G. Buc

env of Digital Equipment Cor oration, the She propression of Digital Equipment Cor oration and si all not be the trains for the manufacture or sale of items without within permission.

CONFIDENTIAL

PDP-X Technical Memorandum # 43

TITLE:

PDP-X Assembler Specifications

AUTHOR(S):

H. G. Bramson

INDEX KEYS:

Software Specifications

XAP

DISTRIBUTION KEY:

B, C

OBSOLETE:

PDP-X Technical Memorandum # 35

REVISION:

None

DATE:

February 1, 1968

PROGRAM SPECIFICATION

PDP-6/PDP-X ASSEMBLER
XAP-6

0.1 OVERALL DESCRIPTION

XAP-6 is the symbolic assembly program for assembling PDP-X programs on the PDP-6. XAP-6 runs under control of the PDP-6 Time-Sharing Monitor. XAP-6 processes input source programs in two passes and requires a minimum of 4K of core memory. Additional core is dynamically added if required. It is completely device independent, allowing the user to select standard peripheral devices for input and output files via a command string.

The normal output of the assembler is a binary object program which can be loaded for debugging or execution by the PDP-X Simulator on the PDP-9 (XSIM-9). XAP-6 prepares the object program in either relocatable binary or non-relocatable binary.

An output listing showing both the programmer's source coding and the object code produced by the assembler is printed if desired.

I. GENERAL SPECIFICATION

1.1 MACHINE REQUIREMENTS

XAP-6 can operate on a 16K PDP-6 under control of the PDP-6 Time-Sharing Monitor system. The minimum peripheral requirement for the normal operation of the assembler is:

> paper tape reader paper tape punch console teletype

1.2 MACHINE OPTIONS

The assembler is device independent and therefore other devices contained in the machine configuration may be selected via the command string. Such devices might include:

Disc DECtape Magnetic Tape Card Reader Line Printer

1.3 SYSTEM REQUIREMENTS

XAP-6 requires the presence of the PDP-6 Time-Sharing Monitor System. This monitor controls all of the input/output activities that may be required by XAP-6.

1.4 RESIDENT PROGRAMS

NOT APPLICABLE.

0.1 OVERALL DESCRIPTION

XAP-6 is the symbolic assembly program for assembling PDP-X programs on the PDP-6. XAP-6 runs under control of the PDP-6 Time-Sharing Monitor. XAP-6 processes input source programs in two passes and requires a minimum of 4K of core memory. Additional core is dynamically added if required. It is completely device independent, allowing the user to select standard peripheral devices for input and output files via a command string.

The normal output of the assembler is a binary object program which can be loaded for debugging or execution by the PDP-X Simulator on the PDP-9 (XSIM-9). XAP-6 prepares the object program in either relocatable binary or non-relocatable binary.

An output listing showing both the programmer's source coding and the object code produced by the assembler is printed if desired.

1. GENERAL SPECIFICATION

1.1 MACHINE REQUIREMENTS

XAP-6 can operate on a 16K PDP-6 under control of the PDP-6 Time-Sharing Monitor system. The minimum peripheral requirement for the normal operation of the assembler is:

paper tape reader paper tape punch console teletype

1.2 MACHINE OPTIONS

The assembler is device independent and therefore other devices contained in the machine configuration may be selected via the command string. Such devices might include:

Disc DECtape Magnetic Tape Card Reader Line Printer

1.3 SYSTEM REQUIREMENTS

XAP-6 requires the presence of the PDP-6 Time-Sharing Monitor System. This monitor controls all of the input/output activities that may be required by XAP-6.

1.4 RESIDENT PROGRAMS

NOT APPLICABLE.

2. DESIGN SPECIFICATIONS

2.1 DESIGN GOALS

XAP-6 is intended to be downward compatible with the eventual PDP-X assembler (XAP), i.e., XAP-6 will not contain features which are unavailable in XAP, and also source programs will be 100% language compatible. In order to facilitate changes and enhance maintainability, XAP-6 will be written in a highly modular form.

The problems of reimplimentation of XAP-6 to XAP will be minimized because the internal structure of the two assemblers will basically be the same.

