout this manual:

1) An ESCape key entered by the operator is echoed to the console and represented in the examples as a dollar sign ($).

2) Control characters in this manual are represented by a "^" followed by the character depressed in conjunction with the control key, e.g., ^P denotes a control P. In general, this representation corresponds to the the way control characters actually appear on the console. Control characters that function as commands to EDT3 are exceptions, e.g., ^A appears as #, ^P is not echoed to the console at all.

3) Whenever a string parameter is part of a command format, it is represented in lower case (e.g., Sstring$ represents a command which might actually be entered as Sgoto, or SBLACK, followed by an ESCape.) In the text that accompanies examples a string is set off by quotation marks. For example, the command CAB$CD$$ appears expanded in the explanation as: C "AB" to "CD".

4) Lower case n represents a positive or negative decimal integer which, when it precedes a command, is related to that command in a certain quantitative way: for example, it can indicate the number of times an action is to be performed. The legal values of a quantifier are given in connection with a specific command when it is discussed. (-65535 to 65535 is the maximum range.) A plus sign before a quantifier is for clarification; its use is optional.

5) A <cr> in an example represents the entry of a carriage-return.

1.3 DEFINITIONS

An input file is a continuous string of characters that EDT3 reads, page by page, from a disk file or receives on-line from the keyboard. A formfeed character marks the end of a page; a page includes all of the characters up to, but not including, the formfeed. The formfeed is not retained within the text buffer; if desired, a formfeed may be written to the output file following a page of text.

A page may be segmented into lines. Each line is a string of characters up to and including the carriage return. A linefeed is assumed after each carriage return, although it does not actually appear as a character within the file.

EDT3 maintains a character pointer (CP) within the text buffer. This pointer is moved through the text by various commands; it should be regarded as always pointing between two characters, rather than at a particular character.

SECTION 2

ACCESSING THE EDT3 PROGRAM

2.1 LOADING

EDT3 is loaded at location 100H. To load the program, type the PTDOS command

*EDT3<cr>

(Do not type the asterisk; it is the PTDOS prompt.)

2.2 EXECUTION

EDT3 is started at location 103 Hex; when the program begins to run, it first clears its scratch area, macro, and command buffers, then searches contiguous random access memory (RAM) to find the last (highest) location that the text buffer can occupy. That address will never be higher than the lowest PTDOS system address, GLLOW.

EDT3 then displays:

```
EDIT 3.0 (mod 0) ZZZZZ
:
```

Where ZZZZZ is the total character space (decimal) available in the text buffer

: is EDT3's prompt

If ZZZZZ is a smaller number than you would expect to find available, given the amount of memory in your computer, EDT3 has probably encountered either read-only memory, or a bad random-access memory location, and discontinued its search for text buffer space.

The text buffer initially takes up all available memory space. The contents of memory (routines, data, etc.) are not altered until text is entered. The last location for memory usage may be specified at any time after initialization is complete. (The memory size command is described in Section 11.)

In case of accidental exit, EDT3 may be restarted by executing address 100H, bypassing all clearing and initialization. If memory has not been changed externally, the text buffer will remain intact. (There are commands which enable you to leave EDT3 intentionally, to execute some other routine or to return to PTDOS. The G and H commands, which serve these functions, are described in Section 10 of this manual.)

SECTION 3

LANGUAGE ELEMENTS

3.1 COMMAND FORMAT

A single instruction to EDT3 has one of the formats listed below. (Spaces are not required of any EDT3 commands, except when desired within strings, and are used in this manual only for clarity.)

1. <command>
2. n <command>
3. <command> string $
4. <command> string1 $ string2 $
5. <command> <hex address> $

The <command> portion is a one or two character mnemonic (such as W or X?). Commands may be typed in upper or lower case.

A signed integer, the n parameter, may be used to quantify a command, that is, to indicate how many times a command is to be executed or how many characters are involved. EDT3 interprets a quantifier as a decimal number. Decimal values may range from -65535 to +65535. (If the operator enters a quantifier where it is not required, EDT3 executes the command, ignores the quantifier, and returns no error message. If a negative quantifier is specified, but is meaningless to the command, it is taken to be positive; no error message is displayed.)

Some commands require at least one string. A string may be zero or more characters in length, and is terminated by the ESCape key (echoed as the $ symbol on the operator's console). A string may include the carriage return character.

EDT3 begins execution of most commands when two consecutive ESCapes are entered. (One of these may be a string terminator.) The commands ^P, ^R, and ^T are exceptions to this rule: they are executed immediately upon being entered by the operator.

NOTE: If you have typed in a command and it appears not to have been accepted properly, make sure that you have actually typed in ESCapes, rather than dollar signs, as command terminators.

EXAMPLES:

: 5L$$ The command to move the cursor to the beginning of the fifth line following its present line position must be terminated by two ESCapes.

: SLXI$$ The command to search for the string "LXI" requires one ESCape as a string terminator, and a second as a command terminator.

3.2 COMMAND SUMMARY

Here is an overview of the commands to be used with EDT3. Note that each command is accompanied by 1) a very brief explanation of its function, and 2) a reference pointing to the section of this manual where a more complete explanation may be found.

The abbreviation, "CP", here and elsewhere in the manual, stands for Character Pointer, as defined in Section 1.3, above.

There are a number of search commands which are listed under COMBINED DISK INPUT AND OUTPUT, rather than under CP CONTROL, because they have tape input/output functions associated with them.

COMMAND	FUNCTION	REFERENCE
SPECIAL CHARACTERS (immediate execution)		
DEL or RUBOUT	Deletes the last character entered.	3.4.1
MODE SELECT or ^@	Cancels a current command string, or halts its execution.	3.4.2
^T	Turns off tab simulation, if on; turns it on, if off. Tabs are predefined at 8-space intervals. There is no provision for changing them.	4.
SPACE-BAR	Alternately suspends and continues output to the console display during P command execution.	4.
^P	Prints the last command string.	12.2
^R	Re-executes the last command string.	12.3

CONSOLE INPUT/OUTPUT

T	Prints buffer on the console output device.	4.
nT	Prints n lines, beginning at the CP, on the console output device.	4.
nTN	Specifies number of nulls to be sent after each carriage return/ line feed on subsequent T commands.	4.
nTW	Specifies the width of an output line (n characters) on subsequent T commands.	4.

OPENING AND CLOSING FILES

<filename$	Opens a file for input.	5.1.1
>filename$	Opens a file for output.	
n;	Sets output file block size.	5.1.2
<=	Prints name of current input file.	5.1.3
>=	Prints name of current output file.	
<$	Closes current input file.	5.1.4
>$	Closes current output file.	

INPUT FROM A DISK FILE

Y	Clears the previous contents of the text buffer, without writing it, and reads the next page.	5.3.2
A	Reads the next page, and appends the input to the current contents of the text buffer.	5.2.2

OUTPUT TO A DISK FILE

P	Writes the entire text buffer with a final formfeed.	5.3.1
nP	Writes n lines from the CP and a final formfeed.	5.3.1
PW	Writes the entire text buffer without a final formfeed.	5.3.2
nPW	Writes n lines from the CP and no final formfeed.	5.3.2
PE	Writes the entire text buffer with a file terminator. Closes output file.	5.3.3
nPE	Writes n lines from the CP with a file terminator. Closes the output file unless n = Ø.	5.3.3

COMBINED DISK INPUT AND OUTPUT

E	Copies current buffer and remainder of input to output file; closes output file.	5.4.1
nR	Executes, n times, the sequence: P, then Y.	5.4.2
Nstring$	Searches buffer for "string"; continues search, page by page, each time writing out the buffer (P) before proceeding to the next page (Y).	7.3
Qstring$	Like Nstring$, but does not write out the text buffer before proceeding to the next page.	7.3
Ostring1$string2$	Changes "string1" to "string2", searching the whole file by doing P's and Y's, as necessary.	8.2.3
Ostring$$	Deletes "string" (changes it to null), searching the file by doing P's and Y's, as necessary.	8.2.3

BUFFER INFORMATION

=	Displays the total number of lines and characters in the text buffer.	6.
@	Displays number of the line in which the CP resides.	6.
W	Returns the number of characters that PRECEDE the CP in the line in which it resides.	6.
:hex address$	Specifies the last available memory location for the text buffer.	11.

CP CONTROL

B	Moves the CP to the beginning of the text buffer.	7.1
Z	Moves the CP to the end of the text buffer.	7.1
L	Moves the CP to the beginning of the current line.	7.2
+ or - nL	Moves the CP forward (+) or backward (-) n lines.	7.2
nJ	Moves the CP to the beginning of the nth line in the buffer. + or - nM	7.2
	Moves the CP forward (+) or backward (-) n characters.	7.3
Sstring$	Moves the CP to the first character after "string".	7.3

Also see N and Q commands in combined disk input and output, above.

ADDITION/INSERTION

Istring$	Inserts a string of characters at the CP.	8.1.1
nI	Inserts the character whose decimal ASCII value is n at the CP.	8.1.2

DELETION/SUBSTITUTION

K	Deletes the entire line, no matter where the CP is located on it.	8.2.1
nK	Deletes from the CP forward over n carriage returns.	8.2.1
+ or - nD	Deletes from the CP forward (+) or backward (-) n characters.	8.2.2
Cstring1$string2$	Changes "string1" to "string2".	8.2.3
Cstring$$	Deletes "string" (changes it it to null).	8.2.3

Also see O commands in COMBINED DISK INPUT AND OUTPUT, above.

MACROS

XMcommand string$$	Defines a macro command string.	9.2
nX	Executes the macro n times.	9.3
XD	Deletes the macro.	9.4
X?	Prints the macro.	9.5

LEAVING EDT3

Ghex address$	Go to an external user routine.	10.2
H	Close files; return to PTDOS.	10.3
?	Go to the Command Interpreter for one command.	10.4

3.3 COMMAND STRINGS

EDT3 is able to execute, not only a single command, but also a group of commands entered as a series. As each command is entered into what becomes a command string, it is placed into the command buffer. An ESCape between commands in a command string is optional, unless either 1) a command includes a string parameter, or 2) the following command might otherwise be construed as part of the first command (e.g., the command string X$D must be differentiated from the single command XD).

EXAMPLES:

: Y55L5T$$	This command string consists of three commands: Y, 55L, and 5T. The command string is terminated by two ESCapes.
: SADD$L1TC1$2$$	This command string consists of four commands: S "ADD", L, 1T, and C "1" to "2". Note that only the strings, in the S and C commands, require the ESCape separator, and that when a character string is the final item in a command string, only one additional ESCape is needed.
: SFILE$ <cr>	This command string consists of four commands: L<cr> S "FILE", L and 5T. A carriage return may be used as a visual separator between commands: in a command string; such carriage returns do not affect the way that the command string is executed.

3.4 COMMAND ERRORS

EDT3 executes a command string one command at a time. If EDT3 encounters a command that cannot be executed, it will print out an error message and the unexecuted portion of the command string, and it will clear the command buffer. EDT3 will ignore a quantifier if it appears without a command.

EXAMPLES:

: YABCZZ33LK$$	Six commands: Y, A, B, C "ZZ" to "33", L,
String not found	and K. EDT3 types an error message
? ? ?	("String not found" and "? ? ?"), the un-
CZZ33LK$$	executed portion of the command string and
:	the prompt symbol.

CAUTION: A carriage return within a text string or between the characters of a two-character command is not ignored.

