DIGITAL -7-3-S

PDP-7
SYMBOLIC ASSEMBLER
PROGRAMMING MANUAL

DIGITAL EQUIPMENT CORPORATION • MAYNARD, MASSACHUSETTS

CONTENTS

CONTENTS (continued)

CHAPTER 1

INTRODUCTION

The use of an assembly program has become a standard practice in the programming of digital computers. This type of processor permits a programmer to code in a more convenient language than the 18-bit binary numbers which are utilized by a PDP-7. The advantages of this practice are widely recognized: Easily recognized mnemonic codes are used instead of numeric codes; instructions or data may be referred to by a symbolic name; decimal or alphabetic data may be expressed in a more convenient form than in a binary number; programs may be altered without extensive changes in the source language; and debugging is simplified.

The basic process performed by the Assembler is the substitution of numeric values for symbols, according to associations defined in the symbol table. In addition, the user may request that the Assembler itself assign values to the user's own symbols at assembly time. These symbols are normally used to name memory locations, which may then be referenced by name.

The value of the ability to use mnemonic names to represent machine instructions cannot be overestimated. For example, the name ADD reminds the user of the addition function, while the number 300000 does not. Consequently, the instructions are easier to remember when mnemonics are used. The same is true of location names. It is much easier to associate the name TOTAL with the location containing the accumulated total, than it is to remember that location 13764 contains the total.

Another advantage is that, since the assignment of absolute numbers to symbolic locations is done by the Assembler, the updating of a program by adding or removing instructions is easy.

In addition to translating statements directly into their binary equivalents, the Assembler will accept instructions for performing translations. These instructions may not look different from other instructions, but they do not generate binary codes. For this reason, they are referred to as pseudo-instructions. For example the pseudo-instruction DECIMAL tells the Assembler that all numbers following in the program are to be taken as decimal rather than octal. This instruction is important to the assembly process but has no binary equivalent in the object program.

Certain other features of assembly can be directed at assembly time by setting the console ACCUMULATOR switches, abbreviated ACS.

The PDP-7 Assembly system consists of the Assembler and the FF Loader (Digital 7-1-1). A source program tape prepared in the source language using ASCII or FIODEC code, produces an object tape punched in FF Binary code in one pass through the Assembler. The object tape is loaded by the FF Loader (included on the object tape), which completes the assembly functions that could not be performed in one pass and loads the resulting binary program representation into the computer ready for execution.

The Assembler requires a basic 4K memory configuration with a teleprinter, perforated tape reader and perforated tape punch. Use of the basic Assembler allows 2202 locations for additional symbol storage during assembly in a 4K machine or 12202 locations in an 8K machine. The extended version of the Assembler has the ability to process the list of extended symbols in Appendix 2, in addition to the basic symbols handled by the basic version. The extended Assembler allows 505 locations and 10,505 locations when used with 4K and 8K machines, respectively. If this storage capacity is exceeded, the user must segment his program and assemble it in sections. When used with extended memory (more than 8K), the Assembler occupies the same area of memory as in an 8K machine.

The source program may be prepared in either ASCII or FIODEC code. Although ASCII has been used throughout this manual in programming examples, equivalent FIODEC characters as shown in Appendix 4 are equally valid.

CHAPTER 2

ILLUSTRATION OF PDP-7 ASSEMBLER FEATURES

THE LOCATION COUNTER

In general, statements generate 18-bit binary words which are placed into consecutive memory locations. The location counter is a register used by the PDP-7 Assembler to keep track of the next memory location available. It is updated after processing each statement. A statement which is assembled into a single machine instruction would update the location counter by one; a statement which is assembled into six binary words would update it by six. The location counter may be explicitly set by an element or expression followed by a slash. The element or expression preceding a slash sets the location counter to the value of that element or expression. Subsequent instructions are assembled into subsequent locations.

Example:

```
100/    The next instruction is placed in location 100.
```

CODING ILLUSTRATIONS

To illustrate some of the features of the PDP-7 Assembler, a small routine has been chosen and coded in a number of different ways. The routine continually adds one to the contents of a location until the result is positive, then halts. The instructions used are represented as their octal codes (more compact than the binary actually used). The code is the sum of the operation code (200000 for the first instruction) and the address in memory of the quantity to be operated on. The number being incremented is in location 200. The notation C(A) means contents of A.

Example 1:

```
100/    200200    /C(200) INTO AC
101/    300201    /ADD C(201) TO AC
102/    040200    /STORE AC IN 200
103/    741100    /SKIP ON PLUS AC
104/    600100    /JUMP TO LOCATION 100
105/    740040    /HALT
200/    0         /WILL CONTAIN NUMBER TO BE INCREMENTED
201/    1         /CONSTANT 1
```

Since the location counter is automatically incremented, specifying sequential addresses could have been avoided after the first address in the progression. In addition, the names of the PDP-7 instructions could be used in place of the octal codes. The octal representation of these instructions is substituted by the Assembler whenever symbols appear in the program.

Example 2:

```
100/    LAC 200
        ADD 201
        DAC 200
        SPA
        JMP 100
        HLT
200/    0
        1
```

The same program could have been written using symbolic address tags. The comma after the symbol A indicates to the Assembler that the location in which it places the instruction LAC B is to be named A. Information associating the symbol A with the number of the actual location is placed in the symbol table. Consequently, when processing the instruction JMP A, the Assembler finds the values of the symbols JMP and A in the symbol table and uses these values to form the binary equivalent of the instruction JMP A.

Example 3:

```
100/
A,      LAC B
        ADD ONE
        DAC B
        SPA
        JMP A
        HLT
200/
B,      0
ONE,    1
```

Unless the user specifically wanted to use locations 200 and 201 for storage, he could let the Assembler assign the locations.

Example 4:

```
100/
A,      LAC B
        ADD ONE
        DAC B
        SPA
        JMP A
        HLT
B,      0
ONE,    1
```

The Assembler also handles literals for the user. The value of the expression contained in parentheses is called a constant and is automatically placed in an unused location by the Assembler. The address of that location is inserted into the instruction. In the example below, the address of the register containing 1 is substituted in place of (1).

The user may also request the Assembler to assign variable storage for him by placing a # within the first six characters of the variable. A symbol which includes this character is automatically assigned a register at the end of the program, and a 0 is placed in that register.

Example 5:

```
100/
A,      LAC #B
        ADD (1)
        DAC B
        SPA
        JMP A
        HLT
```

Even though the actual locations used may not be the same, the results of the program assembled from the above examples will be the same in all cases.

CHAPTER 3

THE SOURCE LANGUAGE

This section explains the features of the ASCII source language available to the user of the PDP-7 Assembler; for equivalent FIODEC characters, see Appendix 4.

THE CHARACTER SET

Letters

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z .

Use of the lower case letters is not permitted. The character period is treated as a letter, and for purposes of discussion, is considered a letter.

Digits

1 2 3 4 5 6 7 8 9 0

Punctuation Characters

NOTE: Since a number of characters are invisible, the following notation is used to represent them in the examples.

␣	space
→|	tab
)↓	carriage return, line feed

The following characters are used to specify operations to be performed upon symbols or numbers.

Character		Use
␣	space	combine symbols or numbers
+	plus	combine symbols or numbers
-	minus	combine symbols or numbers
&	logical AND	combine symbols or numbers

Character		Use
!	inclusive OR	combine symbols or numbers
↵	carriage return, line feed	terminate word
→|	tab	terminate word
,	comma	assign symbolic address
=	equals	define parameter
/	slash	comment, or assign location
(	left parenthesis	initiate constant
)	right parenthesis	terminate constant

Ignored Characters

	form feed	end of a page of a source program

Special Characters

#	designates single register variable
$	designates multiregister variable (normally three registers)

Illegal Characters

All other characters are illegal to the Assembler except in comments and cause the error print ICH. Illegal characters are ignored.

ELEMENTS

Any group of letters, digits, and parentheses which represent binary values less than 2^{18} are elements. Values are assigned to elements by the Assembler or the Loader.