2.2 INPUT

2.2.1 INPUT FORMAT

The input format for XAP-6 is equivalent to the PDP-X Assembler Language.

The remainder of this section is presented in the form of a Language manual.

PDP-X

ASSEMBLER

(XAP)

USER'S MANUAL

PDP-X Assembly Program

0.0 INTRODUCTION

XAP is the symbolic assembly program for the PDP-X. Operating under control of a monitor, which handles I/O functions, XAP processes input source programs in two passes, and requires less than 6K of core memory. It is completely device independent, allowing the user to select standard peripheral devices for input and output files.

XAP makes machine language programming on the PDP-X much easier, faster and more efficient. It permits the programmer to use mnemonic symbols to represent machine operation codes, location and numeric quantities. By using symbols to identify instructions and data in his program, the programmer can easily refer to any point in his program without knowing actual machine locations.

The normal output of the assembler is a relocatable binary object program which can be loaded for debugging or execution by the Linking Loader.

XAP prepares the object program for relocation, and the Linking Loader sets up linkages to external subroutines. Optionally, the binary program may be outputted in non-relocatable code.

The programmer may direct the assembler's processing by the usage of pseudo-operation instructions (pseudo-ops). These pseudo ops are used to set the radix for numerical interpretation, to reserve blocks of storage location, to handle strings of ASCII text, to conditionally assemble certain portions of coding and other functions which will be explained in detail.

An output listing, showing both the programmer's source coding and the object coding produced by XAP, is printed if desired. This listing may include all the symbols used by the programmer with their assigned values. If assembly errors are detected, erroneous lines are marked with specific letter error codes.

Operating procedures for XAP may be found in the appendices of this specification.

1.0 GENERAL SPECIFICATION

1.1 MACHINE REQUIREMENTS

 $\sf XAP$ operates in PDP-X systems with the I/O Monitor and the following minimum hardware configuration:

8K core memory Console teletype Paper tape reader and paper tape punch The assembler is actually device independent. The user preselects device assignments for source program input, output of the binary object program, and output of the printed listing.

1.2 MACHINE OPTIONS

With the addition of bulk storage to the hardware configuration, XAP operates with the Keyboard Monitor, which allows the user flexibility in assigning I/O devices at assembly time.

2.0 DESIGN SPECIFICATIONS

The assembler processes in two passes; that is, it passes over the same source program twice, outputting the object code (and producing a printed listing, if requested), during the second pass.

The two passes are resident in memory at the same time. Pass 1 and Pass 2 are almost identical in their operations, but object code is produced only during Pass 2. The main function of Pass 1 is to resolve locations that are to be assigned to symbols and to build up a symbol table. Pass 2 uses the information computed by Pass 1 (and left in memory) to produce the final output.

The standard object code produced by XAP is in a relocatable format which is acceptable to the PDP-X Linking Loader. Relocatable programs that are assembled separately and use identical global* symbols where applicable, can be combined by the Linking Loader into an executable object program.

Some of the advantages of having programs in relocatable format are as follows:

- a. Reassembly of one program, which at object time was combined with other programs, does not necessitate a reassembly of the entire system.
- b. Library routines (in relocatable object code) can be requested from the system device or user library device.
- c. Only global symbol definitions must be unique in a group of programs that operate together.

2.1 INPUT

XAP programs are normally prepared on a teletype as a sequence of statements. (With the aid of an editing program the program can easily be updated.) Each statement is written on a single line and is terminated by a carriage return - line feed sequence (indicated by in this document). A line feed or form feed are actually the characters that terminate XAP statements. XAP statements are virtually format free, i.e., elements

^{*} Symbols which are referenced in one program and defined in another.

of the statement are not placed in numbered columns with rigidly controlled spacing between elements. The character set that is used as input to XAP is 7-bit ASCII. (See Appendix A.)

2.2 ELEMENTS OF A STATEMENT

There are four elements in a XAP statement which are separated by specific characters. These elements are identified by the order of their appearance in the statement, and by the delimiting character which follows or precedes the element.