AVOID:

: X <cr> D$$	EDT3 executes the XD (a legal command) as X, also a legal command, then deletes one character.

If an error is made while typing a command, it may be corrected in one of the following ways:

KEY	FUNCTION
DEL or RUBOUT	Deletes last character entered. Cursor backspaces.
MODE SELECT or ^@	Cancels a current command string or halts its execution.

3.4.1 Character Level: DEL or RUBOUT

As commands are entered, they are stored in the command buffer. DEL or RUBOUT deletes the last character entered in the command buffer. Several DELs may be entered to delete several characters. If the deletion empties the command buffer, EDT3 issues another prompt.

3.4.2 Command Level: MODE SELECT or ^@

Before a command string is terminated, it can be canceled by issuing a MODE SELECT or ^@. EDT3 stops, empties the command buffer, and issues another prompt. If the interrupted command is performing I/Ø, the Character Pointer is set to the beginning of the text buffer; otherwise, it is left in its current position (nP, nPW, or nPE), or at the end of the previous text (A).

3.4.3 Command Buffer

The command buffer is of sufficient length (124 characters) to accommodate long command strings; if the command buffer length is exceeded, the command input, up to that point, will be executed. The exception to this rule is that the insert command will allow input until the text buffer is filled.

3.5 SPECIAL CHARACTER HANDLING

Several input characters receive special handling by EDT3, depending upon whether the character originates from the console or from a disk file. Where the word "normal" appears in the table below, it is used to indicate that a particular control character, where it is read or inserted into the text buffer, is regarded as though it were any other text character, i.e., it does not initiate any action by the program or the system. Exceptions are noted below.

CHARACTER	HEX	FROM KEYBOARD	FROM DISK FILE
MODE SELECT or NUL, or ^@	ØØ	Executed, not echoed, not retained	Ignored
^A	Ø1	Echoed as #, retained as ^A in the command string	Normal
LINEFEED or LF or ^J	ØA	Not executed	Ignored
DEL or RUBOUT	7F	Executed, not retained	Normal
^T		Executed, not retained	Normal character (no tab)
ESC or ^[or ^SHIFT K		Echoed as $, retained as ESC in the command string	Normal
RETURN or CR or ^M		Echoed as CR/LF, retained as CR	Normal
^L or FF or formfeed	ØC	Echoed as special character, retained as formfeed.	Terminates page, then is discarded
^P	1Ø	Executed and not retained if it is the first character of the command. Otherwise normal.	Normal
^R	12	Executed and not retained if it is the first character of the command. Otherwise normal.	Normal

On output to the console, EDT3 provides a linefeed character after each carriage return.

Nulls, cannot be inserted into the text; control-T ESCape and DELete can be inserted using the I command and a quantifier. All other characters of this set may be keyed in directly.

Note that SHIFT-3 (the # symbol) and SHIFT-4 (the $ symbol) are normal characters.

SECTION 4

CONSOLE INPUT/OUTPUT

4.1 SUMMARY OF CONSOLE COMMANDS

All console input and output is done by way of the CONIN, CONTST, and CONOUT entry points on PTDOS. In this manual, console input is assumed to be from a keyboard.

The speed of output to a VDM is altered if a key representing a digit is hit during printing. The digit 1 causes output to be fastest (no delay), whereas 9 causes it to be slowest. Output to the Video Display is suspended temporarily when the user presses the space bar during printing; it will resume when the next key is pressed subsequently. Output may be aborted with the MODE SELECT key.

COMMAND	EFFECT
T	Send buffer to the output device dictated by CONOUT.
nT	Send n lines following the CP to the output device dictated by CONOUT.
nTN	Specify number of nulls (n) to be sent after each carriage return/linefeed for subsequent T commands.
nTW	Specify output line to be n characters wide for subsequent T commands.
^T	Toggle tabbing mode (on/off).

EXAMPLES:

```
: 2T$$                  The command 2T causes two lines, starting at
DOIT:  STA DPEX         the Character Pointer, to be printed on the
       LDA FLAG         current output device.

: =10T$$                This command string has two commands in it
                        (= and 10T).
2/24                    There are two lines, or twenty-four characters,
TUIT:  ANA B            in the buffer.  n exceeds this number, even
       STA TABLE        assuming that the CP is positioned at the
                        beginning of the text.  Therefore, only two
                        of the ten requested lines are printed.

: SLDA$1T$$             After a SEARCH for "LDA," typing starts at
        PINK            the current position of the CP.  To examine
: SLDA$L1T$$            the entire line, issue the L command before
       LDA PINK         the  nT  command.
```

4.2 NULL: nTN

This command determines how many nulls will be sent to the console after each carriage return or linefeed; n may have a value from Ø to 255. The purpose of sending nulls is to establish a delay that certain output devices require after a carriage return.

4.3 WIDTH: nTW

This command sets the maximum length of an output line, probably corresponding to the width of the console output device. n may have an absolute value of 16 to 255; after EDT3 has printed that number of characters on the console, it will automatically execute a carriage return. (That carriage return is not stored in the text buffer.) If n is negative, the console output device is assumed to be one of the type that backspaces when it receives an underline character (5FH); for example, the width parameter is initialized to -64, which is the specification for a video display like the VDM.

4.4 TABBING: ^T

When the tabbing mode is on, all ^I's (Ø9H) sent to the output device cause spaces to be printed until the next tab stop is reached. Tabs are defined at columns 1, 9, 17, 25, etc. There is no provision for changing these settings. When tabbing is off, ^I's are sent normally.

SECTION 5

DISK FILE INPUT/OUTPUT

5.1 OPENING AND CLOSING DISK FILES

COMMAND	EFFECT
<filename$	Opens the file <filename> for input.
>filename$	Opens the file <filename> for output.
n;	Sets to n the block size for any output file that must be created.
<=	Prints the name of the current input file.
>=	Prints the name of the current output file.
<$	Closes the current input file.
>$	Closes the current output file.

5.1.1 Open File: <filename$ or >filename$

The < command opens the specified file for input operations. The > command opens the specified file for output operations. If there is a file open for either input or output and the corresponding command is given, the open file will be closed, and the named file will be opened.

EXAMPLES:

```
: <TUNA$$  Opens the file TUNA for Y, A, E, R, N, O, AND Q
:          commands.

: >FISH$$  Opens the file FISH for P, E, R, N, O, AND Q commands.
:
```

A legal filename consists of up to eight ASCII characters, none of which may be a blank or a slash. A filename may be followed by a slash and a unit number; the unit number is the number of the disk drive unit in which the disk containing the file resides. If the named input file does not exist on the specified unit, PTDOS will return an error; if the named output file does not exist, it will be created.

5.1.2 Set Block Size: n;

Output files are written on the disk in blocks. If the output file that will be named in the > command does not exist, it must be assigned a block size, i.e., the number of bytes to be put in each block. The default block size is 4CØH; the maximum block size is FFFH. The larger the block size, the quicker disk access is, by virtue of the reduced number of inter-block gaps. The block size must be set BEFORE the output file is opened.

5.1.3 Print Filename: <= or >=

The <= command prints the current input <filename>; the >= command prints the current output <filename>. If the corresponding file is not open, EDT3 will respond with a blank line.

EXAMPLE:

```
: <=$$
TUNA        The last < command was "<TUNA$".
:

: >=$$
FISH        The last > command was ">FISH$".
:

: >=$$
            No output file is open.
:
```

5.1.4 Close File: <$ or >$

The <$ command closes the current input file. The >$ command closes the current output file. If the corresponding file was not open, then the command has no effect.

EXAMPLE:

```
: <$<=$$  Closes the input file and then tries to print its name.
:
```

Any file written in EDT3 must be either closed or endfiled before an attempt is made to read from it. The Put and Endfile (PE) command will automatically close a file; also, leaving EDT3 will result in the closing of all files open at the time that the command is entered. Neither leaving nor closing will endfile an open output file. To end a file without closing it, give the command ØPE$$.

5.2 INPUT FROM A DISK FILE

The following commands allow input from a file that is opened:

COMMAND	EFFECT
Y	Reads the next page into the text buffer and overwrites the previous buffer contents.
A	Reads the next page into the text buffer, appending the input to the end of the current buffer contents.
Q	See Section 7.3

Any quantifier preceding these commands is ignored. Note that the formfeed page terminator is not retained within the text buffer and all input characters are masked to 7 bits.

5.2.1 Yank: Y

The Y command always destroys the current contents of the text buffer, and attempts to read in a page from the assigned file. The CP is positioned before the first character of the new page. If no page is available, EDT3 issues an error message; EDT3 does not complain if the next page is empty (formfeed only).

EXAMPLE:

```
YY$$
```

Yanks one page, then immediately yanks another page from the assigned file. The first page of text yanked is destroyed, but the next page is available for editing.

If the input fills the text buffer before a formfeed is encountered, EDT3 types the "Buffer full" error message. A hundred bytes have been reserved in the text buffer, so that some editing may still be done before the text, or part of it, is written out (See also Section 5.2.2).

5.2.2 Append: A

The A command does not destroy the previous contents of the text buffer. It appends the subsequent page to the current contents. Unless a formfeed character is inserted, the two pages are now concatenated. The CP is positioned before the first character of the appended page.

EXAMPLE:

```
: YAA$$
```

Yank one page, then immediately append two additional pages.

If the input fills the text buffer before the end of file or a formfeed is encountered, EDT3 types the "Buffer full" error message and positions the CP at the beginning of the text buffer.

Some text may be written out of the text buffer, and then deleted from it, to clear space. Some text should be deleted in this manner if another append is desired.

EXAMPLE:

```
: AAAAA$$     Attempt to append five additional pages.
Buffer full
? ? ?
AAA$$         EDT3 types out the unexecuted portion of the
              command string.
```

5.3 OUTPUT TO A DISK FILE

The following commands allow output of a given number of lines or the entire text buffer:

COMMAND	EFFECT
P	Writes the entire text buffer with a final formfeed.
nP	Writes n lines, starting at the CP, and a final formfeed.
PW	Writes the entire text buffer without a final formfeed.
nPW	Writes n lines, starting at the CP, without a final formfeed.
PE	Writes the entire text buffer and endfiles the output file.
nPE	Writes n lines, starting at the CP, and endfiles the output file.

These commands do not move the CP; it is used to locate where writing should start when n is specified.

All output characters have the high bit set equal to zero (no parity).

5.3.1 Put: P

The entire contents of the text buffer are written to the assigned file and a formfeed, terminating the page, is written as the final character. If the text buffer is empty, only the formfeed is written. If n is specified, EDT3 writes n lines, starting at the current CP, and a final formfeed; the CP is not moved. If n is Ø, only a formfeed is written.

EXAMPLE:

: 2ØP$$ Twenty lines and a final formfeed are written.

5.3.2 Put without Formfeed: PW

The PW command is like the P command, except that the entire buffer (PW) or the specified number of lines from the current CP (nPW) are written without the final formfeed.

EXAMPLE:

```
: Y$$     Four commands (Y, PW, Y and P) that combine two pages into
: PW$$    one.  The command string YAP$$ could also be used.
: Y$$
: P$$
```

5.3.3 Put and Endfile: PE

The PE command is like the P command, except that the entire text buffer (PE) or the specified number of lines from the current CP (nPE) are written and the file is endfiled. No final formfeed is written. If n is Ø, then the file is just endfiled.