Number

Any sequence of digits delimited by punctuation characters forms a number.

Examples:

1
12
4371

The radix control pseudo-instructions indicate to the Assembler the radix to be used in number interpretation. The initial radix is octal. The pseudo-instruction DECIMAL indicates that all numbers are to be interpreted as decimal until the next occurrence of the pseudo-instruction OCTAL.

The pseudo-instruction OCTAL indicates that all numbers are to be interpreted as octal until the next occurrence of the pseudo-instruction DECIMAL.

Symbol

Any sequence of letters and digits beginning with a letter and delimited by punctuation characters is a symbol. Although a symbol may be any length, only the first six characters are considered, and any additional characters are ignored; symbols which are identical in their first six characters are considered identical. Note that period (.) is treated like a letter. However the user should neither begin nor end a symbol with the character period, since this use has been reserved for the FORTRAN system.

The Assembler has, in its permanent symbol table, definitions of the symbols for all the PDP-7 operation codes, operate commands, and many IOT commands (see Appendix 2 for a complete list). These may be used without prior definition by the user.

Examples:

LAC	is a symbol whose value of 200000_8 is taken from the operation code definitions.
SQRT	is a user-created symbol. When used as a symbolic address tag, its value is the address of the instruction it tags. This value is assigned by the Assembler.

Parameter Assignments

A parameter may be defined by use of the equal sign. The symbol to the left of the equal sign is assigned the value of the expression to the right.

Examples:

```
A = 6
EXIT = JMP I 20
TST = 477
```

If the parameter is used in a program as an operand address, the contents of the corresponding register are taken as the operand.

```
LAC TST     loads the AC with the contents of register 477
LAC (TST    loads the AC with the value 477
```

If the expression to the left of the equal sign does not consist of a single symbol or the expression to the right is not terminated by a tab or carriage return, the error message IFP is printed. An undefined symbol appearing in the expression to the right of the equal sign causes the error print UPA. No error message is printed if a defined symbol is redefined in a parameter assignment unless it is a permanent symbol. In this case, if the old value is not equal to the new, the error message RPS is printed and the symbol is redefined.

Variables

The user may request the Assembler to assign storage registers for him. These registers, whose value may be changed while the program is running, are designated as variables. A symbol (permanent symbols and pseudo-instructions must not be used) containing # or $ as one of its first six characters, which is not explicitly defined elsewhere by use of a comma or equal sign, is a variable. A symbol may contain a # any number of times without becoming multi-defined, but this character is required only once, not necessarily on the first occurrence of the variable. Currently unassigned variables are defined as the addresses of successive registers at the occurrence of the pseudo-instruction VARIABLES. The pseudo-instruction VARIABLES may be used repeatedly in a program.

Examples of variables:

```
#WHAT
WAI#T2
LEVEL#
```

If the pseudo-instruction VARIABLES is not used, the table of variables is automatically assigned a storage block at the end of the program. Upon loading the program, the contents of all locations assigned by use of the variable facility are zeros.

A variable containing $ causes a multiple number of registers to be reserved. The number of registers to be reserved may be controlled by the pseudo-instruction BAR n. The element or expression n specifies how many locations are to be allocated for $ variables. n is initially set to three.

Example:

To reserve seven registers for WANT and three register for GET at the occurrence of VARIABLES, use these instructions:

```
                    .
                DAC $GET
                    .
                    .
                    .
VARIABLES
                    .
                    .
                DAC $WANT
                    .
                    .
                    .
BAR 7
VARIABLES
```

Undefined Symbols

If any symbols remain undefined at the termination of assembly, they are automatically defined as the addresses of successive registers following the variables storage block and their definitions printed. That is, they are treated as variables containing #, but the user is informed of the assignment.

Current Address Indicator

The single character period (.) is assigned the current value of the location counter (see page 3) at assembly time. (Note that if a letter or digit appears on either side of the period, a symbol is formed, defeating the address indicating function of the period.)

Examples:

200/	JMP .+2	This instruction is identical to a jump to location 202.
	JMP .2	However, this instruction is a jump to the address represented by the symbol .2.

Literal

A storage register whose contents remain the same throughout the running of a program is designated as a constant. A constant can be represented by using a literal: an element or expression contained in parentheses. This type of element causes a register to be reserved in the constants table by the Assembler.

Example:

```
        ADD (1)
```

or

```
        ONE = 1
          .
          .
        ADD (ONE)
          .
          .
```

is equivalent to

```
        ADD ONE
          .
          .
          .
ONE,    1
```

except that in the first cases, the number 1 is stored automatically and its address substituted during loading. Unique constants are stored only once in the table, so that many uses of the same constant result in only one memory location being allocated for that constant. For example, the three statements,

```
ADD (1)
SAD (-1)
LAC (1)
```

result in two registers being allocated in the constants table, one for 1 and one for -1. The contents of the parentheses may be any element or expression:

```
LAC (JMP .-4)
```

The closing parenthesis may be deleted for brevity:

```
LAC (JMP .-4
```

Only one level of parenthesis may be used. Constants are automatically stored beginning in the register indicated by the location counter when START or PAUSE is encountered.

Indirect Addressing

If bit 4 in the binary representation of a memory reference instruction is 1, instead of taking the memory location referenced as the effective address of the operand, the contents of the location referenced are taken as the effective address of the operand. Indirect addressing is represented in the source language by the character I following the operation symbol thus:

```
      LAC I 500
500/  407
407/  -40
```

This instruction does not place the contents of location 500 in the accumulator as it would without the I. Rather the contents of location 500 are taken as the location of the quantity to be placed in the accumulator. After the execution of this instruction, the accumulator contains -40.

Since indirect addressing sets bit 4, the value of the element I may be represented as 020000.

EXPRESSIONS

Expressions are strings of elements separated by arithmetic or logical operators. Expressions represent numeric values less than 2^{18} in magnitude. The value of an expression is calculated by first substituting the numeric values for each of the elements and then performing the operations. The allowable operators are:

Operators		Combine by
␣	space	addition (1's complement)
+	plus	addition (1's complement)
-	minus	subtraction (1's complement)
&		logical AND
!		inclusive OR

When combining elements, operations are performed as encountered from left to right. In general, expressions may be of two classes: those expressions (called storage words or instructions) which occupy space in the binary version of the program; and those expressions which are used during the assembly process. Examples of such expressions would be symbolic address tags, location assignments, or parameter assignments.

COMMENTS

If the character slash (/) occurs, not immediately preceded by an element or expression, all characters between the slash and the next carriage return are ignored. Illegal characters (see page 7) are permitted within comments. Parity errors are ignored within comments also.

Examples:

```
/THIS IS A COMMENT
      LAC A          /AS IS THIS
400/  0              /AND THIS ALSO!
```

INSTRUCTIONS

Instructions are elements or expressions which make up the binary program (storage words). Memory reference instructions always have an effective operand address. This may be the operand address itself or the I address modifier and an address. In addition, if the address portion of the memory reference instruction is a literal, the effective address is the location in memory which contains that literal.

Examples:

SZA	A symbolic instruction from the operate group. An element.
LAC (407	An expression, consisting of the operation LAC and the literal (407.
DAC I XIT	An expression, consisting of the operation DAC, address modifier, and address. The effective address is a combination of the last two elements.
060342	An element, numerical representation of the preceding instruction where XIT is the tag for location 342.

STATEMENTS

Statements are combinations of elements, expressions, and comments delimited by carriage return, line feed pairs (↵ ↓). To achieve clarity, the components of a statement normally appear in three areas or fields on a line delimited by tabs (→|) or the carriage return, line feed pair (↵ ↓). In the leftmost field are parameter assignments, location assignments, or symbolic address tags. In the middle field are instructions or constants. Though the rightmost field is usually used for comments, they may appear in any of the fields.