Statements are written in the general form:

LABEL:

OPERATOR

OPERAND, OPERAND

;COMMENTS &

The assembler interprets and processes the statements, generating one or more binary instructions or data words, or performing an assembly process. A statement must contain at least one of these elements, but it may contain all four.

2.2.1 LABELS

A label is the symbolic name created by the user to identify the statement. If present, the label is written first in a statement, and is terminated by a colon (:). No spaces are allowed between the last character of the label and the colon (:).

Examples:

ABC:

TAG:

TAG1: TAG2: TAG3:

All 3 labels point to the same location.

LABEL A.

TAGIT :

Illegal, no spaces are allowed within the label for between the last character and the colon.

Labels are not redefinable by another label, direct assignment or EOPDEF. They can not appear after an operator or operand has preceded it on a line.

Examples:

LABEL:

LDA 4,5

EOPDEF

LABEL, 10

LABEL= 1Ø

;EOPDEF can not redefine a label. (See 3.1)

;Direct assignment can not redefine a label.

(See 2.3.1)

LDA 4,5 LABEL:

;A label can not be preceded by an operator

or operand

2.2.2 OPERATORS

An operator may be:

- a. Any one of the mnemonic machine instruction codes (see appendix B).
- b. An assembler pseudo op, which directs assembler processing.
- c. EOPDEF.

If there is no label associated with the statement the operator may appear as the first element of the statement. The operator field is terminated by any one of the following delimiters:

```
1) △ (space)
2) → (tab)
3) ; (semicolon)
4) ⟨ (line feed)
```

Examples of Operators:

```
LDA ;mnemonic machine instruction.

LOC ;an assembler pseudo op.

EOP ;legal only if defined via EOPDEF.
```

In order for a symbol to be interpreted as an operator it must not be part of an expression. It must be used as a free standing symbol. If it is used in an expression, it will be treated as an operand and must therefore be user defined.

Examples

```
EOPDEF EOP, 25 ;defines EOP as a user defined operator
LDA 3, LOC ;"LDA" is an operator, "LOC" is an operand
LOC 50 ;"LOC" is a pseudo op
LOC+50 ;"LOC" is an operand
EOP TAG ;"EOP" is an operator, "TAG" is an operand
EOP+1 ;"EOP" treated as an operand
```

As an operator, a mnemonic machine instruction or a pseudo op takes precedence over identically named user symbols.

Example

Source	Would Assemble As:	
LDA=5		
LDA 4,LDA +LDA	Ø,4,Ø,5 🧎 see 6 ØØØØØ5 🐧	s.ø

2.2.3 OPERANDS

Operands are usually the symbolic addresses of the data to be accessed when an instruction is executed, or the input data or arguments of a pseudo op. In each case, the interpretation of operands in a statement depends upon the statement operator. Operands are separated by commas (if operator requires more than one operand) and terminated by a space (\triangle) , tab (-), semicolon (;), or line feed (\bigvee) .

RULES ABOUT COMMAS

A comma terminates the current operand/argument and implies that another operand/argument follows. Spaces and tabs may precede and follow commas.

Examples:

		· · · · · · · · · · · · · · · · · · ·	
Statement	Operand 1 value	Operand 2 value	Operand 3 value
 1 1, , 1,2 1,2 1,2] 0 0 1 1 0	• Ø Ø Ø 2 2 2 2 2 2	
1, 2,	Ø 1 1	Ø 2 2	ø ø

flagged as an error

Symbols used as operands must have a value defined by the user. If a symbol, used as an operand, is the same as a mnemonic machine instruction or pseudo op, it will not be interpreted as such, but rather as a user defined symbol. EOPDEF defined symbols may be used as operators or operands.

Examples

```
LOC=5
STA: 1
LDA 4,STA ;STA is user defined
STA 4,LOC ;LOC is user defined, STA is a machine op code
LOC 5 ;LOC is a pseudo op
```

Many instructions reference an accumulator and a memory location. If the first operand is an accumulator it must be terminated with a comma (,). If an accumulator is not specified but the operator requires one, accumulator 4 is assumed and the instruction will be flagged. The value of the accumulator is truncated to the 3 least significant bits.