Remember that a file should be either closed, or endfiled, or both, if it is to be used later as an input file.

5.4 COMBINED DISK INPUT AND OUTPUT

COMMAND	EFFECT
E	Copies current buffer and remainder of input file to the output file and closes the output file.
nR	R does a P followed by a Y; n indicates how many times R is to be performed.
N	See Section 7.3.
O	See Section 8.2.3.

None of these commands may be entered after an EOF is encountered on the input file.

5.4.1 End: E

The E command copies the input file page by page, preserving page structure (i.e, formfeeds) until the end of file. When the end of file is encountered, the output file is endfiled and closed, the message "EOF" is printed, and the processing of the command string resumes with the next command.

EXAMPLE:

```
: YCMVI$MOV$E$$  Reads a page, changes the first occurrence of
EOF              MVI to MOV, copies input file to output file and
:                endfiles the latter.
```

5.4.2 Put and Yank: nR

R does a put, followed by a yank, n times. If the end of file is encountered during the yank portion of the command, the R command is terminated immediately and EOF is printed. In such a case, the last page of text has NOT been written to the output file. The processing of the command string resumes with the next command.

EXAMPLE:

```
:Y5RSend$-3D$$  Yanks a page, does a put followed by a yank, etc.,
EOF             until the end of file is encountered during a yank.
                The page has not been written out.
```

SECTION 6

BUFFER CONTENTS

The commands listed below provide information about the text buffer and the position of the Character Pointer.

COMMAND	EFFECT
=	Displays the total number of lines and characters in the text buffer.
@	Displays the number of the line in which the CP is currently positioned.
W	Returns the number of the character immediately after which the CP is positioned.

EDT3 displays each value as a decimal number. If the text buffer is empty, the = command displays a Ø/Ø value (no lines, no characters); the @ command displays 1 (CP is in the first line), and the W command displays Ø.

If a quantifier precedes any of these commands, it is ignored.

EXAMPLES:

```
: =$$
2370/16878  There are currently 2370 lines and 16878 characters
:           in the text buffer.

: Y=$$      As part of a command string of two commands:
33/1109     yank, display total lines and total characters.
:

: @W$$      Where is the CP?
206         On line 206, character 15.
15
:
```

SECTION 7

CONTROLLING THE CHARACTER POINTER

Several commands are available for positioning the Character Pointer at the page, line or character level.

7.1 PAGE LEVEL: B and Z

The following commands move the Character Pointer to a specified position relative to the entire page of text being edited.

COMMAND	EFFECT
B	Moves the CP to the beginning of the text buffer (before the first text character).
Z	Moves the CP to the end of the text buffer (after the last text character).

Any quantifiers used with the B and Z commands are ignored; EDT3 executes the command once, and issues no error message.

EXAMPLE:

```
: @B2T$$
287
      LDA FLAG
      ORA A
```

The @ command finds the line position of the CP (287); the B command moves it to the beginning of the buffer and the 2T command types the first two lines in the buffer.

7.2 LINE LEVEL: L and J

The following commands move the Character Pointer to a given line within the buffer.

COMMAND	
L	Moves the CP to the beginning of the current line.
+nL	Moves the CP forward n lines (over n carriage returns).
-nL	Moves the CP backward n lines (over n carriage returns).
nJ	Moves the CP to the beginning of the nth line in the buffer.

The L (line) commands position the CP before the first character of a specified line. Where a quantifier is supplied, it indicates how many carriage returns the CP must encounter in order to reach the appropriate line. In the absence of a quantifier, the current line is assumed; the CP is moved back to the last carriage return and then forward one position.

When moving backward (the minus sign is required) through the text, EDT3 proceeds by counting n+1 carriage returns back, and 1 position forward. The beginning of the buffer is equivalent to a carriage return. If n directs the CP beyond the limits of the buffer, the CP is left positioned after the last character in the buffer (+n) or before the first (-n).

EXAMPLE:

```
: @-3L@$$  CP starts in line 8 and moves backward 3 lines to
8          the beginning of line 5.
5
:
```

The J (jump) command moves the CP to the beginning of the text buffer, then forward over n-1 carriage returns, finally positioning the CP before the first character of the nth line. If n is omitted, the CP is positioned at the beginning of the text buffer. If line n does not exist, EDT3 issues the "? ? ?" error message.

EXAMPLE:

```
: @32J@$$  The CP located in line 47 moves to the beginning
47         of the buffer, then jumps to the 32nd line.
32
```

7.3 CHARACTER LEVEL: M, Sstring$, Qstring$, and Nstring$

The following commands move the Character Pointer to a given character position, either relative to where the CP resides when the command is given, or to a group of characters (i.e., a character string).

COMMAND	EFFECT
+nM	Moves the CP forward n characters.
-nM	Moves the CP backward n characters.
Sstring$	Searches the buffer for "string" and positions the CP after the last character of "string."
Qstring$	Searches the file, page by page, for "string." Positions the CP after the last character of "string."
Nstring$	Like Qstring$, but copies the input file to the output file during the search.

The M (move) command moves the CP forward or backward n character positions through the text buffer. If n is omitted, the CP is moved one position. If n is positive, the CP moves forward n characters and is positioned before the n+1th character. If n is negative, the CP moves backward n characters and is positioned before the nth character back from the current position. If n exceeds the limits of the text, the CP is left positioned after the last character (+n), or before the first character (-n) in the text buffer.

EXAMPLE:

```
: 1T5M1T$$    Three commands:  1T, 5M, and 1T.  The CP moves from
LXI  D,FUN    from the beginning of the line to the D.
D,FUN
:
```

The S (search) command moves the CP forward from its current position while searching for "string" in the text buffer. If "string" is found, the CP is positioned after the last character in "string". The search ends when the first occurrence is encountered. If the text buffer is searched to its end and "string" is not found, EDT3 types the "String not found" error message and positions the CP at the beginning of the text buffer.

```
: 1TSADC$1T$$    Three commands:  1T, S "ADC" and 1T.
      ADC L      After finding the string, the CP is positioned
 L               after the 'C'; the next command (1T) types from
:                the CP to the end of the line.
```

The search string may include masked character positions. Enter ^A (echoed as #) to mark character positions that are to match ANY character.

EXAMPLE:

```
: SA#B#C$L1T$$  The command searches for the occurrence of
A2BXCL 123      A, B and C, each separated from the next by any
:               one character, and types the line.
```

The N and Q commands are similar to S, except that they search the entire file, page by page. If the end of file is reached and "string" is not in the buffer, then the "String not found" message is printed. The N command copies the input file to the output file as it searches, and closes the output file if the search is unsuccessful; if the end of file is encountered, processing of the command string resumes with the next command.

SECTION 8

ALTERING BUFFER CONTENTS

8.1 ADDITION/INSERTION

The following commands allow individual characters and text to be inserted into the text buffer:

COMMAND	EFFECT
Istring$	Inserts a string of characters at the CP.
nI	Inserts the character whose decimal ASCII value is n at the CP.

8.1.1 Text: Istring$

The "string" may be null, one character, several lines, or the entire text buffer in length. The CP is left positioned after the last character of the "string".

EXAMPLE:

```
: I     LDA     UP      The inserted string is composed of three lines.
        MOV     M,A
        INX     H
$$
```

A "long" insertion operation proceeds to fill the command buffer before entering the text into the text buffer; as each line is ended with a carriage return, it is entered into the text buffer. MODE SELECT will cancel only the line being entered and DEL will delete only characters back to the beginning of that line.

8.1.2 Single Character: nI

This command makes it possible to insert into text a character that the keyboard does not have (such as a lower case character) or one that receives special handling. The numeric value of a single character is masked to 7 bits and inserted; the CP is left positioned after the character. The n parameter, which may range from 1 to 126, corresponds to the decimal ASCII representation of the character to be inserted. Formfeeds (ØCH) may be inserted in this way to divide the text buffer into two or more pages.

Exceptions: NUL (ØØH), are ignored.

EXAMPLES:

```
: 16I$$   Inserts the decimal value of ^P into the text buffer :
at the current Character Position.
```

EXAMPLE OF ERROR:

```
: 128ILlT$$          Attempted to enter three commands:  inserting
                     the decimal integer 128, L and lT.  EDT3
? ? ?                masks the converted integer to 7 bits, so it
1281LlT$$            becomes Ø, and aborts command string processing
                     with an error message, since the insert
                     command would have tried to insert a NUL.
```

8.2 DELETION/SUBSTITUTION

Several commands are available to delete text at either the line, character or string level. Substitution of text or a null string (effective deletion) is available only at the string level.

8.2.1 Line Level: K

COMMAND	EFFECT
K	Deletes the entire current line, including the carriage return.
+nK	Deletes from the CP, forward over n carriage returns.

The K (kill) command counts carriage returns and regards a line as all the characters up to and including the carriage return. The K alone deletes the entire current line, no matter where the CP is located on it. If n is specified, EDT3 deletes the remainder of the current line, as well as all text encountered for the next n carriage returns. The limits of the buffer are the limits of the K command.

EXAMPLES:

```
: 4JlT$$             Jumps to the 4th line and displays it.
       STA TABF
: B3KlT$$            Moves the CP to the beginning of the buffer,
       STA TABF      and kills three lines; the old line 4 is now
                     the first line in the buffer.
```

8.2.2 Character Level: D

COMMAND	EFFECT
+nD	Deletes from the CP forward n characters.
-nD	Deletes from the CP backward n characters.

The D (delete) command, like the M (move) command, is easier to understand if the CP is regarded as pointing between two characters. The limits of the buffer are the limits of the command.

If n is Ø or omitted, one character is deleted.

Since a carriage return is one character, this command may also be used to join lines by deleting the carriage return.

EXAMPLES:

```
: lTS $lDLlT$$      Five commands:  lT, S ' ', lD, L and lT.
JMP @l              The command string types the current line, finds
JMP l               the space, deletes the following character, moves
:                   the CP to the start of the line and displays the
                    entire line.

: B2T$$             The command string types the original two lines.
     XCHG
     RET
: SG$lDLlT$$        Four commands:  S "G", lD, L and lT.  After a
      XCHG  RET     search for the "G", the next character, a
:                   carriage return, is deleted, so that the two
                    lines previously separated by the carriage return
                    are concatenated.
```

8.2.3 String Level: C or O

COMMAND	EFFECT
Cstringl$string2	Changes "stringl" to "string2".
Cstring$$	Deletes "string" (changes it to null).
Ostringl$string2$	Changes "stringl" to "string2," searching the entire file while doing P's (puts) and Y's (yanks)
Ostring$$	Deletes "string," searching the entire file, entire file, if necessary.

The C (change) command moves the CP forward from its current position

while searching for the first occurrence of "string1". When located, the string is deleted and replaced with "string2"; the CP is positioned after the last character of "string2". If the end of the text buffer is reached before "string1" is found, EDT3 types the "String not found" error message, and moves the CP to the start of the text buffer.

Note that if "string2" is null (Cstring$$), "string" is deleted. This form of the C command will automatically be the last command in a command string since the two ESCapes force execution of the command string.

Any quantifier preceding C is ignored.

EXAMPLES:

```
: CFILE$TEST$L1T$$        Three commands:  C "FILE" to
      LDA       TEST      "TEST", L and 1T.  After replacing
:                         "FILE" with "TEST", EDT3 types the line.

: ABC69$DE$$              The first two commands (A and B) are
                          executed:  the search for "69" was
                          unsuccessful.
String not found

? ? ?
C69$DE$$
```

The search string ("string1") may contain masked character positions. Enter ^A (echoed as #) to mark character positions that are to match ANY character.