Examples:

```
STORE = 30
TST = 777776                    /PLACE -1 IN TEST WORD
/THIS IS A ROUTINE TO SORT A TABLE
40/
BEGIN,                          CLA          /START PROGRAM; CLEAR AC
                                LAC XIT -4
REMOVE,                         AND (TST
                                  .
                                  .
                                  .
400/                            126          /LENGTH OF 1ST LIST
```

The Assembler interprets both carriage return, line feed (↵ ↓) and tabs (→|) as field delimiters. Consequently, in addition to the statement format suggested above, any format can be used which separates statement components with a carriage return or tab, regardless of line length. For example, when assembling tables or repetitive instructions, which would take large amounts of space if listed on individual lines, the following format could be employed.

```
table,    →| 15 ↵ ↓
          →| 35 ↵ ↓
          →| 31 ↵ ↓
          →| 20 ↵ ↓
          →| 12 ↵ ↓
          →| 01 ↵ ↓
          →| 25 ↵ ↓
          →| 34 ↵ ↓
```

may be listed thus:

```
table,  →| 15   →| 35   →| 31 ) ↓
        →| 20   →| 12   →| 01 ) ↓
        →| 25   →| 34 ) ↓
```

Similar treatment may be given instructions:

```
RR9,    RTR     RTR     RTR
        RTR     RAR
```

CHAPTER 4

PROGRAM PREPARATION

A program is prepared in ASCII or FIODEC code on 8-channel punched paper tape, using an off-line typewriter or the on-line program EDITOR with the PDP-7. In general, a program should begin with about 2 feet of tape feed (only the feed hole punched) to allow easy placement in the reader. Deleted characters (seventh hole punched) and tape feed may be used freely throughout the tape and are always ignored.

PROGRAM TAPE

The program tape itself consists of three sections, described below.

Title

All text between the first character (other than initial carriage return, line feed pairs) and the next carriage return, line feed is taken as the title of the tape. The first line of all symbolic programs should be a title line. This title is printed on the teleprinter at assembly time, as well as punched in readable format on the front of the binary tape. The title is not subject to normal program conventions; it need not begin with a slash to indicate it is not a part of the program.

Program Body

The text consisting of statements and pseudo-instructions, follows the title. Redundant carriage return, line feeds and tabs are ignored and may be used for formatting. A suggested program body format is described in the preceding section.

The character form feed should be used as a page separator (with both the tape EDITOR and Teletype Model 33KSR) although pages have no meaning to the Assembler. New pages should begin with a carriage return, line feed.

Deleted characters (rubout overpunch), tape feed, and form feeds are ignored by the Assembler during processing.

NOTE: To avoid erroneous assembly, the first statement in a program or the first statement after an absolute address assignment must not contain more than one symbol which is undefined at that time.

Terminating Pseudo-Instruction

The last line of a program consists of the pseudo-instruction START, or PAUSE, followed by either the starting address of the program and a carriage return, line feed or by a carriage return, line feed alone. Either pseudo-instruction indicates the end of the symbolic program.

If START is followed by an address, control is transferred to that address when loading is completed. In this manner, the program is immediately executed. If PAUSE is followed by an address, the computer halts. By depressing CONTINUE, control is transferred to the specified address and the program executed. If either START or PAUSE is used with no ensuing address, the computer halts after loading the program. To execute the program, the user must place the starting address in the ADDRESS switches and depress START.

Constants are stored starting at the address in the location counter when START or PAUSE is encountered on the last source tape. This normally follows the program unless the current location was reset (using the slash) immediately preceding the START.

LOCATION ASSIGNMENT

The use of a slash (/), if immediately preceded by an element or expression, sets the location counter equal to the value of that element or expression.

Examples:

300/	LAC (56	
BEG-240+A/	LAC (56	The instruction is stored in location number BEG-240+A.

SYMBOLIC ADDRESS TAG

An element or expression which represents a location in memory may be assigned a value in a number of ways. The user could utilize the parameter assignment feature thus:

```
A=.          ADD 100
```

The symbol A is assigned the value equal to the location in which the instruction ADD 100 is placed by the Assembler. If the symbol already has a definition, it would be redefined. The user can reference this location later by the symbol A:

```
             JMP A
```

The s.mplest way to assign a value to a tag is by use of the comma.

```
A,           ADD 100
```

The value of A would be the same as in the first case; the only difference would occur if A had previously been defined, which would result in the diagnostic MDT.

The Assembler, if possible, sets the element or expression to the left of a comma equal to the current value of the location counter. If the comma is not preceded by an expression, the diagnostic IFC occurs. A single undefined symbol or an expression containing only one undefined symbol preceding the comma has its value set equal to the current location.

An expression preceding the comma which contains more than one undefined symbol causes the error print TUA. If the expression to the left of the comma contains no undefined symbols but is equal in value to the current location, it is ignored; otherwise the error print MDT occurs. This feature is useful for verifying table lengths.

Examples:

A,	Where A and B are previously undefined symbols.
B+1,	
101,	Where the number is the same as the current value of the location counter.
GEORGE+HARRY-4,	Where either GEORGE or HARRY are previously undefined symbols.

PROGRAM EXAMPLE

```
THIS IS A SAMPLE PROGRAM.
/IT ROTATES A BIT THROUGH THE AC AT A RATE
/DETERMINED BY THE AC SWITCHES

GO,     LAS
        SPA!CMA         /EXAMINE AC SWITCHES
        JMP GO          /WAIT UNTIL ACS0=0
        DAC CNTSET
        LAC ONE         /1 IS A CONSTANT
        DAC BIT
        CLL             /CLEAR THE LINK

LOOP,   LAC CNTSET
        DAC CNT
        LAC BIT

LOOP1,  ISZ CNT         /LOOP UNTIL CNT GOES TO ZERO
        JMP LOOP1       /JUMP TO PRECEDING LOCATION
        RAL
        DAC BIT         /ROTATE BIT
        LAS
        SMA             /IF ACS0=1, RESET TIME CONSTANT
        JMP LOOP
        JMP GO

/STORAGE FOR PROGRAM DATA
CNT,    0
BIT,    0
CNTSET, 0
ONE,    1

START GO
```

The result of assembling this and requesting a symbol print would be:

```
THIS IS A SAMPLE PROGRAM

GO      22
LOOP    31
LOOP1   34
CNT     44
BIT     45
CNTSET  46
ONE     47
```

The same program could be written using the additional facilities for constants, variables, and current location indicator.

```
THIS IS A SAMPLE PROGRAM
GO,         LAS
            SPA!CMA
            JMP GO
            DAC #CNTSET
            LAC (1
            DAC #BIT
            CLL
LOOP,       LAC CNTSET
            DAC CNT
            LAC BIT
            ISZ #CNT
            JMP .-1
            RAL
            DAC BIT
            LAS
            SMA
            JMP LOOP
            JMP GO
START GO
```

In this case, the constant 1 occupies location 47 in the constant area following the program.

CHAPTER 5

ASSEMBLER OUTPUT

The Assembler processes the symbolic source tape, types the program title, and punches a binary object tape. Error messages are typed out during assembly in the formats described on page 27. A printout of user-defined symbol values may be requested at the completion of assembly.

The user's program is punched on the object tape in a code called FF Binary. While storage words on binary tapes may be read directly from bits 6-1, a FF Binary tape is punched in a complicated form to be used by a sophisticated loader to produce the storage words of the user's program. For this reason, no attempt is made to explain how to read or interpret the FF representation of a program. See the FF Loader (Digital 7-19-1) for this information.

In the ensuing sections, memory locations are given for the 8K memory configuration. These locations also work properly with 4K memories or extended memory configurations, though the 4K locations are actually 10000 less.

OBJECT TAPE

The tape is punched in the reverse direction from which it will be loaded (the termination block is punched first; the readable title, last). The tape consists of five sections, explained below in order of appearance when loading. See Figure 1 for an example.

Title

The first data on the object tape is the title, punched in readable form.

Loader

Following the title is the FF Loader in binary, preceded by a 6-instruction loader-loader in read-in-mode also punched in binary. See page 35 for descriptions of the role of each loader.