If an accumulator is specified on an instruction that does not require one, it will be flagged as an error. Any reference to accumulator 1 will get flagged as an error, because AC 1 is the hardware program counter and must not be referenced. (See Appendix E for expected formats)

Examples:

```
AC5 = 5
       LDA
              TAG
                          ;error, AC 4 assumed
       LDA
              ,TAG
                         ;AC Ø implied
       LDA'
              AC5,4
                         ;AC 5 referenced
                         ;error, AC 1 can not be referenced
       LDA
              1,TAG
       LDA
              25, TAG
                         ;AC 5 referenced and flagged
              LOC
                         ;correct form
              AC 5, LOC ; error, instruction does not require an AC-5 assembled
```

2.2.4 COMMENTS

The programmer may add comments to a statement. Such comments must be preceded by a semicolon (;). Comments do not affect the assembly process, but are used mainly for documentary purposes.

Examples:

```
this is a comment; this also is a comment; this also is a comment;
```

2.3 SYMBOLS

The programmer creates symbols for use in statements to represent labels, operators and operands. A symbol contains one to six characters from the following set:

The letters A-Z
The digits Ø-9

Two special characters \$ (dollar sign)

%(percent)

The first character of a symbol must be a letter or dollar sign or percent. It must not be a digit.

The following symbols are legal:

A \$%TAG \$ A% TAG25 P9% %TAG % \$25

The following symbols are illegal:

8TAG First character may not be a digit TAG?1 ? is an illegal character in a symbol

Only the first six characters of a symbol are meaningful to the assembler, but the programmer may use more for his own information. If he writes,

SYMBOL1: SYMBOL2: SYMBOL3:

as the symbolic labels on three different statements in his program, the assembler will recognize only SYMBOL and indicate error flags on the statements containing SYMBOL1, SYMBOL2, and SYMBOL3, because to the assembler they are duplicates of SYMBOL.

2.3.1 DIRECT ASSIGNMENTS

The programmer may define a symbol directly into the symbol table by means of a direct assignment statement, written in the form

where value can be any statement, including literals and data generating pseudo ops. The expression to the right of the = assumes the operator field. The value of the direct assignment can not generate more than one word.

Direct assignments are redefinable. They may only redefine other direct assignments. They may not redefine user symbols.

Examples:

```
A=1
B=A+3; B is defined as A+3=4
A=2
A=A+1; redefinition of A
GETS=LDA 4,5
LIT1=[1]; LIT1 is address of Literal
MSG=ASCII/AB/; Stored as Ø411Ø1
CHK=CMP 4, ['C']; Flagged as an error, value more than 1 word.
```

The = sign must immediately follow the symbol. However, the value to the right of the equal sign may have preceding spaces or tabs.

Examples:

Legal			Illegal
$A = 5$ $B = \triangle 100$ $C = -101 200$	• .	•	A ->1=5 B∆=∆1Ø

Direct assignment statements do not generate instructions or data in the object program. They are used to assign value so that symbols can be conveniently used in other statements.

In general, it is good programming practice to define symbols before using them in statements which generate storage words.

Example:

$$Z=5$$

 $X=Z$
 $Y=X$
LDA 4,Y ;same as LDA 4,5

A symbol may be defined after use.

Example:

BR Y Y=1

This is called a forward reference, and is resolved properly in Pass 2. When first encountered in Pass 1, the BR Y statement is incomplete because Y is not yet defined. Later in Pass 1, Y is given the value 1. In Pass 2, the assembler finds that Y = 1 in the symbol table, and forms the complete word.

Since the assembler operates in two passes, only one-step forward references are allowed. The following sequence would be illegal.

BR Y Y = Z Z = 1

The assembler will list, during Pass 1, direct assignments whose values are incomplete. In pass 2, statements containing references to unresolved direct assignments will be flagged with an "E".

2.4 NUMBERS

Numbers used in source programs may be signed or unsigned integers in single or double precision, or they may be floating point numbers. Negative numbers are represented in twos complement. The numbers are interpreted by the assembler according to the radix specified by the programmer, where;

 $2 \le \text{radix} \le 10$.