EXAMPLE:

```
: CA##B$BA$$    The command searches for the first occurrence
                of A and B separated by any two characters and
                changes those four to BA.
```

The substitute string ("string2") must not contain any mask characters, or a command error will result.

The O command is similar to the C command, except that if "string1" is not found in the buffer, a P (Put) followed by a Y (yank) is done and the search is continued. If the search is unsuccessful, then the "String not found" error message is printed and the output file is closed.

SECTION 9

MACROS

9.1 SUMMARY OF MACRO COMMANDS

EDT3 offers the capability of retaining a command string and executing the string repeatedly by issuing a single command. The command string is stored in the macro buffer. The following commands are associated with macro handling.

COMMAND	EFFECT
XMcommand string$$	Defines a macro command string.
nX	Executes the macro n times.
XD	Deletes the macro.
X?	Prints out the current macro.

9.2 DEFINE MACRO: XMcommand string$$

Any previous macro is destroyed by the definition of a new macro. Any quantifier is ignored when this command is executed. A macro definition becomes the last command in a command string since it is terminated by two ESCapes. The macro command string length is limited to 122 characters.

EXAMPLES:

```
: XMYS$1 $-3D$$   Defines the macro as a command string containing
:                 three commands:  Y, S "$1 " and -3D.

: <TEST$XMY=$$    The command string opens the disk file TEST for
:                 input and then defines a macro containing two
                  commands:  Y and = .
```

9.3 EXECUTE MACRO: X

The commands X, ØX and 1X are equivalent: the command string currently retained in the macro buffer is executed once. If n is greater than 1, the macro is executed n times.

Any positioning of the CP corresponds to the specific commands that are contained in the macro. Any error situation caused by an individual command in the macro causes EDT3 to halt execution, type the appropriate error message, and return control to the operator.

If a macro is undefined or contains the X command (recursive execution of the macro), EDT3 types "Macro error." A macro must not contain other macro commands.

EXAMPLE:

```
: XMYAP$$
: 1ØX$$     Where the macro buffer contains the command string,
:           YAP, executing the command successfully ten times
            produces 1Ø pages from the 2Ø that are read in.
```

As with any command, MODE SELECT or ^@ may be used to halt execution.

9.4 DELETE MACRO: XD

The macro might best be deleted in cases where a powerful macro would destroy the contents of the text buffer if the X command were issued inadvertently.

EXAMPLE:

```
: XD$$   Deletes the macro.
```

9.5 PRINT MACRO: X?

Printing the current macro can save re-entering a complicated command string, or can be used to determine exactly what effect the last X command had.

EXAMPLE:

```
: X?$$
15L15T$$  The last command that defined a macro was:  XM15L15T$$
```

SECTION 10

LEAVING EDT3

10.1 SUMMARY OF EXIT COMMANDS

The user may leave EDT3, using the G command to execute another routine anywhere in memory, or the H command to return to the PTDOS Command Interpreter. This procedure is comparable to the use of a CALL instruction. After either the G or the H command, executing address 0100 Hex will result in a safe re-entry to EDT3, with the buffer intact, unless it has been changed externally.

COMMAND	EFFECT
Ghex address$	Generates a return address and goes to the routine at the given location.
H	Close input and output files and return to PTDOS Command Interpreter.

10.2 GO TO USER ROUTINE: Ghex address$

Leading zeros in the hex address are not required. If G is not the last command in the command string, subsequent commands are held until a proper return to EDT3 is made. The return location is placed on the EDT3 stack; therefore, if the stack is not modified, a RET instruction in the external routine is all that is required to return to command string processing.

10.3 HALT: H

This is the normal method of leaving EDT3. Any files opened by EDT3 will be closed. If H is issued accidentally, the restart address (0100H) may be executed to recover the text buffer. Because the input and output files are closed by this command, the previous state of the output file, and the position of the input file, cannot be recovered by an execution of the restart address.

10.4 GO TO COMMAND INTERPRETER: ?

When this command is entered, the standard PTDOS prompt (*) will appear, and one command may be entered. If the PTDOS command terminates normally, EDT3 will be re-entered automatically; if an error occurs during the execution of the command, the restart address (100H) must be used to re-enter EDT3. If the Command Interpreter input file is not empty when the ? command is given, the command to be executed will be read from that file.

Only those PTDOS commands designated as [SAFE] in Section 1 of the PTDOS Manual may be executed in this context without encroaching upon EDT3. Do not use the ? command to reload or restart EDT3.

SECTION 11

SETTING THE TEXT BUFFER LIMIT

The text buffer uses the highest portion of memory related to EDT3. At system start up, EDT3 takes as much contiguous RAM memory as it can find for the text buffer. The highest address used by the text buffer will never exceed the lowest address assigned to PTDOS (i.e., the GLLOW parameter in the System Global Area); all memory below that address is tested but not changed. If the operator wishes , at any time, to specify the last location to be made available for text, the size command may be issued. Issuing the size command, to limit the text buffer area prior to putting in text, will prevent the possible destruction of memory contents above the specified last available memory location.

COMMAND	EFFECT
:hex address$	Specifies the last available memory location for the text buffer.

The hex address must be higher than the present last location used for text and lower than the end of contiguous memory, or EDT3 issues the "? ? ?" error message.

After the size command is issued, EDT3 will display the total buffer character space (decimal) available.

EXAMPLE:

```
: :1FFF$$  Limit the text buffer not to expand past memory
3446       location 1FFF hex.  EDT3 indicates that there is
           now space for 3446 characters in the text buffer.
```

SECTION 12

IMMEDIATE COMMANDS

12.1 SUMMARY OF IMMEDIATE COMMANDS

There are three single-character commands which are executed immediately upon being entered. They do not affect the command buffer.

COMMAND	EFFECT
^P	Reprint the last command string entered.
^R	Re-execute the last command string entered.
^T	Toggle the tabbing switch. See Section 4.

12.2 PRINT LAST COMMAND STRING: ^P

If ^P is entered as the first character after execution of a command string, that command string is printed. The ^P is not echoed and not retained. If ^P is entered other than as the first character, it is echoed and treated normally. This command is useful as a means of seeing what was just executed, and also as a prelude to the ^R command.

12.3 RE-EXECUTE LAST COMMAND STRING: ^R

If ^R is entered as the first character after execution of a command string, that command string is re-executed. The ^R is not echoed and not retained. If ^R is entered other than as the first character, it is echoed and treated normally. This feature allows the contents of the command buffer to be treated like a second macro.

NOTE: A ^P or ^R may be entered successfully after a ^P or ^R. ^P and ^R will not work after an error has occurred, because the command buffer is cleared when there has been an error.

SECTION 13

ERROR MESSAGES

When an error is encountered, EDT3 types an error message and the unexecuted remainder of the command string, beginning with the command in error.

ERROR MESSAGE	EXPLANATION
Unknown command ? ? ? <remainder of command string>	EDT3 encountered a command that it was not able to execute. Check the command.
String not found ? ? ? <remainder of command string>	The string was not found.
Buffer full ? ? ? <remainder of command string>	The capacity of the text buffer was exceeded--before a form feed was encountered. The text in the text buffer may be edited and output. This message is also typed when an A command is issued in spite of an already full text buffer.
Macro error ? ? ? <remainder of command string>	At the time the X command was issued, the macro buffer was empty, or the macro buffer contained the X command. Define a macro that does not include an X command.
No input file open No output file open	A P, Y or A command was issued and there was no corresponding file open.
Not allowed after EOF is reached on input file	An E, R, N, or O command was issued after an EOF was found on the input file.

NOTE: If a macro command is the first command in the <remainder of command string> message, the error message is expanded to show the macro command quantifier (n), the unexecuted-remainder of the macro command string (in brackets), and the remainder of the command string. The quantifier given for the macro is the decimal number of macro execution attempts left at the point where the condition occurred to end macro command string execution.

EXAMPLE:

```
: XMCA$a$$          defines a macro.
: B5ØØX$$           executes the macro.

String not found    is an error message.

?  ?  ?
495[CA$a$$]$$       is the unexecuted portion of the command string.
```

8 0 8 0 F O C A L

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

SECTION PAGE

1.0 FOCAL - DISK FOCAL INTERPRETER

1.1 INTRODUCTION

FOCAL is the name of a computer language as well as the name of the program which translates and executes programs written in that language. The program, FOCAL, belongs to a class of language processors called "interpreters," and this means that FOCAL, while operating, has complete control of the machine, and thus can assist in storing, editing and running programs. Externally, FOCAL communicates with a user through an input/output device like a teletype. Internally it divides up memory into three sections containing the FOCAL program itself, the user's stored program and any variables the user may have created. A minimum memory size of 8K is necessary for FOCAL, and additional memory allows FOCAL to store larger programs and more variables. The additional machine requirements are described in the section called "Running FOCAL."

The user controls FOCAL by typing a line of characters followed by a carriage return. The input line can be a command to FOCAL which it must execute immediately, or it could be a program line to be stored for later execution. These types of input lines can be intermixed as there is no interference between them. An input line, which is to be stored as a program statement must begin with a number identifying its location within the program. A line without a number is not stored but executed immediately. FOCAL determines the order of execution for lines in the stored program from their line numbers, not the order they were typed, which makes adding or replacing lines during a debugging session very easy. The following sequences, listed as they were input, will both result in the same sequence of calculations.

```
:1.0  SET A=1
:2.0  SET B=A*3.2
:2.21 SET C=A+B
:4.0  SET X=A+C/2

:2.21 SET C=A+B
:1.0  SET A=1
:4.0  SET X=A+C/2
:2.0  SET B=A*3.2
```

The digits to the left of the decimal point in a line number make up what is called the GROUP NUMBER, and can be used in some statements to identify a block of statements. The digits to the right of the decimal are known as the STEP NUMBER and these have no special significance; step numbers can be assigned values in the range 01 to 99; 0 and 00 are illegal. Groups of statements will later be referenced by certain commands by a

group number followed by a zero step number. The reader should note that there is no single line in the program with a zero step number; this number refers to the entire block having that group number.

Most computer systems draw a distinction between commands and statements. Commands are input lines given to a program, usually called an "operating system," which controls the entire machine. Statements, on the other hand, are lines written in a strictly defined language, and these are interpreted by a program subordinate to the operating system; this program could be a "compiler," an "assembler" or an "interpreter." FOCAL is unusual in that it has the functions of both an operating system and an interpreter, and this gives it an enormous flexibility. It can handle both statements from the language FOCAL and commands of a supervisory nature. This distinction between operating systems and interpreters, statements and commands is further weakened because FOCAL allows statements from the language to be executed immediately much like commands in other machines. This manual will put little emphasis on the differences between commands and statements; in fact, the terms will be used almost interchangeably.

At any given instant FOCAL will be in one of three states or operating modes READY, EXECUTION or PROGRAM INPUT. FOCAL enters the READY mode after it finishes the last command given to it, and as it enters this state it issues a colon (:) to the user's terminal. It remains in this mode until the user has typed an input line followed by a carriage return. In the short sequences given on the first page of the introduction, the colons were supplied by FOCAL and the user typed the remainder of the line shown. The carriage return, needed at the end of every input line, forces FOCAL to leave the ready mode and enter the EXECUTION mode to perform the actions specified by the input line. This may mean that it only has to store the line as part of the program under construction. The input line could also make FOCAL execute the input line as though it were a part of a program; it could ask FOCAL to perform some supervisory function, or it could have FOCAL execute the stored program.