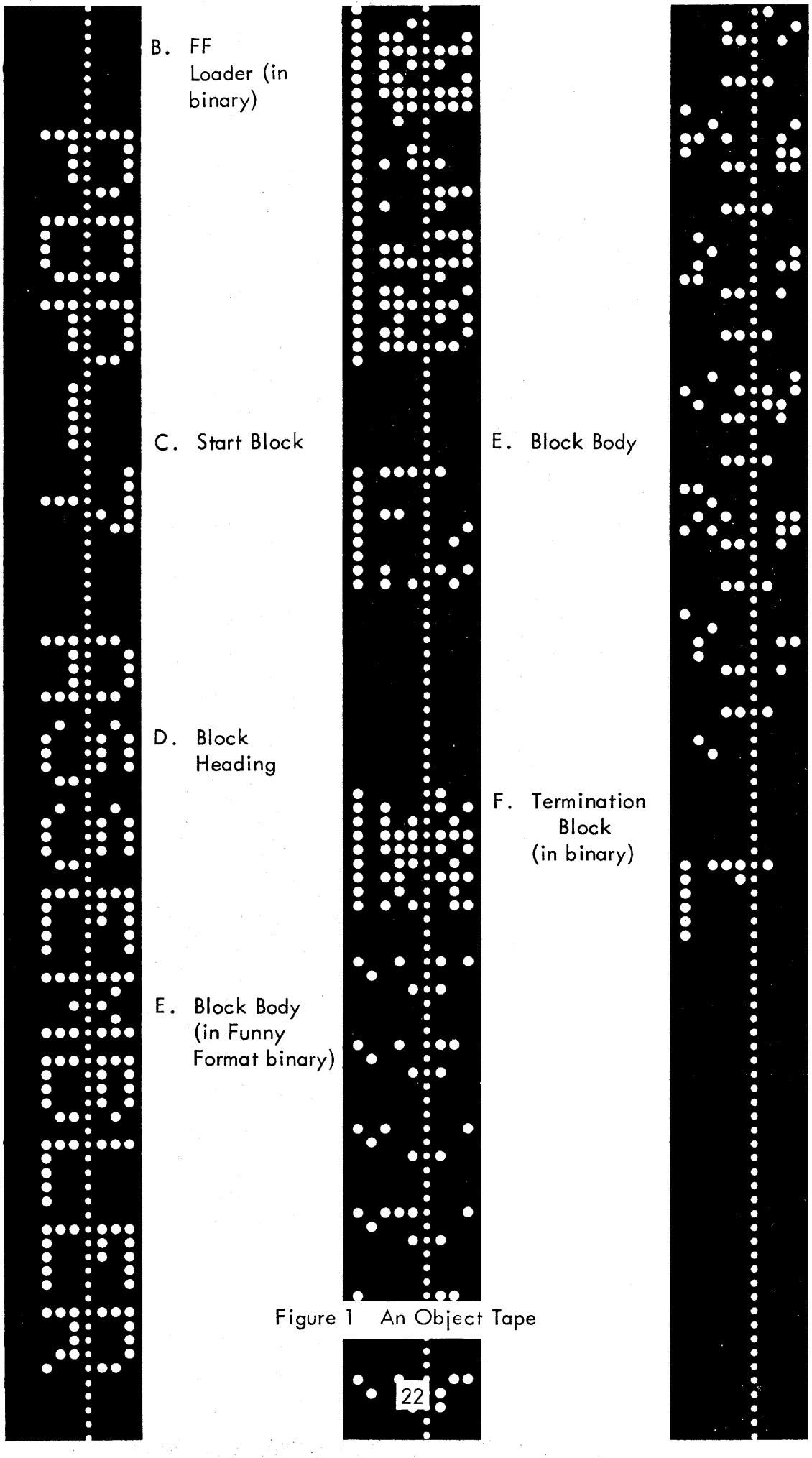

Figure 1 An Object Tape

If the FF Loader is expected to be in memory when the object tape is loaded, the pseudo-instruction NOINPUT can be used to instruct the Assembler to suppress punching of the loader and punch instead a JMP 17600, the starting address of the FF Loader. A tape in this form may be loaded by the loading commands in DDT-7 as well as through a FF Loader already in memory.

Starting Block

Three binary words are punched:

1. The instruction HLT or NOP depending upon whether PAUSE or START was used.

2. A JMP to the starting address following the START or PAUSE. If no address followed, a HLT is punched.

3. The address of the register preceding the constants table (location counter when START or PAUSE was encountered). Constants are stored beyond this location; the program is loaded backward from this location.

The following table summarizes the first two words of the starting block produced by the possible pseudo-instruction forms.

	Address Specified	Address Not Specified
START	NOP JMP ADDRESS	NOP HLT
PAUSE	HLT JMP ADDRESS	HLT HLT

Data Blocks

The program is punched in data blocks, each consisting of the following components.

Block Heading

Three binary words:

1. DAC LA where LA is the largest address in which an instruction from the block will be loaded (the first instruction encountered by the loader). The block is loaded backward from this location.

2. -N where N is the number of words in the block on tape. The loader reads only N words in FF Binary, then looks for new commands. Since every line is read, N words are equivalent to 3N lines on tape.

3. The checksum, the sum of all words in the data block excluding the checksum. This is compared to a computed sum to insure correct loading.

Block Body

The words in the block body are punched in the alphanumeric mode, delivering 24 information bits to each 3 lines of tape (FF Binary). Eighteen of these form a data word; the remaining six are a code word which indicates to the FF Loader the type (or use) of the data word. The words are punched as the symbolic program is read. Since this tape is produced backward, the first instruction in the block body when loading would correspond to the last instruction in this block of the symbolic program.

Termination Block

Two binary words:

1. A SKP instruction to indicate the end of the tape to the loader. This also causes the loader to execute the instructions generated by the terminating pseudo-instruction (see Starting Block).

2. A dummy word to stop the reader. This second word is necessary since the loader commands the reader to read the next word while the previous word is being processed.

SYMBOL PUNCH

The ensuing sections are concerned with how to obtain and use a symbol punch. No effort is made to explain the internal procedures used by DDT or the Assembler to assimilate a symbol punch or how the formats as punched in FF Binary differ.

A symbol punch is the definition of source program symbols, punched on the object tape in one of two formats. When encountered by the loader, the symbol definitions are ignored; the only advantage of punching symbol definitions is for use in debugging work with DDT-7 (to enable the user to refer to his own symbolic tags) or for the Assembler. The Assembler can load Assembler format symbol definitions into its permanent table from a symbol punch. When assembling programs which refer to an often-used subroutine, a printout routine for example, the subroutine need not be assembled with each program to obtain definitions for the subroutine symbols. Instead a symbol punch of the necessary symbols can be loaded into the Assembler and these symbols referred to without further definition by programs being assembled.

The symbol definition punching formats are different for DDT-7 and the Assembler; the user must be careful to obtain the punching format required for his purpose.

Symbol Punch for DDT-7

The Assembler normally punches symbol definitions for DDT-7. Each symbol defined during assembly is punched on the object tape in DDT-7 format at time of definition unless suppressed by the AC switch settings (see page 32).

Symbol Punch for the Assembler

The pseudo-instruction SYMBOLS causes the Assembler to suspend DDT symbol punching, as controlled by the AC switches, until the occurrence of NOSYMBOLS. In the interim as symbols are defined, they are punched in Assembler format. The pseudo-instruction NOSYMBOLS terminates the punching of Assembler format symbol definitions begun by SYMBOLS and restores the DDT format specified by the AC switches (see Appendix 2 for an example).

Because of internal operating procedures, for a symbol punch to be loaded into the Assembler, the tape must not contain the FF Loader. Consequently, the pseudo-instruction NOINPUT

must be included on any tape which will be used to add symbol definitions to the Assembler's table at a later time. The symbol punch may be added to the Assembler's symbol table by following the directions for loading a symbol punch on page 32. Upon loading, the Assembler reads the tape ignoring all data except symbol definitions which are added to the previous symbol table. Symbols which appear both on tape and in the Assembler symbol table are redefined to the tape definitions (see page 32).