The programmer may use an assembler pseudo op, RADIX (see 3.12), to set the radix for all numerical interpretation. If the RADIX pseudo op is not used, the assembler assumes a radix of 8 (octal).

The radix may be changed locally, to decimal, for a single number by following the number with a period (.). If the period is followed by a digit, the number will be interpreted as a floating point number.

SOURCE STATEMENT	GENERATED VALUE (OCTAL)	RADIX IN EFFECT
#4. ************************************		£
256	ØØØ256	OCTAL
+135	øøø135	OCTAL
- 75	1777ø3	OCTAL (Twos complement)
1ØØ.	ØØØ1 44	DECIMAL
-4ø.	1 7 773Ø .	DECIMAL (Twos complement)

2.4.2 DOUBLE PRECISION INTEGERS

Double precision integers are specified by the letter D terminating the number which indicates that they will occupy two memory locations with the least significant digits right justified. As with single precision integers, a negative double precision integer will be represented in two complement form.

Examples:

SOURCE STATEMENT	GENERATED VALUE (OCTAL)	RADIX IN EFFECT
+125 D 63572643 D -735 D -100 . D -63572643 D	000000 000125 000316 172643 177777 177043 177777 177634 177461 005135	OCTAL OCTAL OCTAL (Twos complement) DECIMAL (Twos complement) OCTAL (Twos complement)

2.4.3 BINARY SHIFTING

An integer or symbol value may be logically shifted left or right by following it with the character pound sign ($^{\#}$) followed by a number, \pm nn, which represents the number of places to be shifted. "nn" always represents a decimal number from Ø-31. If "nn" is positive, the value will be shifted left. If "nn" is negative, the value will be shifted right. Bits leaving one end are lost and zeros enter at the other end. Shifting may only be used with integers.

Examples:

	SOURCE		GENERATED VALUE
A=25			
	A [#] 5		ØØØ124
	125 [#] 12	•	ø5øøø
	-35. [#] -3		Ø17773
	72345635D#+2		ØØ1647 027164

2.4.4 FLOATING POINT NUMBERS

If a string of digits contains a decimal point, which is followed by a digit, it is evaluated as a floating point DECIMAL number.

Examples:

SOURCE STATEMENT		GENER. VAL	
+.19		040060	050754
-183.72 +23.279	-		050754 043554

Floating point decimal numbers may also be written, as in FORTRAN, with the number followed by a signed or unsigned exponent which represents a power of 10. The exponent will be treated as a decimal number.

Examples:

SOURCE STATEMENT	GENERATED - VALUE	
* 1.5E5	042444	117400
1.5E+2	041226	000000
1.5E-3	037142	046722

The preceding form of a floating point decimal number represents single precision, in that it causes two words to be generated.

To express a double precision floating point decimal number, the number is followed by the letter D. In addition, an exponent can be represented by following D with a signed or unsigned number which represents a power of $1\emptyset$.

Examples:

SOURCE	GE	NERATE	D	
STATEMENT	VALUE			
36D-6	137623	072274	065176	174733

2.5 EXPRESSIONS

Expressions are strings of symbols and numbers separated by arithmetic or boolean operators.

The following are the allowable operators.

0	PERATOR	FUNCTION
+	(plus)	add
_	(minus)	subtract
*	(asterisk)	multiply
/	(slash)	divide
&	(ampersand)	AND
	(exclamation)	inclusive OR Boolean
\	(back slash)	exclusive OR

The assembler computes the 16 bit value of the series of numbers and/or symbols connected by the arithmetic and boolean operators, truncating from the left, if necessary. Operations are performed from left to right (i.e., in the order in which they are encountered). For example: A+B*C+D/E-F*G is equivalent to the following algebraic expression:

$$(((A+B)*C+D)/E-F)*G$$

Examples:

Assume the following symbol values:

SYMBOL		VALUE (OCTAL)
Α		2
В	,	10
C		3
D	•	5

The following expressions would be evaluated occording to the above rule.