The last mode, PROGRAM INPUT, occurs when FOCAL encounters and executes a special instruction, ASK, which will be described in full later. At that time FOCAL issues a question mark to the terminal and waits, as in the READY mode, until the user enters a number followed by a carriage return. The difference between this mode and READY is that the user is "talking" to his program through FOCAL. The execution mode is automatically re-entered after the input is completed.

1.1.1 GO And QUIT Commands

The stored program begins execution when FOCAL is given the GO command. This execution begins at the lowest line number in the stored program and will proceed from there to higher line numbers as the program logic allows. The running program can be stopped by the user, by FOCAL or by the program itself; regardless of the reason, control will pass to the "READY" mode of FOCAL at the conclusion of the run.

The program can stop itself with the QUIT statement, which can be stored anyplace, any number of times, throughout the stored program. The instant this statement is executed the program stops and returns control to FOCAL which will issue a ready prompt to signal the end of the program. The program is not altered in any way by running so that it can be immediately re-run if desired. Examples of the GO and QUIT statements appear all throughout this manual.

In the event the program begins acting in some undesirable way, the user can halt the program by typing "MODE" (this can also be done by hitting the "CNTL" key and the "@" simultaneously). FOCAL should respond with the READY prompt; if not,the program has managed to annihilate part of the program FOCAL, and FOCAL will have to be reloaded.

FOCAL will stop a program, if an error is discovered, or the program executes its highest numbered line and does not jump back into the rest of the program. In this latter case, FOCAL tries to find a higher numbered line, and failing it returns to the ready mode with a colon. Errors terminate a job with question marks and an error code which can be deciphered with the table given in a later section, except that errors which occur in attempting to execute an operating system command (like deleting a file) are explained in English by the operating system's error-handling routines.

1.2 CONVENTIONS

1.2.1 NUMBERS

All numbers in FOCAL are internally treated as floating point numbers occupying four memory bytes apiece. The largest number which can be represented by FOCAL is 3.6 times 10 to the 38th power, and the smallest non-zero positive number is 2.7 times 10 to the minus 39th power. This same range applies to negative numbers as well.

The accuracy for a number anywhere in this range is limited to approximately 7 decimal places making 850.0000 equivalent to 850.00003. Any number can be given to FOCAL as an integer (without a decimal point) as a floating point number (a number containing a decimal), or as a number in scientific notation. Numbers in the scientific notation format consist of a mantissa and an exponent; the mantissa is written in decimal form followed by an E, followed by the exponent value. In the scientifically formatated number, -7.2E-11, the number -7.2 is the mantissa and -11 is the exponent. The value of this number is -7.2 times 10 to the -11th power or -.000000000072. Any form of number input may be signed (+ or -) or unsigned.

In FOCAL the following numbers are equivalent:

```
    700.3240
    700.32403
      7.003240E2
      7.00324E+2.0
      7.00324E01.0
     70.0324E01.0
700324.0E-3.0
       .0700324E4
```

All numbers printed by FOCAL can contain up to 7 decimal digits (excluding the sign).

1.2.2 VARIABLES

A variable is a uniquely named storage location having an associated arithmetic value. Its name consists of a sequence of letters and/or numbers, the first character being any letter other than F. (Names beginning with F are assumed to be function names.)

FOCAL variable names are unique in their first two characters only. Thus, the variable names SA, SAM and SAMMY refer to the same storage location.

LEGAL NAMES	FOCAL RECOGNIZES	
KNOCK	KN	
PROFIT	PR	
COST1	CO	(Thus COST1 and COST2
COST2	CO	are the same!)

ILLEGAL NAMES	REASON
3ARM	First character must be be alphabetic.
FOOT	First character must not be F.

To facilitate storage of large amounts of information, FOCAL allows variables to be subscripted. This feature will be described later.

1.2.3 EVALUATING EXPRESSIONS

FOCAL, which is a contraction of FOrmula CALculator, allows the user to construct arithmetic expressions or formulas using the following symbols:

^	EXPONENTIATION
*	MULTIPLICATION
/	DIVISION
+	ADDITION
-	SUBTRACTION

8080 FOCAL is similiar to DEC FOCAL in that the evaluation of arithmetic expressions proceeds according to standard operator priority. This priority follows the table above with the "^" operation having the highest priority.

Occasionally it becomes necessary, due to the complexity of an expression, to NEST parts of the equation in parentheses. Just as a single pair of parentheses reorder the sequence of calculations in the above example, the "sub-expression" within parentheses can be reordered by separating its parts with parentheses. For instance,

```
*SET    X=CA*A+B*C+B/2.6
*SET    X=(CA*A+B)*(C+B/2.6)
*SET    X=(CA*(A+B))*((C+B)/2.6)
```

all contain legal expressions. Each will, however, use a different sequence of multiplications, additions, etc., which will produce a different value for the variable "X". The internal sequence of steps for evaluating the last of the above examples would be:

1. ADD A TO B. PUT SUM IN TEMPORARY LOCATION "1" (TEMP)
2. ADD C AND B. SUM TO TEMP "2"
3. DIVIDE VALUE IN TEMP2 by 2.6 AND STORE QUOTIENT IN TEMP2
4. MULTIPLY CA BY TEMP1. PRODUCT STORED IN TEMP 1
5. MULTIPLY TEMP1 BY TEMP2. PRODUCT STORED IN X

The level of a nest, or "Level number," is equal to the number of left parentheses minus the number of right parentheses found to the left of the term in question. FOCAL, as shown, evaluates the terms with the highest level number first, and works down from there. Any level of nesting is allowed, as long as the statement occupies only one line.

1.2.4 MATH FUNCTIONS

FOCAL provides eleven standard math functions along with a user-defined assembly language function. A function is a routine internal to FOCAL which performs an arithmetic calculation on a value called an "argument," which is given to it. This argument must be enclosed in parentheses immediately following the functions's name. FOCAL's generality permits this argument to be a constant, variable or expression. The following is a list of the function names recognized by FOCAL (throughout this list, the character "X" represents the argument):

FUNCTION	USE
FABS(X)	ABSOLUTE VALUE
FSGN(X))	"SIGN" OF X. VALUE RETURNED IS -1 WHEN X IS NEGATIVE, 0 WHEN X=0, AND +1 WHEN X>0
FITR(X)	INTEGER PART OF X
FRAN(X)	RANDOM NUMBER BETWEEN .5 and 1.0 WITH RANDOM SIGN
FATN(X)	ARC TANGENT
FEXP(X)	EXPONENTIAL -- E^X
FLOG(X)	NATURAL LOG
FSIN(X)	SIN OF X
FCOS(X)	COSINE OF X
FSQT(X)	SQUARE ROOT OF ABSOLUTE VALUE OF X
FHYS(X)	HYPERBOLIC SIN
FUSR(X)	USER DEFINED. WHEN UNDEFINED, THIS RETURNS THE VALUE OF THE ARGUMENT UNCHANGED

In a statement of the form SET Y=FSIN(PHI), the variable Y is given the computed value of the sine of the angle, "PHI". All of FOCAL's trigonometric functions assume that their arguments are given in radians (FATN returns a radian value from minus pi to +pi). To convert degrees to radians simply divide by 57.29579.

1.2.5 COMMAND LINE INTERPRETATION

FOCAL allows and even encourages the programmer to put more than one statement on an input line. The additional commands (statements) need to be separated by semicolons, and in one case, the "FOR" statement, the entire line must be ended with a semicolon. This multiple statement line feature can be used in both program store mode and immediate mode.

FOCAL also allows, for efficiency's sake, abbreviated commands; thus, for example, "SET," "GO" and "QUIT" could all be written as "S," "G" and "Q." Internally all commands are identified by their first letter only, so that "SHAKE", "GASP" and "QUAKE" could be used for "SET," "GO" and "QUIT." All of these statements and their functions will be described shortly.

To simplify the command recognition process, the FOCAL language has been constructed with a mild emphasis on blanks. In the commands to be discussed in the upcoming sections, all must begin with an easily recognized word (or abbreviation) like "SET," "GO," "ASK" and "Q." This word, called a KEYWORD, must be followed by at least one blank. This is a common source of error for people new to the language. Line numbers in stored programs also require a trailing blank for the same reason.

Most of the FOCAL language statements will expect a sequence of characters following the keyword and its blank. The form and content of this part of the statement will depend on the command in question. Those commands not requiring more than the keyword will ignore anything between the keyword and the next semicolon or carriage return. Telling FOCAL to ":QUIT YER COMPLAININ'" will only cause it to QUIT. Telling it to ":GO TO HEAVEN" (?) will cause an error, because "TO HEAVEN" is not the line number expected by the "G" or "GOTO" command. For the same reasons, "GO TO 5.1" will force an error because "TO 5.1" is not a legal line number. This command will be described later.

1.3 THE SET COMMAND

The most fundamental command in FOCAL is the SET command. In its general form it looks like:

```
:<line number> SET <variable>  =  <expression>
```

This command can also be used in the immediate mode by omitting the line number.

The variable names (X, Y, Z and D in the example below) are defined by FOCAL and are given values (25.1, -6.88, etc.). A memory location is associated with the variable name. If the specified variable name has not yet been encountered by FOCAL, a new memory location is set aside and is associated with the name.

```
:10.4  SET Z=12.01
:10.5  SET X=25.1
:10.6  SET Y=-6.88
:10.7  SET D=FSQT((X*X)+(Y*Y)+(Z*Z))
:GO
:
```

After the program has been run and FOCAL returns to the ready mode, the memory location for the variable D (from above) contains the value of the expression FSQT((X*X)+(Y*Y)+(Z*Z)).

The execution of the SET command produces the same results whether the command was stored and executed, or executed in immediate mode. It is often very effective to use the command in both ways during a series of runs with a program. Before the RUN command is given, the controlling values for the problem can be set or defined allowing the program to be very general. The following illustrates a typical sequence of runs using this feature.

```
:1.1   SET A=FSQT(B/C+B^3)*D                INPUT THE
:1.2   SET . . . . . .                      PROGRAM
: .
: .
: .
(generalized program)
: .
: .
: .
:99.9   QUIT
:SET    B=10.1; SET C=12.77; SET D=60       SET PARAMETERS
:GO                                         RUN THE PROGRAM
(results)                                   EXAMINE RESULTS
:SET    B=10.6; SET C=11; SET D=100         MODIFY PARAMETERS
:GO                                         RERUN THE PROGRAM
(results)                                   EXAMINE
```

1.4 INPUT/OUTPUT COMMANDS

1.4.1 THE TYPE COMMAND

Every FOCAL program must contain at least one TYPE statement if it is to produce printed results. The TYPE statement prints values of variables, text strings and results of expressions. These can be combined using commas to separate the items into a list. The following example shows several TYPE statements and their resultant printout:

```
:SET A=1;SET B=2;SET AB=-6
:TYPE A,#
 1.000000
:TYPE A,B,AB,#
 1.000000 2.000000-6.000000
:TYPE "QUOTATION MARKS START AND END TEXT"
QUOTATION MARKS START AND END TEXT:TYPE A,#
 1.000000
:TYPE !,"NOTE HOW THE # AND ! WORK",#

NOTE HOW THE # AND ! WORK
:TYPE A, !,B,#
 1.000000

 2.000000
:TYPE %5.02,AB,"hi",12345.5456
  -6.00hi2345.55:
```

The examples are in immediate mode where their results were immediately visible. The only modification for program storage would be the addition of line numbers. It is important in the examples to watch how the special characters comma, #, ! and " are used.