Symbol Punch by PUNDEF

The definitions of symbols occurring in the list following the pseudo-instruction PUNDEF are punched on the output tape in Assembler format. Such symbols must be defined prior to the use of PUNDEF. This is equivalent to surrounding the specified symbols with the SYMBOLS - NOSYMBOLS combination except that with PUNDEF, DDT symbols can also be punched when the symbol is defined. In addition, PUNDEF automatically invokes the pseudo-instruction NOINPUT suppressing the punching of the FF Loader.

The PUNDEF list must consist of the symbols the user wishes punched, separated by commas, and terminated by a carriage return. The error prints LIT, IFL, LNS may occur indicating, respectively, illegal terminating punctuation and two types of illegal format in a list.

Note that the PUNDEF tape may be physically separate to avoid having to load an entire program tape to obtain a few definitions. In this case, the proper format is:

```
TITLE
PUNDEF SYMB1, SYMB2, SYMB3, etc.
START
```

This tape should be assembled before symbol definitions are erased from the Assembler symbol table (that is, by depressing START to begin assembling the tape). ACS 15 must have been left down during the assembly which produced the desired symbols (see page 32).

PUNCH

The pseudo-instruction PUNCH causes the expression following punch to be punched on the tape in binary. With PUNCH, the user can obtain a tape which can be loaded by a binary

loader such as the RIM Loader (see PDP-7 manual) directly into memory. (Binary, not FF Binary, is read three lines at a time, six bits per line (holes 1-6) to form an 18-bit word. Only lines whose eighth hole is punched and whose seventh, delete, hole is unpunched are read.) The FF Loader is not needed to load binary tapes. Consequently, after the job of the FF Loader is completed, other data could be loaded into locations 17600 to 17761 by using PUNCH with RIM. (See page 35 for a description of the role of the various loaders.)

SYMBOL PRINTOUTS

After the completion of assembly, symbol printouts can be requested in alphabetic or numeric order (see operating instructions). These give all symbol definitions which were added to the Assembler's permanent symbols during the assembly. Symbols can be printed only once; after printing, an internal indicator is set to suppress the printing of any symbol which was printed previously. Both printouts can be obtained in succession, however.

Undefined Symbol Assignments

At the end of assembly, before the loader is punched, any undefined symbols are automatically defined. Each undefined symbol which was used in a storage word is defined as the address of a register at the end of the program and the definition printed. If the symbol was not used in a storage word, the symbol is printed but not defined.

ERROR MESSAGES

The error message appears in one of the following three formats. With the exception of SCE (storage capacity exceeded) and ILP (illegal parity), assembly continues automatically after the error message has been printed.

Format A

The appearance of a diagnostic printed in format A:

ERROR PREVIOUS VALUE SYMBOL NEW VALUE

Whether the new value was actually incorporated into the symbol table depends upon the particular error.

Error	Meaning
MDT	A previously defined symbol was redefined with a comma. (See page 17.)
RDA	An attempt was made to redefine a permanent symbol with a comma. The symbol was not redefined.
RPS	A permanent symbol was redefined. (See page 8.)

Format B

The appearance of a format B diagnostic is:

ERROR OCTAL ADDRESS SYMBOLIC ADDRESS

The general error message is printed in Format B.

Error	Meaning
IFC	Illegal format in symbolic address tag. The tag is ignored. (See page 17.)
IFI	An expression using CHAR or FLEX was formed improperly. (See page 38.)
IFL	Illegal format in a PUNDEF list. (See page 26.)
IFP	Illegal format in a parameter assignment. The assignment is ignored. (See page 8.)
IFS	START or PAUSE used incorrectly. Assembly continues as if START or PAUSE had been used with no expression following. (See page 17.)
ILF	Illegal format in a pseudo-instruction such as BAR. The pseudo-instruction is ignored.
INS	An illegal format in a PUNDEF list--two commas appeared in a row or a digit appeared.
LIT	An illegal character was found in a PUNDEF list. The character is taken as a terminator.
MDT	The value of the complex symbolic address assignment (tag) and the location counter disagree. The symbolic address tag is redefined if possible. (See page 17.)

Error	Meaning
SCE	Storage capacity of the symbol table was exceeded. No recovery is possible.
TUA	Too many undefined symbols appeared in a symbolic address assignment (tag). Location counter remains unchanged. (See page 17.)
UBR	An undefined symbol appeared in a BAR pseudo-instruction. The setting of BAR remains unchanged.

Format C

The appearance of a format C diagnostic is:

ERROR OCTAL ADDRESS SYMBOLIC ADDRESS CAUSE

Format C is an expanded version of Format B. CAUSE is additional information to help the programmer ascertain the cause by an undefined symbol which will be printed. ASCII codes are printed when the cause is a character.

Error	Cause	Meaning
ICH	character	A character not part of the Assembler's source language was used. The character is ignored. (See page 7.)
ILP	character	A character read from tape did not have an odd number of holes across the line. Place the correct character (if possible) in bits 12-17 of the ACS and press CONTINUE.
UAA	symbol	An undefined symbol appeared in an absolute address assignment (/). The current address indicator remains unchanged.
UPA	symbol	An undefined symbol appeared in a parameter assignment. The assignment is ignored. (See page 8.)
UPN	symbol	An undefined symbol appeared in a PUNCH pseudo-instruction. The symbol is ignored.

Error	Cause	Meaning
UST	symbol	An undefined symbol appeared in a START or PAUSE instruction. The symbol is ignored and the START or PAUSE taken alone. (See page 17.)

CHAPTER 6

OPERATING THE ASSEMBLER

OPERATING INSTRUCTIONS

1. Load the Assembler by placing the binary tape of the Assembler in the reader and starting the RIM (Readin Mode) Loader in location 17770. It is assumed that the RIM Loader will be prestored in core memory. Press START.

2. Place the symbolic source language tape in the reader, and set the ADDRESS switches to 20. Set AC switch 10 up to indicate ASCII symbolic tape (down for FIODEC).

3. The operator may choose, at this point, to begin a normal assembly or command the Assembler to execute special functions as indicated by the AC switches.

   a. Normal assembly, restore symbol table to permanent symbols--depress CONTINUE.

   b. Special functions--set ACS (see page 32) and depress START. When the pseudo-instruction START or PAUSE in the source tape is encountered, the Assembler stops with all ones in the AC.

   NOTE 1: To assemble more than one symbolic tape into one binary output tape (a main program and subroutines, for example), the sequence of steps in assembly is altered. After Step 3, the next symbolic tape is put in the reader. With 20 in the ADDRESS switches, depress the START key. Repeat these steps for remaining symbolic tapes. The title of the first tape and the START from the last tape are incorporated into the binary output tape unless otherwise specified by ACS3. When all desired symbolic tapes have been assembled, continue with Step 4.

4. To complete the normal assembly, depress CONTINUE. The Assembler punches the variables, the undefined symbols (listing these on the on-line Teletype), the starting block, and the loader and punches the title in readable form. Then the Assembler stops with all ones in the AC. The assembly of a loadable object tape is complete at this point.

> NOTE 2: To restore the Assembler's symbol table to permanent symbols before beginning another assembly, put up AC switch 15. Then after completing Step 5, return to Step 2. If no symbol printouts are desired, depress CONTINUE and return to Step 2.

5. To print the symbol definitions, set the AC switches (see below) and depress the CONTINUE key. When the printouts are completed, a halt occurs with all zeros in the AC.

LOADING A SYMBOL PUNCH

A symbol punch in Assembler format (see page 25) can be loaded into the Assembler at any time, but the suggested time is prior to assembling the first tape (before Step 2). To load a symbol punch, place the tape in the tape reader, set ADDRESS switches to 4, and depress the START key. The symbol definitions are added to the Assembler's permanent symbol table; restoring the Assembler's symbol table has no effect on them. To start an assembly, return to Step 2 above.

AC Switch Control

Throughout the assembly of a program, ACS 10 indicates the symbolic tape code: ASCII (up) or FIODEC (down). This may be reset if desired for each program or subprogram assembled. In Step 3, the AC switches perform the following functions:

AC Switch Up	Meaning
0	Examine AC switches 1-4 further.
1	Suppress punching.
2	Suppress punching of symbols for DDT-7. Save space on tape unless needed for DDT work.