EXPRESSION	EVALUATION	N (OCTAL)
A/B+A*C B/A-2*A-5	000006 177777	Remainder of A/B is lost
C+A&D	000005	(-1)
A+B*C&D	000004	
1+A&C	000003	
,50B	000052	

2.6 LOCATION ASSIGNMENTS

As source program statements are processed, the Assembler assigns consecutive memory locations to the storage words of the object program. This is done by reference to the Location Counter, which is initially set to zero. Machine instructions may cause the Location Counter to be incremented by either one or two. Other statements such as those used to enter data or text, or to reserve blocks of storage words, cause the Location Counter to be incremented by the number of storage words generated.

2.6.1 SETTING AND REFERENCING THE LOCATION COUNTER

The programmer may set the Location Counter by using the pseudo op LOC which will be described later on. He may reference the Location Counter directly by using the symbol, period (.).

Consider the following example:

LOCATION COUNTER	STATEM	ENT	FORM	
100	BR	.+5	SHORT (1 word)	
101	LDAL	4,5000	LONG (2 words)	
103	STAL	4,6000	LONG	
105	LDA	5,20	SHORT	

The first statement, BR .+5, refers to 5 locations away from the current instruction and therefore references location 105. If the BR .+5 instruction was long form it would have to be .+6 to provide the same results.

2.6.2 INDIRECT ADDRESSING

The character @ prefixing an operand causes the assembler to set bit \emptyset , indicating indirect addressing.

If the statement contains both an operator and an operand, two words will be generated for the statement. If there is only an operand, only one word will be generated for the statement.

Examples:

STATEMENT	GENERATED VALUES	NOTES
LDA=50 TAG=20 LDAL 4,@TAG LDAL 4,@LDA	Ø,4,Ø,2ØØ 100020 Ø,4,Ø,2ØØ 100050	Assignments, no values generated Two words generated
€TAG.	100030	One word generated, also forces operand and therefore LDA interpreted as operand rather than operator.

2.6.3 INDEXING

If the programmer wishes to index an operand of a statement he may do so by enclosing a value or expression within parenthesis suffixed to the operand.

Examples:

X2=2 X3=3 LDA 4,TAG(X2) BR Ø(X3) or BR (X3) STA 5,TAG(2) LDA 4,@TAG(X2) ;indexed, indirect

The left parenthesis" (" is mandatory, while the right parenthesis")" is optional. In addition, the following statements will produce the same results without causing an error.

 (\emptyset) (\emptyset) will all produce an index value equal to \emptyset

2.6.4 LITERALS

In assembler statements, a symbolic data reference may be replaced by a direct representation of the data enclosed within brackets ([]). (The left bracket ([]) is mandatory, while the right bracket ([]) is optional). This direct representation is called a literal. The literal may be any legal assembly statement including data generating pseudo ops. The literal can not generate more than 1 word. Literals may not be nested, i.e., a literal can not reference another literal.

Examples:

LDA 4,[1]
LDA 4,[2*A-1]
LDA 4,[LDA 4,TAG]
SHFT 4,[SHFTL+7]
LDA 4,[ASCII /ABCD/] ;error, too

;error, too many words generated

2.6.4.1 ALLOCATION OF LITERALS

Literals are either pooled or cause the immediate mode to be generated depending upon the form of the referencing statement.

2.6.4.1.1 LONG FORM REFERENCES

Literals referenced on long form instructions cause the immediate mode to be generated for the referencing instruction with the value of the literal as the immediate word.

Examples:

sou	RCE		GENERATED VALUE
TAG TAG		#	
	LDAL	4,[TAG+3]	Ø,4,1,2ØØ
	CMP	5,[TAG1]	ØØØØ1Ø 6,5,1,111 ØØØØØ3
	СМР	4,[1Ø]	6,4,1,111 ØØØØ1Ø
			υκικονίας συ

2.6.4.1.2 SHORT FORM REFERENCES

Literals referenced on short form instruction (or if no operator is present on the referencing line) are pooled on the occurrence of an LPOOL pseudo op (see 3.13) or the END pseudo op (see 3.11.2).