The % begins a field description of the form %w.0d which describes how numbers should be printed. The w is the width to be used (maximum no. of digits), and the d is the number of these digits which are to appear after the decimal point. The 0 is required. In the example above, note that truncation occurs if the number exceeds the field width. (Six digits were retained because there is room for a sign.) The field description remains in effect for all TYPE statements until another field description is seen.

The most readable output is usually made by combining text with printed values. The program can thus identify a value as well as print its value. The following shows this feature used in program store mode.

```
:1.1   SET G=32;SET T=5;SET D=.5*G*(T^2)
:2.1   TYPE "FOR ACCELERATION",G,"AND TIME",T,"SECONDS",#
:2.51  TYPE "AN OBJECT FALLS",D,"FEET.",#
:GO
FOR ACCELERATION 32.00000 AND TIME 5.000000 SECONDS
AN OBJECT FALLS 400.0000 FEET.
:
```

Labeling results is a very good programming practice usually ignored by beginning programmers. It does require more time and effort, but this is more than offset by the amount of clarity added to the code and its output. For programs which could be stored for any amount of time or for lengthy programs, any kind of documentation is very helpful and this labeling with TYPE statements is a very good form of documentation.

If a $ appears in the list of things to print (not in quotes) FOCAL will print out all the variables in use and their corresponding values. The $ terminates the print list, that is, anything following it won't be printed.

A very powerful use of the TYPE statement comes from its ability to print the result of whole expressions. This means your computer can be used as a super calculator which understands variables. This capability is generally used in the immediate mode as shown below. The variables used are the same as those stored for use by the program.

```
:TYPE 5^2*16.,#
400.0000
:TYPE FSQT(2*D/32),#
5.000000
:TYPE "TIME TO FALL",D,"FEET IS",FSQT(D/16),"SECONDS.",#
TIME TO FALL 400.0000 FEET IS 5.000000 SECONDS.
:SET D=144
:TYPE"TIME TO FALL",D,"FEET IS",FSQT(D/16),"SECONDS.",#
TIME TO FALL 144.0000 FEET IS 3.000000 SECONDS.
```

1.4.2 THE ASK COMMAND

Input to a FOCAL program is handled by the ASK command. It is used in stored programs to define or redefine the values of program variables. The command can contain a text string and a list of variables. No expressions may appear in the ASK command although they are allowable responses to the command. When executing an ASK, FOCAL issues a question mark to request a value. The value, or expression, for which FOCAL can compute a value, must be followed by a carriage return. FOCAL then issues a question mark for the next variable to be defined, and so on. Text is printed as encountered in the command. In use this looks like:

```
:70.6   ASK "DEFINE STARSHIPS X,Y,Z COORDINATES",X,Y,Z,#
:70.61  ASK "DEFINE X,Y,Z FOR KLINGON SHIP",XK,YK,ZK,#
:70.65  SET XD=XK-X; SET YD=YK-Y; SET ZD=ZK-Z
:70.66  SET DIST=FSQT((XD*XD)+(YD*YD)+(ZD*ZD))
:70.69  TYPE "DISTANCE TO ENEMY IS",DIST,"LIGHT YEARS",#
:GO
DEFINE STARSHIPS  X,Y,Z COORDINATES?4?5?6
DEFINE X,Y,Z FOR KLINGON SHIP?9?4?1
DISTANCE TO ENEMY SHIP IS 7.141429 LIGHT YEARS
```

The ASK command also has provisions to allow a defined value to remain unchanged. The user can type the ESCAPE key in response to the question mark, and the corresponding variable will be unchanged. Typing a carriage return will result in the variable being set to 0. Typing an expression (which may even contain variable and functions) will cause FOCAL to evaluate the expression and assign the resulting value to the variable in question. ASK will continue to issue question marks for the remaining variable in its list. Although the user follows each entry with a carriage return, a line is advanced during an ASK command only when a colon or exclamation point appears in the statement.

1.5 BRANCH COMMANDS

The computer's ability to alter the sequence of commands it will execute is known as branching. This very powerful ability is represented in FOCAL by three commands: GOTO, IF and JUMP. These can alter the program flow rather than executing statements in their numeric order. The computer can send control to a program line number specified in the command. These commands differ in that GOTO always transfers control to the single statement number given to it, while JUMP and IF transfer to one of a number of possible statements based on a test.

1.5.1 The GOTO Command

In the following example, the GOTO statement sends control back to a statement that counts the number of times it has been executed. Readers new to programming are strongly advised to follow the example and the results closely.

```
:1.1  SET N=0
:1.3  SET N=N+1
:1.6  TYPE "LOOP NUMBER =",N,#
:1.7  GOTO 1.3
:GO
LOOP NUMBER = 1.000000
LOOP NUMBER = 2.000000
.
.
```

Unfortunately, this program never ends, and the programmer will never see the READY colon from FOCAL. Program segments which repeat are called "loops." The program shown above is an example of an "infinite loop". To escape such a loop, type MODE and the program will stop.

GO (GOTO) starts a program at the line number specified or at the lowest line number in the program, if no line number follows. The GOTO statement can also be used in the immediate mode to transfer control to the program.

In the next example, the instruction GOTO 1.8 passes control to statement 1.8, and starts the loop with the loop counter already equalling 12.

```
:1.1        SET N=0
:1.3        SET N=N+1
:1.8        TYPE "LOOP NUMBER =",N,#
:1.9        GOTO 1.3
:SET N=12; GOTO 1.8
LOOP NUMBER = 12.00000
LOOP NUMBER = 13.00000
LOOP NUMBER = 14.00000
LOOP NUMBER = 15.00000
```

1.5.2 The IF Statement

In the above example, the program will again cycle indefinitely, since it has no condition for ending itself. For this reason, FOCAL includes the IF command which transfers control CONDITIONALLY. The basic form of an IF statement is;

:<line number> IF (<expression>) L1,L2,L3

where L1,L2, and L3 represent statement numbers, and the expression, always enclosed in parentheses, stands for a single variable or arithmetic formula containing variables.

When FOCAL encounters an IF statement, and the value in parentheses is negative, the control is transferred to the first statement number in the list. If the value is zero, control goes to the second, and if it's greater than zero, it transfers to the third. FOCAL recognizes abbreviated forms of the 'IF' statement containing one or two statement numbers rather than three. Should the IF statement only contain 2 statement numbers in its transfer list, control will be given to the statement following the IF statement when the value is greater than zero. Similarly, when an IF statement contains only one statement, a value greater than or equal to zero will have control transferred to the next sequential command. These different styles of IF statements are shown in the following examples.

```
:1.2  IF (A-B) 1.5,1.4,1.3
:1.3  TYPE "A IS GREATER THAN B"; QUIT
:1.4  TYPE "A IS EQUAL TO B"; QUIT
:1.5  TYPE "A IS LESS THAN B"; QUIT
.
.
.
:22.1   IF (MONEY) 22.28, 22.28; TYPE "YOU STILL HAVE
        FUNDS",#
:22.3   DO 24.0; GOTO 15.4
.
.
.
:40.6   IF (II) 40.8; DO 70.0; GOTO 40.6
:40.8   TYPE "II IS FINALLY NEGATIVE.  GOODBYE",#
:40.9   QUIT
```

Note in the above that a space always separates the IF and the open parenthesis mark. These examples are shown only to exercise the various aspects of the IF statement. They are not meant to be working parts of a single program.

1.5.3 The JUMP Statement

The last of the branch instructions is the JUMP command. This statement is frequently used when a program needs to transfer to one of more than three locations. The general form of this multibranch instruction is

*<line number> JUMP (<expression>), L1,L2,L3,L4, . . .,Ln

L1 through Ln are the statement numbers much like those in the IF command definition, but there may be as many numbers given as can fit in the command line. EXPRESSION, as before, is any single variable or arithmetic combination of variables. If the value of the expression equals 0, control transfers to the first statement number given. When the value equals 1, the second statement number is chosen, and so on. Should the value contain a fractional part, like 2.37 or 2.98, only the integer part is considered. The values 2.37 and 2.98 would both transfer control to the third statement listed. The following shows this command being used to select a part of a program given some input from the user.

```
:1.2  ASK "1) RIGHT 2) LEFT 3) UP 4) DOWN 5) NO CHANGE",N
:1.4  JUMP (N-1), 10.1,15.34,12.3,65.98,2.02
.
.
```

If the computed value of the bracketed variable or expression is less than zero, control goes to the first statement listed and if it's too large for the list, FOCAL sends control to the last statement listed. If the user had responded 5 or larger to the ASK statement above, control would have gone to the statement numbered 2.02.

1.6 SUBROUTINES

1.6.1 The DO And RETURN Commands

A subroutine, sometimes called a "routine," is a special sequence of statements with the same group numbers. A group number, as mentioned, is the integer part of a statement's line number and the fractional part is the step number. A subroutine, for instance, could be the sequence of statements between line numbers 52.01 and 52.99. This sequence is "special" because any part of the program can send control to this block of statements and receive it again when the block, the subroutine, has finished. The "sending of control" is known as a subroutine "call" and the process used for a subroutine to return this control to the code which called it is a "return" from subroutine. In FOCAL the subroutine is thrown into execution by the DO statement, and the return from the routine by a RETURN statement.

The DO statement must specify a line number containing the group number for the subroutine to be called and a step number of zero; thus "DO 52.0" is acceptable, whereas "DO 52" is not. The RETURN statement requires no arguments; it simply returns control to the statement following the DO statement which called it. Note, in the example below, that 1) the same subroutine can be called from many places, and 2) a subroutine may call another subroutine, which may call yet another, which. . .etc.

```
:10.1  DO 20.0; DO 18.0
:10.2  C IF J NOT RIGHT, CALL 20.0
       AGAIN
:10.3  IF (-J) 10.4; K=20*T; DO 20.0
       .
       .
:20.1  F I=1,10; DO 15.0;
:20.6  R
       .
       .
:18.1  DO 20.0; T K,#
:18.2  DO 16.0
:18.3  R ; C THESE EXAMPLES WERE NOT
       TAKEN FROM
:18.4  C A WORKING PROGRAM
```

The next example further exercises the flexibility of the subroutine calling structure. In this example, a subroutine calls ITSELF until a certain condition is satisfied, and then it begins a series of RETURNS while calculating a factorial for a number. A return to itself is made for each call it made to itself; the last return sends control back to the DO statement which originally called this factorial subroutine. This sort of subroutine calling is known as RECURSION. Computer theory buffs should note that the initial variable list is used for all levels of the calling sequence; FOCAL does not dynamically allocate new memory for copies of the variables.

```
:1.1  A N; C ASK FOR THE NUMBER TO USE
:1.2  S NF=1; DO 2.0; T N,"FACTORIAL IS",NF,#;Q

:2.1  C SEE IF N IS GREATER THAN 1.  IF SO SUBTRACT
:2.2  C ONE AND CALL THIS ROUTINE AGAIN UNTIL IT IS 1
:2.3  C THIS ROUTINE RETURNS AS MANY TIMES AS IT WAS CALLED
:2.4  C AND THIS CONTROLS THE FACTORIAL CALCULATION
:2.5  IF(N-1) 2.8,2.8 ; S N=N-1; DO 2.0
:2.6  C RETURNS ENTER HERE
:2.7  S N=N+1; S NF=NF*N
:2.8  R ; C RETURN FROM LAST CALL
```

If FOCAL runs out of step numbers for a subroutine, thus threatening to continue into the next group of line numbers, it issues the RETURN. This makes it perfectly valid to omit the return statement from a subroutine. This optimizes memory requirements at the expense of a program that becomes more difficult to read.