AC Switch Up	Meaning
3	Take the title on this tape. The title from the current tape replaces the first tape's title on a single binary output tape (see Note 1).
4	Restore the Assembler when restarting an unfinished assembly.

In Step 5, the switches have the following meaning:

AC Switch Up	Meaning
15	Restore symbol table to permanent symbols (after symbol printouts if requested). Starting the next assembly with CONTINUE has the same effect.
16	Symbol printout, numerical order.
17	Symbol printout, alphabetical order.

In addition, the following switches have meaning throughout an assembly.

AC Switch Up	Meaning
10	ASCII symbolic tape
11	Causes all printing to be done on the high speed line printer.

HALTS DURING ASSEMBLY

The following are all possible abnormal halts during assembly, the cause, and the action which can be taken.

Cause	AC Contents	Action
illegal parity	character	1. Place correct character in ACS. 2. Depress CONTINUE.
storage capacity exceeded, printout SCE	——	Segment program and reassemble.
offensive interrupt	status register	See below.

When a device other than the reader, punch, or Teletype causes a program interrupt, the Assembler halts with the status register displayed in the AC. CONTINUE clears some standard device flags, not including those of the devices used by the Assembler, and proceeds.

If this fails to clear the offending device's flag, the instruction to clear that flag must be loaded and executed. To accomplish this, deposit the required IOT in location 6. In location 7, deposit a JMP to the register specified by the program counter when the halt occurred. Then set the ADDRESS switches to 6 and depress START.

Since the Assembler uses the program interrupt, users at installations which have special equipment connected to the program interrupt system must take special care to insure that the associated flags are cleared before assembly starts. The devices which are cleared by the Assembler are:

Perforated Tape Reader
Perforated Tape Punch
Teleprinter
Clock
Type 30D Display
Light Pen
Character Generator
Type 57A Mag Tape Control
Card Reader
Card Punch
Line Printer

THE FF LOADER

The Assembler performs its action in one pass; that is, the source language tape is processed only once to produce the binary object tape including a 162 (octal) location FF Loader. Certain functions which cannot be handled at assembly time must be handled by this loader when the program is loaded into memory.

The first of these is the insertion of symbol definitions for symbols which were undefined during assembly. When the Assembler first encounters an undefined symbol, the symbol is tagged as

undefined and assigned a register. Each time the symbol is used before it is defined, the address of the assigned register along with an identifying code is punched on the binary object tape. When the symbol is subsequently defined, both the defined value and the address of the assigned register are punched on the binary tape. The assigned register is used to contain the defined value during loading. When loading, since the end ot the binary tape which is punched last is the end read first, the definition of a symbol is encountered before any use of the undefined symbol. Thus, the loading process is accomplished correctly.

The second problem handled by the loader is the setting up of constant tables. When constants are encountered during assembly, the Assembler does not know where they are to be stored. Thus, the constants are punched on the binary output tape with an identifying code. When a constant is encountered by the loader, the constant table, which is built up by the loader at the end of Assembler assigned storage, is searched for previous assignments. If no assignment is found, the new constant is added to the table. The address of the constant is recorded, and this address replaces the constant.

The FF Loader uses registers 7 and 10 during the loading process. Upon completion of loading, register 7 contains the address of the first location after the constant table, which is normally the first free location available following the program. This number is also in the AC at the time the first instruction of the program is executed to allow the program to set up storage areas after the program.

LOADING THE OBJECT PROGRAM

To load the object tape, place the tape in the reader title end first, and depress START with 17770 in the ADDRESS switches. The RIM Loader must be in memory. The FF Loader is normally punched at the beginning of the object tape in binary. It is preceded by a 6-instruction loader-loader punched in Readin Mode for the RIM Loader. If the object tape has no loader (NOINPUT was used), the FF Loader must be in memory in addition to the RIM Loader. (Read in any object tape with the loader punched on it.) The FF Loader occupies registers 17600 to 17761. While the FF Loader is being read in, the loader-loader is stored from 17572 to 17577.

When an object tape is being loaded normally, the RIM Loader reads in the loader-loader which in turn reads in the FF Loader. The FF Loader then loads the user's object program (punched

in FF Binary). If START followed by an address was used to terminate the program, it is executed immediately. If PAUSE was used followed by an address, depressing CONTINUE causes execution to begin.

Halts During Loading

A checksum is computed as each block is read from tape and compared with the checksum read from the block heading. If these differ, the loader halts, displaying (in the ACCUMULATOR) a word whose 0 bits are those which differ between the computed and the read checksum. If repeated loadings cause the same difference to appear in the AC lights, the object tape is probably faulty and should be reassembled. If the difference varies, the computer or reader may be the difficulty. In any case, depressing CONTINUE causes the loader to ignore the checksum seen.

APPENDIX 1

PSEUDO-INSTRUCTION

There are certain symbols which, when used in a program, are commands directly affecting the assembly process without appearing in the output. These pseudo-instructions have no other effect upon assembly and are ignored when forming storage words. These symbols may not be used as address tags or variable names.

RADIX CONTROL

DECIMAL
OCTAL — All numbers not imbedded in symbols can be interpreted as decimal or octal, respectively. The initial mode is octal (see page 7).

TABLE FORMATION

SHIFT — Causes the word preceding SHIFT to be rotated left nine binary positions and masked with 777000, leaving the right half blank. SHIFT is useful in forming double entry tables, tables with one value stored in the leftmost nine bits and another value in the rightmost nine bits.

SYMBOL TABLE CONTROL

EXPUNGE — Removes all symbols (except pseudo-instructions) from the Assembler's symbol table.

FIX — Resets the symbol table so that all currently defined symbols are part of the permanent symbol table. (This instruction overrides the ACS option to restore at assembly time.)

VARIABLE CONTROL

VARIABLES — Places all currently defined variables at addresses beginning with the address indicated by the location counter before processing of the program continues.

BAR N — Allows N registers for each variable containing the character $ (see page 10).

PUNCH CONTROL

PUNDEF SYMB1, SYMB2, SYMB3	Punches definitions of the listed symbols in Assembler format.
SYMBOLS	Punches ensuing symbols in Assembler format.
NOSYMBOLS	Stops the punching of Assembler symbols and restores the mode set by ACS concerning DDT symbols.
PUNCH A	Punches the value of the expression A in binary at this time.

See pages 25-26 for a more detailed explanation.

END OF PROGRAM

NOINPUT	Suppresses punching of the loader--punches a JMP 17600, the starting address of the FF Loader.
START A	Upon loading, causes the program to start at register A.
PAUSE A	Upon loading, causes the computer to halt. When CONTINUE is pressed, the program will start at A. (See page 17.)

TEXT HANDLING

The following pseudo-instructions pack character codes (to be output upon execution of the program) into storage words in the computer.

CHAR RA CHAR LA CHAR MA	Code values for single characters (represented here by A) are assembled into the right, left, or middle six bits of the word following to the character mode below.
FLEX ABC	Code values for three characters (represented here by ABC) are assembled into a single register from left to right according to the character mode below.
TEXT /THIS IS TEXT./	A string of characters of any length is assembled, three to a word, into successive registers according to the character mode below. The string is terminated by the second occurrence of the delimiting character chosen by the user. / has been chosen here. In order to separate the string from other data following it, a termination code determined by the character mode is inserted automatically after the last character code of the string. If FIODEC characters

are used, double-punch characters--center dot, period (:), center dot, comma (;) and vertical stroke, capital S ($)--may not be used as delimiters. Note the space after the instruction TEXT.

CHARACTER MODE CONTROL

The mode control pseudo-instructions specify the character code to be used when evaluating the instructions TEXT, FLEX, and CHAR. Initially, the mode is Teletype.

TELETYPE — All characters are to be converted to their respective 6-bit packed Baudot codes. In order to print on line on the Teletype Model 33KSR or 28KSR using Output Package (Digital 7-10-O), it is necessary to restrict the characters used in these pseudo-instructions to those listed in Appendix 4 in the column titled Baudot. The termination code in TELETYPE mode is the code 00.