Duplicate literals, completely defined when encountered in the source during pass 1, are stored only once so that many references to the same literal in a given "pool area" result in only one (1) memory location being allocated for the literal.

Examples:

	PROGRAM	GENERATED	SOURCE
	COUNTER	CODE	CODE
•		CARROLL SELVER TO CARROLL SERVE SELVER SELVE	
			A=77
	2 ØØ		LOC 200
	2ØØ	Ø,4,Ø,2Ø7	LDA 4,[TAG]
	2Ø1	Ø, 4, Ø, 205	LDA 4,[77]
	2ø2	Ø, 4, Ø, 2Ø6	LDA 4,[2ØØ]
	2ø3	Ø, 4, Ø, 2Ø5	LDA 4,[A]
	2Ø4	Ø, 4, Ø, 2Ø7	LDA 4,[TAG]
		ρ, ., ρ, 2γ.	LPOOL
	2Ø5	ØØØØ77	
	2ø6	ø øø2øø	
	2Ø7	ØØØ211	
	•	, , ,	occurs "TAG" not difficulty
		ra location allocated in pass 1 be	
	211	Ø, 4, Ø, 216	TAG: LDA 4,[1]
	212	ØØØ216	[1]

PROGRAM COUNTER	GENERATED CODE	SOURCE CODE
213 214 215	ØØØ217 Ø,4,Ø,22Ø ØØØ22Ø	[77] LDA 4,[TAG] [TAG] END
216 217 22Ø	000001 000077 ØØØ211	

2.7 INSTRUCTION FORMS

The permanent symbol table of the assembler contains the machine opcode set of the PDP-X. This is comprised of the basic opcodes, extended opcodes and the I/O opcodes. In addition, the basic opcode mnemonics have been extended, by suffixing them with the letter "L", in order to have them be assembled as long form instructions.

For example;

LDA 4,5 would occupy 1 memory location, while LDAL 4,5 would occupy 2 memory locations.

The user must make sure that long form be specified whenever an indirect reference is made or whenever a reference to an EXTERNAL is made.

Examples:

EXTERNAL EXTSMB (see 3.9.2)
LDAL 4,@5
BSRL EXTSMB
LDA 4,@6 ;would be flagged as an error

2.7.1 EFFECT OF INDEX VALUES ON INSTRUCTIONS

Index values cause the assembler to generate specific forms of addressing. The index values and their affect on short form and long form instructions are as follows:

2.7.1.2 SHORT FORM INDEX VALUES

INDEX		
VALUE	MEANING	COMMENTS
NONE	Direct or relative whichever is possible	 Direct is legal only if operand is absolute and ≤ 377 and ≠200. Relative is legal only if the location and operand have same form (abs/abs - rel/rel) and difference between operand and program counter is within ±0-177.
Ø	Direct	Legal only if operand is absolute and ≤ 377 and ≠200.
1	Relative	Legal only if the location and operand have same form (abs/abs - rel/rel) and difference between operand and program counter is within ±0-177.
2	Indexed with 2	Legal only if operand is absolute and is within ±Ø-177.
3	Indexed with 3	

With respect to the nature of the form of the location and its operand, the following table shows the legal forms. If a form is specified that is illegal, only 1 location will be allocated for it and it will be flagged with an "A".

LOCATION	OPERAND	LEGAL SHORT FORMS
RELOCATABLE	RELOCATABLE	RELATIVE ONLY
RELOCATABLE	ABSOLUTE	direct only
ABSOLUTE	RELOCATABLE	must be long form
ABSOLUTE	ABSOLUTE	DIRECT OR RELATIVE

Whenever the value of a literal or direct assignment contains relative addressing, i.e., X=1, it is computed relative to the program counter where it was encountered and flagged as a possible error.

Example:		
ASSUMED LOCATION COUNTER	GENERATED VALUE	STATEMENT
57Ø	Ø,4,1,Ø1Ø	B=6,000 A=LDA 4,B;

;flagged as possible error because the relativity may be incorrect.