1.7 LOOPS

1.7.1 The FOR Command

Program loops can be constructed in FOCAL with the FOR command. This command executes the remaining statements on its SAME LINE a specified number of times. The number of loops depends upon the numbers given to the FOR command. In its full form, FOR uses 4 values; an index variable, a start value, increment and stop value for the index. These, in order, look like:

```
:12.1 FOR I=1,3,200 ; J=I/2 ; TYPE J;
```

In the above, the values 1,3, and 200 could have been variable names, and the second and third statements on that line could have been any legal statements in the language. They must be followed by a blank and the ENTIRE LINE must be followed by a semicolon. In the following example,

```
:12.4 FOR II=VN,NN,Q ; TYPE II, FSQT(II),#;
```

the variable II is initially given the value VN. On successive loops its value increases by the amount NN, and when this value exceeds Q control is passed to the next LINE NUMBER. If only 2 values follow the equal sign, it is assumed that the increment has been omitted, leaving only the start and end values for the next II. In this case, the increment is set to 1.0.

Since the FOR command executes only the statements on its same line, it is convenient to use it in conjunction with the DO command. The polynomial graphing given on this page shows this in use.

```
:20.2   S LX=40; S LY=70; S YN=0; S YX=100
:20.3   S XN=0; S XX=100
:20.5   S SX=(XX-XN)/LX ; S SY=(YX-YN)/LY
:20.6   A "DEFINE A,B,C FOR AX^2+BX+C" ,A,B,C
:20.7   T #,"GRAPH Y=AX^2+BX+C, X IS DOWN,"
:20.71  T "Y ACROSS",#
:20.9   F X=XN,SX,XX; DO 60.0;
:20.95  Q

:60.1   S Y=X^2*A+(B*X)+C
:60.3   I (Y-YX) 60.5; S Y=YX; G 60.8
:60.5   I (YN-Y) 60.8; S Y=YN; G 60.8
:60.8   T "I"
:60.83  F J=YN,SY,Y; T "*";
:60.9   T # ;C RETURN
:GO
DEFINE A,B,C FOR AX^2+BX+C?1/80?-1?40
(PRINTS OUT GRAPH HERE)
```

The reader should study the example shown here, and compare this to the fully commented version. It should also be very instructive to run this program as shown, then modify both the program and the data as desired.

The term "loop" refers to any sequence of statements which can be executed repeatedly; the "FOR" statement is only one way of forming a loop. A common way of setting up a loop uses "SET," "IF" and "GOTO" statements in such a way that a counter (or "loop index"), an increment value and a limit are manipulated by the program directly. The following shows the code necessary for such a loop:

```
:23.25  C INDEX IN IS SET 1 INCREMENT LOWER THAN 1ST VALUE
:23.3   SET IN=0
:24.1   C INCREMENT LOOP INDEX. 24.4 IS LOOP START
:24.4   SET IN=IN+IC
:25.1   C TRANSFER OUT OF LOOP WHEN INDEX EXCEEDS LIMIT
:25.2   IF (IN-LIMIT) 27.6
:25.7   C HERE STARTS THE CODE FOR THE BODY OF THE LOOP.
        .
        .
        .
        .
:27.5   GOTO 24.4 ; C FORCE NEXT LOOP
:27.6   C THIS STATEMENT IS OUT OF THE LOOP
:27.7   C REST OF PROGRAM CONTINUES FROM HERE
        .
        .
```

Another example is given in the next section on subscripted variables.

1.7.2 Subscripted Variables

The variables in a program generally represent the physical entities of the problem under study. The programmer can think of his variable, "T," as containing the current time in seconds or another variable, "SP" as representing a vehicle's speed. This association between variables and their physical meaning is fundamental to any type of computer programming. Quite often, however, several values must be simultaneously associated with a single concept, and thus, the programmer would like to have a single variable name represent these many values. A chess board is a good example of this, since the programmer would like 64 values held for the single board. While it would be possible to assign each of the squares a separate name, FOCAL's subscripted variable feature allows all the squares to be referenced with the same name. A variable which has many values is called an ARRAY, and its separate values may be selected by means of a SUBSCRIPT or INDEX. A subscript is an integer tag identifying a particular value within an array. It is enclosed in parentheses immediately following the array name. "BD(1)", for example,

might be the first square of the board, while "BD(64)" might be the last. Little advantage would be realized were it not for the fact that subscripts themselves can be variable names or even expressions. In other words, any expression can be put into the parentheses following the array name; FOCAL merely calculates the value of the expression, drops any fractional part, and uses the resultant integer to select a single value from the array.

As an illustration, the following sequence of statements, written as a subroutine, counts the number of pieces a chess bishop can threaten from his square. Some initial definitions at the beginning of the program are shown as is the routine itself; the actual call(s) from the main body of the program have been omitted.

```
:1.11   C 8 POSSIBLE DIRECTIONS OF MOVEMENT STORED IN DX,DY
        ARRAYS
:1.1    C DEFINE ARRAYS TO BE USED BY LATER SUBROUTINES
:1.12   S DX(1)=1 ; S DX(2)=1 ; S DX(3)=-1 ; S DX(4)=-1
:1.12   S DX(5)=1 ; S DX(6)=-1 ; S DX(7)=0 ; S DX (8)=0
:1.13   S DY(1)=1 ; S DY(2)=-1 ; S DY(3)=1 ; S DY(4)=-1
:1.15   S DY(5)=0 ; S DY(6)=0 ; S DY(7)=1 ; S DY(8)=-1
:1.16   C FIRST 4 USED BY BISHOP. LAST BY ROOKS. ALL BY QUEEN/
        KING
:       .
:       .
:       .
:       .
:       .
:       .
:       .
:70.02  C THIS SUBROUTINE COUNTS THE NUMBER OF OPPOSING PLAYERS
:70.04  C THREATENED BY A KNIGHT AT ROW "NR", COLUMN "NC"
:70.06  C ASSUMES THAT OPPOSING PIECES ARE CODED AS NUMBERS WITH
:70.08  C OPPOSITE SIGNS AND THAT EMPTY SQUARES CONTAIN ZEROS.
:70.10  C "BD" HAS THE ENTIRE BOARD OF 64 SQUARES. "ZAP" COUNTS
:70.12  C THREATS FOR THE CALLING ROUTINE AND "LP" WILL
:70.14  C BE THE LOOP DIRECTION COUNTER INTERNAL TO THIS ROUTINE
:70.16  C START BY ZEROING COUNT AND STARTING DIRECTION INDEX
:70.17  S ZAP=0 ; S LP=0
:70.18  C PUT BISHIP'S VALUE INTO BT TO COMPARE LATER. FROM NR,
        NC
:70.20  S BT=BD(NC-1*8+NR)
:70.22  C START LOOP - RETURN WHEN LP PAST 4
:70.24  SL P=LP+1 ; I (LP-5) 70.26 ; RETURN
:70.26  C SET INITIAL POSITION OF MOVING SQUARE
:70.28  S NX=NR; S NY=NC
:70.30  C LOOP THROUGH NEXT SQUARES ON CHOSEN DIRECTION
:70.32  S NX=NX+DX(LP) ; S NY+DY(LP)
:70.34  C SEE IF YOU'RE STILL ON THE BOARD.  1 < or = to NX,
        NY < or = to 8
:70.36  I (NX) 70.24, 70.24 ; I (9-NX) 70.24,70.24
:70.38  I (NY) 70.24, 70.24 ; I (9-NY) 70.24,70.24
```

```
:70.40  C CALCULATE POSITION (INDEX) IN BOARD FOR THIS SQUARE
        (NX,NY)
:70.42  S SQ=BD(NY-1*8+NX)
:70.43  C MULTIPLY BY BISHOP'S VALUE TO CHECK SIGNS
:70.44  S PR=SQ*BT ; I (PR) 70.46,70.32,70.24
:70.45  C FOUND OPPONENT - COUNT IT AS THREATENED. LOOP AGAIN
:70.46  S ZAP=ZAP+1 ; DO 71.0
:70.50  G 70.24; C END OF SUBROUTINE

:71.1   C PRINT OUT THREATS - MONITOR PROGRAM PROGRESS
:71.2   T "PIECE AT ROW",NX,"COLUMN", NY," IS THREATENED BY",#
:71.25  T "PIECE AT", NR,NC,#
:71.3   R
```

FOCAL allows subscripts to have any values from -2047 TO +2047

1.7.3 The COMMENT Statement

FOCAL allows comments to be inserted into a program with the C command. This command requires a line number as any other command in a stored program, but when FOCAL encounters this statement, it simply skips to the next command. This statement begins with a line number, the letter C, and at least one blank following the C. The rest of the line, up to the semicolon, is ignored by FOCAL. Since these statements have line numbers, branches can be made to them. In this case, comments can be thought of as being "continue" statements (as in FORTRAN).

```
:19.1   C -- POLYNOMIAL GRAPHING PROGRAM --
:19.2   C -- DOCUMENTED VERSION --

:20.1   C PREPARE SCALING VALUES THAT RELATE THE SIZE
:20.12  C OF THE PHYSICAL GRAPH TO THE FUNCTION VALUES
:20.14  C TO BE PLOTTED.
:20.2   S LX=40 ; S LY=70; S YN=0; S YX=100
:20.22  C XN,XX ARE THE MINIMUM AND MAXIMUM X VALUES
:20.24  C YN,YX ARE THE MINIMUM AND MAXIMUM Y VALUES
:20.3   S XN=0; S XXS=100
:20.42  C COMPUTE LENGTH BETWEEN SPOTS ON PLOT BY
:20.44  C COMPARING BOUNDS TO LENGTH AND WIDTH
:20.5   S SX=(XX-XN)/LX ; S SY=(YX-YN)/LY
:20.52  C INPUT PARAMETERS FOR THE POLYNOMIAL
:20.6   A "DEFINE A,B,C FOR AX^2+BX+C ",A,B,C
:20.7   T #, "GRAPH Y=AX^2+BX+C X DOWN, Y ACROSS",#
:20.8   C EACH LOOP OF 20.9 DOES 1 LINE OF PLOT
:20.9   F X=XN,SX,XX; DO 60.0;
:20.95  Q

:60.05  C THIS ROUTINE COMPUTES POLYNOMIAL FOR WHATEVER
:60.07  C VALUE OF X IS PASSED.  IT'S THEN PLOTTED
:60.09  C (IF POSSIBLE) WITHIN THE DEFINED BOUNDS.
```

```
:60.1    S Y=X^2*A+(B*X)+C
:60.24   C IF Y VALUE TOO LARGE OR TOO SMALL, PLOT ON EDGES
:60.3    I (Y-YX) 60.5; S Y=YX; G 60.8
:60.5    I (YN-Y) 60.8; S Y=N; G 60.8
:60.8    T "I"
:60.81   C PRINT ASTERISKS UNTIL INDEX AS LARGE AS Y
:60.83   F J=YN,SY,Y; T "*";
:60.88   C FINISH THIS LINE WITH CR AND RETURN FOR NEW X
```

1.8 SUPERVISORY FUNCTIONS

1.8.1 The WRITE Command

For editing purposes, FOCAL provides the ability to print all or parts of the program text with the WRITE command. It can be used to copy the entire program text to paper tape for storage or can be used to print single lines or subroutines. WRITE 2.2 will print just the line which is numbered 2.2. WRITE 2.0 will print just all lines between 2.01 and 2.99, and the command WRITE ALL will print the entire program ordered by increasing line number.