The 6-bit code consists of a 5-bit Baudot character (most significant bits), see PDP-7 Manual, F-75P, and a case bit (least significant bit) which is 1 if the character is in upper case in this code and 0 if the character is in lower case. Tab is converted to enough spaces to space to the next tab stop. Tab stops are internally set to every ten spaces.

ANELEX — Indicates that all character translations are to DEC high-speed printer code. (See PDP-7 Manual, F-75P). The termination code in ANELEX mode is 00.

FIODEC — Indicates that all character translations are to FIODEC code. The termination code in FIODEC mode is 13 (stop code). The conversion from ASCII to FIODEC code is not always 1 to 1 since case shifts do not generate codes in ASCII whereas in FIODEC they appear as the first character of the string of characters to which they apply. TICTOC (Digital 7-11-IO) provides for printing FIODEC codes. on the Teletype.

Codes for the Teletype characters used in the example below are:

Character	Code
A	60
B	46
C	34
Ø	33
.	17

..... A code is 30 (11 000 in binary); appending a 0 for lower case yields the binary code 110 000 or 60 in octal.

Examples of use of characters input pseudo-instructions:

Symbolic	Result	
CHAR RA	000060	
CHAR MB	004600	
CHAR 1Ø	330000	
FLEX A.B	601746	
TEXT .ABC.	604634	
	000000	/NOTICE 00 TERMINATION CODE ADDED
CHAR RA+4	000064	
LAW CHAR R.	760017	

APPENDIX 2

PERMANENT SYMBOLS

MODIFYING PERMANENT SYMBOL TABLE

It is often desirable to make modifications to the permanent symbol table of the PDP-7 Assembler. This may be accomplished in the following way:

1. Assemble a symbolic tape which has the following format:

```
TITLE
NOINPUT
SYMBOLS
(body)
NOSYMBOLS
START
```

where the body consists of all symbols with definitions as parameter assignments that the user wishes to add to permanent symbol table.

2. Splice this on the end of a binary tape of the Assembler, cutting the Assembler just before the last block on tape (the termination block, two binary words), and cutting the other tape after the first binary word following the title.

If the user wishes to delete any permanent symbols, the entire permanent symbol table must be removed and replaced with a table containing only those symbols desired.

To cause symbols which are already defined to be punched on the symbol tape being prepared, define the symbol equal to itself. For example, to cause the symbols LAC and SZA to be punched on the new binary symbol table tape, write in the body of the symbolic tape

```
LAC = LAC
SZA = SZA
```

To remove the entire symbol table, cut the Assembler tape after the block which consists of one binary word (SKP), which should be before the next to the last block on tape.

BASIC SYMBOLS

DAC	040000
JMS	100000
DZM	140000
LAC	200000
XOR	240000
ADD	300000
TAD	340000
XCT	400000
ISZ	440000
AND	500000
SAD	540000
JMP	600000
IOT	700000
OPR	740000
CAL	0
LAW	760000
LAM	777777
I	020000
NOP	740000
CLA	750000
CLL	744000
CMA	740001
CML	740002
CLC	750001
CCL	744002
RAL	740010
RAR	740020
RTL	742010
RTR	742020
RCR	744020
RCL	744010
OAS	740004
LAS	750004
LAT	750004
HLT	740040
SPA	741100
SMA	740100
SPL	741400
SML	740400
SZA	740200
SNA	741200
SKP	741000
SZL	741400
SNL	740400
GLK	750010
STL	744002

XX	740040

Interrupt

IOF	700002
ION	700042
ITON	700062
CAF	703302

I/O States

IORS	700314
SKP7	703341

Clock

CLSF	700001
CLOF	700004
CLON	700044

Perforated Tape Reader

RSF	700101
RSA	700104
RSB	700144
RRB	700112
RCF	700102

Tape Punch

PSF	700201
PLS	700206
PCF	700202
PSA	700204
PSB	700244

Keyboard

KSF	700301
KRB	700312

Teleprinter

TSF	700401
TLS	700406
TCF	700402
TTS	703301

Display 30D

DXL	700506
DXS	700546
DYL	700606
DYS	700646
DLB	700706
DXC	700502
DYC	700602

Light Pen Type 370

DSF	700501
DCF	700601

EXTENDED SYMBOLS

Type 57A Mag Tape

MTS	707006
MTC	707106
MCD	707042
MNC	707152
MRC	707244
MRD	707204
MTRS	707314
MCEF	707322
MEEF	707342
MIEF	707362
MCWF	707222
MEWF	707242
MIWF	707262
MSEF	707301
MSWF	707201
MSCR	707001
MSUR	707101
MCC	707401
MCA	707405
MWC	707402
MRCA	707414
MDEF	707302
MDWF	707414

Card Punch

CPSF	706401
CPSE	706444
CPLR	706406
CPCF	706442

Symbol Generator Type 33

GPL	701002
GPR	701042
GLF	701004
GSF	701001
GCL	700641
GSP	701034

Card Reader

CRSF	706701
CRSA	706704
CRSB	706744
CRRB	706712

Line Printer

LPSF	706501
LPCF	706502
LPLD	706542
LPSE	706506
LSSF	706601
LSCF	706602
LSLS	706606
LPB-1	706504
LPB-2	706524
LPB-3	706544
PRI	706604
PAS	706624

DECtape

MMRD	707512
MMWR	707504
MMSE	707644
MMLC	707604
MMRS	707612
MMDF	707501
MMBF	707601
MMEF	707541

Extended Arithmetic Element Type 177

EAE	640000
LRS	640500
LRSS	660500
LLS	640600
LLSS	660600
ALS	640700
ALSS	660700
NORM	640444
NORMS	660444
MUL	653122
MULS	657122
DIV	640323
DIVS	644323
IDIV	653323
IDIVS	657323
FRDIV	650323
FRDIVS	654323
LACQ	641002
LACS	641001
CLQ	650000
ABS	644000
GSM	664000
OSC	640001
OMQ	640002
CMQ	640004

Automatic Priority Interrupt Type 172

CAC	705501
ASC	705502
DSC	705604
EPI	700004
DPI	700044
ISC	705504
DBR	705601

Precision Incremental Display Type 340

IDLA	700606
IDSE	700501
IDSI	700601
IDSP	700701
IDRS	700504
IDRD	700614
IDRA	700512
IDRC	700712
IDCF	700704

Memory Extension Control Type 148

SEM	707701
EEM	707702
LEM	707704
EMIR	707742

Serial Drum Type 24

DRLR	706006
DRLW	706046
DRSS	706106
DRCS	706106
DRCS	706204
DRSF	706101

DRSN	706201
DRCF	706102

Multiplexer Control Type 139

ADSM	701103
ADIM	701201

A-to-D Converter Type 138B

ADSC	701304
ADRB	701312
ADSF	701301

APPENDIX 3

THE FORTRAN ASSEMBLY SYSTEM

The FORTRAN Assembler is a modified version of the PDP-7 Assembler. The FORTRAN Assembler produces a relocatable object program unless absolute address assignments are used. Relocatable programs are loaded by the Linking Loader consecutively from location 22. The loader also joins programs by supplying definitions for symbols which are referenced in one program and defined in another. Any program written for the PDP-7 Assembler can be assembled by the FORTRAN Assembler. The differences present in the FORTRAN Assembler are:

1. The addition of pseudo-instructions to define symbols used by the loader to link relocatable programs to each other. These pseudo-instructions are EXTERNAL, INTERNAL, and LIBFRM.

2. Error printouts associated with these three pseudo-instructions have been included.

3. The object programs produced by the FORTRAN Assembler are relocatable; the programs are loaded into an area of memory determined by the position of other programs at load time.

4. DDT cannot be used with relocatable programs since symbol definitions are not established until loading.

5. START and PAUSE can be used as previously described. However, execution of a program assembled and loaded by the FORTRAN system can only be accomplished by placing the starting address (usually 22) in the ADS and depressing START.