1.8.2 The ERASE Command

The ERASE command is used to delete lines or groups of lines from a program. To erase a single line from the text, the user only has to type ERASE followed by the line number as in ERASE 22.34. To erase an entire group of lines such as a subroutine, the user can type ERASE followed by the group number. To delete a subroutine with the group number 95, the user should type ERASE 95.0.

ERASE can also be used to clear an entire program, its variables and their values. This is only done when the user wants to write a new program. The ERASE ALL releases all the memory assigned to the last program so that it can be used by the new one. It is naturally a good habit to save any lengthy programs on paper tape before erasing them.

1.8.3 The MODIFY Command

The MODIFY command is used in immediate mode to edit portions of lines in a FOCAL program. It accepts a line number designating the statement to be edited; this line number must be followed by a carriage return. The actual editing is performed on the line following the MODIFY command. The user must guide the MODIFY editor with certain non-printing commands. After the carriage return, MODIFY waits for the user to type a single character from the keyboard; this character will be used as the "search character." MODIFY will print the line in question up to and including this search character, or the user can direct modify to perform one of several other tasks described below. Each of these tasks is selected with a special character which is typed after the search character. If MODIFY does not recognize this character as being a member of its special list, it assumes that a text insertion is being made.

1. "CNTRL/G". This does nothing to the text already defined, but prepares the MODIFY function for a new search character. This new search character must follow immediately; neither will be printed. Any searches through the rest of the current line will use this new character.

2. "CNTRL/L" or "FORM FEED". This command restarts the search procedure in MODIFY which will begin typing the rest of the line until the current search character is found. As in the first search sequence undertaken by MODIFY, the search character will be typed, and MODIFY will then wait for more commands.

3. "DELETE". This deletes the last character in the line. Successive DELETE's delete characters in order from right to left towards the line number.

4. "CNTRL-X". This deletes everything in the line up to and including the last character printed. The rest of the line, as yet unprinted, remains intact. It will be shifted over, however, so that it follows the line number.

5. "RETURN". This deletes anything to the right of the last character last printed.

6. "LINE FEED". This instruction tells MODIFY to save the line as presently defined.

1.8.4 LIBRARY DISK COMMANDS

Processor Technology Disk Focal has an interface with PTDOS to save and recall programs and data from disk files. This interface is through a series of LIBRARY statements:

LIB SAVE <file name>	Save current program
LIB LOAD <file name>	Get program from disk
LIB DELETE <file name>	Kill file
LIB OPEN <#>,<file name>	Open a data file
LIB TYPE <#>,<var list>	"Type" data to file
LIB ASK <#>,<var list>	"Ask" for data from file
LIB REWIND <#>	Rewind a data file
LIB POINT <#>	Point to end of file (EOF)
LIB ENDFILE <#>	Make current position EOF
LIB CLOSE <#>	Close data file
LIB FILES [/u]	List all FOCAL files
LIB QUIT	Return to PTDOS

<file name>	is any valid PTDOS file name
<#>	is a digit between 0-9
<var list>	Valid FOCAL variable list which may contain constants, variables and expressions for TYPE, just variables for ASK. Must not contain quoted text, "$","#","?" or "!".

All LIBRARY commands may appear in a program or may be used in the immediate mode. Except for L TYPE and L ASK, none of the library commands should precede other commands on the same line, since the rest of the line will be ignored.

The L LOAD command may be used within a program to chain to another program. This will be discussed more fully under LIB LOAD below.

Naturally, all library commands can be abbreviated. For example LIB ENDFILE 5 can be written as L E 5.

LIB SAVE <file name> will save an exact copy of the internal form of the current program in memory onto file <file name>. If <file name> doesn't exist, it will be created type 03. If it does exist, it must already be type 03 or an error will occur.

LIB LOAD <file name> will load in the data in the file (which must exist, type 03) over any program which is currently in memory. If the LIB LOAD command was issued from a program, the newly read-in program will be executed starting at its lowest line number. This allows chaining of FOCAL programs. To pass data from the first program to the second, it should be written onto a data file by the first program (using LIB TYPE--see below) and read back from the same file number by the second program. (using LIB ASK). It is not necessary to re-OPEN the data files during a chain as long as the same file numbers are to be used in both programs. When chaining, variable's values are lost, unless they are saved on a data file.

LIB DELETE <file name> will delete the named file provided:

(1) it exists
(2) it is either type 01 (FOCAL data file) or type 03 (FOCAL program).

LIB OPEN <#>,<file name> is used to set up a data file for subsequent LIB TYPE or LIB ASK statements. If the file doesn't exist, it is created type 01. If it does exist it must already be type 01, or an error will occur.

The opened file is assigned the number <#>, and this number must be used to refer to the file in any subsequent data file operations such as TYPE, REWIND, etc.

LIB TYPE <#>,<var list> is similar to the regular TYPE command except that its output goes to file <#> where the number <#> has been assigned to a file using the LIB OPEN command. In addition, only numerical data can be written to a FOCAL data file--no quoted text, #, !, $ or ?. The data is written to the file in binary form--4 bytes per number. This data is written starting at the current cursor position for the file. For example, the sequence

```
:LIB OPEN 5,POTTS
:LIB POINT 5
:LIB TYPE 5,A,B^10,9*1024
:LIB CLOSE 5
```

results in the values of the three expressions A, B^10 and 9*1024 being added to the end of the file POTTS. (POINT puts the file cursor at the end of the file.)

LIB ASK <#>,<file name> is used to read values from a file which LIB TYPE has written on. The number <#> must have previously been assigned to <file name> in an OPEN statement.

LIB REWIND <#> sets the cursor for file <#> (which has been assigned in an OPEN) to the beginning of that file. This is where the cursor is when the file is first opened. LIB REWIND can be useful when chaining between programs, since the second program must start at the beginning to read the data from a file.

LIB POINT <#> sets the cursor for file <#> (which has been assigned in an OPEN, naturally) to point to the end of that file. LIB POINT is used when one wants to add data to the end of a file without losing any old data which may be there already.

LIB ENDFILE <#> makes the current cursor position the end of file <#>. Any data after this position in the file is lost. For example, the sequence

```
:LIB REWIND 2
:LIB ENDFILE 2
:LIB CLOSE 2
```

removes all data from file 2 (which has of course been assigned to some file in an OPEN). ENDFILE should be used any time that new data is being written on an old file which may contain garbage from a previous use of the file.

LIB CLOSE <#> removes the association between the given number and the file to which it was assigned in a previous LIB OPEN, writes out the file's buffer if necessary, and frees the buffer space for later use. After executing a CLOSE, the specified number may be re-used in a LIB OPEN statement. Re-using a number without first closing it will result in the file it used to belong to remaining open, and there is no way to close it short of LIB QUIT.

LIB FILES [/u] is used to obtain a list of all FOCAL program and data files which are currently on disk unit u. If no unit is specified, then FOCAL files on the default unit are listed.

LIB QUIT leaves FOCAL and returns to the Command Interpreter Before leaving, QUIT closes all files.

1.8.5 The TRACE Feature

The TRACE feature is provided to help debug stored programs. It can be activated from almost anywhere in a program and can be deactivated as easily. While operating, trace types out each line it sees being executed by FOCAL and reports on any variable values that are changed during each line of the program. A question mark is the symbol used to both activate the trace and deactivate it. Any question mark encountered outside a comment statement and outside the text parts of the "ASK" and "TYPE" commands will change the trace mode. If the question mark is encountered while trace is active, the trace will be deactivated. If seen while trace is "OFF" the trace mode will be turned "ON".

1.9 RUNNING FOCAL

1.9.1 Hardware Requirements

In order to run FOCAL, an 8080 based computer must be equipped with at least 8K of resident random access memory. This memory should be addressed starting at zero and continue upward contigous with preceding blocks. FOCAL is set up to use 16K of memory allowing more than 4K for programs and data. More than 12K can be used by changing memory location 107 and 108 to the number of bytes of continuous RAM in the computer.

For example, if 20K is available the location should be as follows:

ADDRESS	0107	00H
	0108	50H

These values may be changed any time after loading FOCAL. The number of memory locations used by any stored program can be calculated from the rule+ SIZE = 8S+C+4L where S is the number of variables, C is the number of characters in the stored program text, and L is the number of lines in the program. FOCAL can be re-"imaged" with the new value, if desired.

1.9.2 Miscellaneous Notes

Should you accidentally leave FOCAL before saving your program, you can restart it by typing EXEC 100 to the Command Interpreter or SOLOS.

FOCAL does not understand lower case letters. All variables, commands, functions, etc. must be in UPPER CASE. Lower case is okay within quotes or file names.

Remember that binary arithmetic is not exact--it is only very close. Therefore, you may expect very small (but nevertheless disconcerting) errors such as

```
:T 23+31
 54.00001
```

which are caused by FOCAL's attempts at rounding off numbers which it cannot store exactly. You can make these errors much less frequent by not printing as many digits to the right of the decimal point. (See TYPE)

1.9.3 Errors

The following is a list of error codes issued by FOCAL. The error number represents the address in the FOCAL program where the error was detected. Errors which occur during LIBRARY commands are self-explanatory and are not listed here.

ERROR CODE	MEANING
?00.00	Restart (you hit MODE or CNTL-@).
?01.DD	Input buffer overflow.
?03.CB	Bad line number.
?03.A6	Bad line number.
?04.32	Bad char in expression.
?04.37	Bad char in expression.
?05.52	Irrecoverable memory overflow. FOCAL must be reloaded.
?05.5B	Recoverable memory overflow. Enter ERASE to clear variables.
?06.55	Bad line number.
?06.CB	No such group.
?07.0A	DO references missing line.
?07.2A	GOTO references missing line.
?07.79	Invalid command.
?07.F1	Missing (in JUMP.
?08.09	Missing (in IF.
?08.52	Left of = bad in FOR or SET.
?08.72	Extra) in FOR or SET.
?08.89	Bad expression in FOR.
?0A.9E	MODIFY references missing line.
?0B.FE	Missing operator before (.
?0C.52	Arithmetic overflow.
?0C.F0	Missing (in function reference.
?0D.08	Parentheses error.
?0D.31	Unbalanced parentheses.
?0D.46	ERASE F ???
?0D.4E	Bad argument to ERASE.
?11.96	Can't raise to a negative power.
?11.FE	Invalid library command.
?12.10	Missing comma in LIB OPEN.
?12.8A	Missing comma in LIB ASK or LIB TYPE.
?14.00	No / before unit in LIB FILES.
?14.19	Garbage after unit in LIB FILES.
?15.65	File name missing.
?15.84	Name too long.
?15.B5	Bad file number.
?15.BA	Bad file number.