NOTE: To avoid improper loading, all absolute parameter assignments should precede any references to them in the program.

The following definitions are used in this appendix:

main program	A program which is terminated by START or PAUSE and may contain only EXTERNAL lists.
subroutine	A program which contains INTERNAL symbols and is terminated by START or PAUSE. An EXTERNAL list can be included also.
library routine	A program in library format; that is, a subroutine terminated by a LIBFRM list.

The linking pseudo-instruction is a convenient method for a program to call subroutines and library routines. Library routines may be a group of often-used functions or subroutines which are assembled in library format (LIBFRM used). After loading a main program and subroutines, any number of library format routines can be read in. Only those routines called by previous programs will be loaded. By placing 5 in the address switches and depressing START, the user can obtain the locations of INTERNAL symbols and check to see that all referenced routines have been loaded (see Loading A Relocatable Program). Again, the loader handles the linking of programs so that assembly can take place separately and symbol punches are not required.

Since programs are packed consecutively, memory space is also conserved by using relocatably assembled object programs.

The major disadvantage of relocatable assembly is that DDT cannot interpret the symbols produced. Methods of surmounting this problem will be discussed under Debugging A Relocatable Program.

THE LINKING PSEUDO-INSTRUCTIONS

EXTERNAL SYM1, SYM2,....

An external list of subroutine or library routine symbols must precede any reference to the symbols. Symbols are not allowed to appear inmore than one EXTERNAL list. The list is in the same format as a PUNDEF list (see page 26). Of the three linking pseudo-instructions, only EXTERNAL can be used in a main program.

INTERNAL SYM1

Internal is used only in subroutines and library routines. It is followed by one symbol and immediately precedes the comma definition of that symbol. INTERNAL forms the other half of the link established by EXTERNAL. Using INTERNAL also causes the automatic punching of a secondary entry to the Linking Loader. This loader must already be in memory when a subroutine is loaded. In general, the Linking Loader is punched on main programs only; it is always suppressed if INTERNAL occurs.

LIBFRM SYM1, SYM2,....

Any program with INTERNAL symbols may be assembled in library format by replacing (or preceding) START or PAUSE with LIBFRM. When a LIBFRM list is encountered, the FORTRAN Assembler punches out a library format tape immediately with no operator action required. Symbols are not printed. Library format is essentially the format generated by using INTERNAL, but preceded by a heading block containing all INTERNAL symbols as specified in the LIBFRM list.

Another way to obtain a library format assembly is to prepare a tape consisting of a dummy title and a LIBFRM list. Assembling the subroutine and LIBFRM tape as a 2-tape program produces a library format tape.

Example:

Five programs are to be assembled relocatably:

1. A MAIN PROGRAM containing

```
EXTERNAL SR1X, SR2X, LR1X, LR2X
        .
    JMS SR1X
        .
    JMS LR2X
        .
    JMS LR1X
        .
    CAL SR2X
        .
START GO
```

2. A subroutine SR2X containing

```
INTERNAL SR2X
SR2X,       ...
              .
EXTERNAL SR1X, LR2X
              .
        JMS SR1X
              .
        JMS LR2X
START
```

3. A subroutine SR1X containing

```
internal SR1X
SR1X,       ...
              .
START
```

4. A library routine LR1X containing

```
INTERNAL LR1X
LR1X        ...
              .
EXTERNAL LR2X
              .
        JMS LR2X
              .
LIBFRM LR1X
```

5. A library routine LR2X containing

```
INTERNAL LR2X
LR2X,       ...
              .
              .
LIBFRM LR2X
```

After assembly the main program must be loaded first, placing the Linking Loader in memory.

The subroutines can be loaded in any order following the main porgram.

An EXTERNAL call for a symbol must have been encountered by the loader before the library routine containing that symbol as an INTERNAL symbol will be loaded. Consequently to avoid repeated readins of routines, the library routines called by subroutines or other library routines should follow them in the loading sequence.

When an INTERNAL symbol is encountered by the Linking Loader, its definition is saved. The definition then replaces the memory reference in an EXTERNAL call. Thus, the main program and associated routines are joined together when loaded by the Linking Loader.

FORTRAN ASSEMBLER ERROR PRINTOUTS

The error printouts produced by the FORTRAN Assembler are identical to those produced by the PDP-7 Assembler with the addition of:

SYS	Internal symbol previously defined or incremented in this definition.
IFZ	More than one symbol in an internal symbol definition.
IFY	Internal symbol not defined by comma.
IFX	External symbol referenced before the external declaration occurred or external symbol already defined.
IFQ	Illegal format in library list.
LIQ	Illegal term punctuation in library list.

RELOCATABLE OUTPUT

When loaded by the Linking Loader, relocatable programs are placed in consecutive memory locations beginning at 22. This is also the first address of the main program. It must be loaded first since the loader is punched only on its tape.

Absolute program segments or routines are placed in memory where specified by their absolute address assignment. Care should be taken to insure that relocatable programs and absolute programs are not loaded over each other. To facilitate the detection of overlays, at the end of any stage of the loading process, the loader will display in the ACCUMULATOR lights the value of the next sequential memory location following the register just loaded.

In the loading procedure, constants are stored following the last location in the last program loaded. If an absolute segment is loaded last, all the constants will be stored following that segment. If this is undesirable, the absolute segment could be preceded by a symbolic address tag assignment and followed by the tag used as an address assignment (slash). Constants would then be stored following the location of the tag.

DEBUGGING A RELOCATABLE PROGRAM

A relocatable program cannot be debugged with DDT. If problems are encountered, the best solution is to assemble the main program and called routines together absolutely using the PDP-7 Assembler. DDT can be used to debug the absolute program; then assemble the corrected program relocatably.

No changes need be made in the relocatable source program to do this. The PDP-7 Assembler ignores the three linking pseudo-instructions. However if a tape has been prepared using LIBFRM, it should be followed with a START. When read by the FORTRAN Assembler, the START is not encountered; when read by the PDP-7 Assembler, the LIBFRM is ignored and the START interpreted as usual. If this format is used, the routine is compatible with either Assembler.

LOADING THE RELOCATABLE OBJECT PROGRAM

1. Load the main program, with the Linking Loader punched on the tape, through RIM (ADS=17770, depress START).

2. Load subroutines through RIM which are called by other programs.

3. Load library routines by depressing START with ADS=6. Any number may be assembled together and read in from one tape. Only those routines called are loaded into memory. If a library routine calls other library routines, it should precede them in the loading sequence.

4. To obtain a printout of the locations of subroutine and library routine symbols and to find if all called routines have been loaded, depress START with ADS=5. If a routine has been called but not loaded, its symbol is printed preceded by a minus sign. The address of the first reference to this symbol is also printed. If further routines are needed, they should be loaded as in Steps 2 and 3 above.

5. To execute the relocatable program in memory, depress START with ADS=22.

APPENDIX 4

CHARACTER SETS

	ASCII	FIODEC	Teletype (Baudot)
†	A-Z	A-Z	A-Z
†	0-9	0-9	0-9
†	!	V	!
	"	"	"
†	#	¯ (overbar)	#
†	$	_ (underbar)	$
	%	⊃	no equivalent
†	&	∧	&
	'	'	'
†	(	(	(
†	)	)	)
	*	x	n.e.
†	+	+	n.e.
†	,	,	,
†	-	-	-
	.	.	.
†	/	/	/.
	:	: (center dot, period)	:
	;	; (center dot, comma)	;
	<	<	n.e.
†	=	=	n.e.
	>	>	n.e.
	?	?	?
	@	→	n.e.
	[	[	n.e.
	\	|	n.e.
	]	]	n.e.
	↑	↑	n.e.
	←	~	n.e.
†→|	tab	tab	|→ bell
† ↓	line-feed	(included in ↓)	↓ line-feed
† ↓)	carriage return	↓↓) carriage return	↓) carriage return
†	space	space	space
	rub-out	n.e.	n.e.
	blank	n.e.	n.e.
†	form feed	stop code	n.e.

† designates basic character set