2.7.1.3 LONG FORM INDEX VALUES

Index values for long form instructions do not take on any special meaning for long form instructions other than what they normally would mean. There are no restrictions as to their usage.

The state of the s	
INDEX_VALUE	EFFECTIVE ADDRESS
NONE Ø	D2
1	PC+1 (IMMEDIATE)
2	D2+X2
3	D2+X3

2.7.2 ADDITIONAL NOTES ON LONG FORM INSTRUCTIONS

If a long form instruction is immediately suffixed with an up arrow (1), it indicates to the assembler to output only 1 word (16 bits) and update the program counter by 1 rather than by 2.

Examples:

		Generates
CMP↑	4,(1)	6,4,1,111
CMP1	4,5(2)	6,4,2,111

2.7.2.1 CALCULATION OF D2 FOR LONG FORM INSTRUCTIONS

IF the index value =1, D2 is a 16 bit value. IF the index value ≠1, D2 is a 15 bit value and if indirect was specified, bit Ø is set.

Examples:

A = -1	Generates	
CMP 4,[A]	6,4,1,111	177777
SUB 4,A	6,4,Ø,112	Ø77777
SUB 4,@5	6,4,Ø,112	100005

3.0 · PSEUDO OPS

Pseudo ops are statements which direct the assembler to perform certain assembler processing operations, such as producing text strings or reserving blocks

of memory. Some pseudo ops generate object code and some do not. In all cases a pseudo op will only be interpreted as such only if it is an operator.

3.1 EOPDEF

The programmer can define his own extended op code operator using an EOPDEF statement written in the following general form:

EOPDEF NAME, DI, R

where: 1. "NAME" is the name of the user defined operator.

2. "D1" is the value to be assigned to the D1 portion of the instruction when the EOPDEF operator is referenced.

3. "R" is the value to be assigned to the R portion of the instruction

DI and R may be symbols, numbers or expressions. In addition, an opcode of 6 will be generated when the EOPDEF operator is referenced.

EOPDEF is the method for the user to define unused extended opcodes (UUO's).

When an EOPDEF is used, it must follow the same syntax rules as extended op codes (see appendix E).

Examples:

EOPDEF Eopdef	A,12Ø B,121,2		
A 6,5 A 5 B 5 B 4,5	generates generates generates generates	OP,R,X,D1 6,6,Ø,12Ø 6,4,Ø,12Ø 6,2,Ø,121 6,4,Ø,121	ダダダダダ5 000005 and flagged (AC=4 assumed) 000005 ダダダダダ5 and flagged

EOPDEF defined symbols may be redefined by other EOPDEF's.

If an EOPDEF defined symbol is redefined by any other means, the EOPDEF definition remains and the line that attempted to do the redefinition will be flagged with an "I".

If an EOPDEF is encountered and the symbol has already been entered into the symbol table (but not as an EOPDEF), the EOPDEF is ignored and flagged with an "I".

Examples:

	eopdef A	A,13Ø TAG	
A: B:	LDA LDA	4,5 4.5	;Line flagged and label "A" ignored
	EOPDEF	B,131	;Line flagged and EOPDEF ignored

3.3 RESERVING STORAGE

3.3.1 UNDEFINED SYMBOLS

If any symbols, except EXTERNAL symbols (see 3.9.2), remain undefined at the end of Pass 1 of assembly, they are automatically defined as the addresses of successive registers following the locations reserved for literals, if any, at the end of the program.

All lines which referenced the undefined symbol will be flagged with an error code. One memory location will be reserved for each undefined symbol with the initial contents of the reserved location being unspecified.

At the end of Pass 1, the assembler will output the names of all undefined symbols and the locations allocated to them.

3.3.2 RESERVING A BLOCK OF MEMORY

The user may request the assembler to reserve a block of memory by the usage of the BLOCK pseudo op written in the following form:

BLOCK value

BLOCK reserves a block of memory equal to "value". "Value", which may be a number, symbol, or expression, must be predefined; otherwise, phase errors will occur during Pass 2 of assembly. The assembler will output a message, in Pass 1, if the value is not predefined. The initial contents of the reserved location are unspecified.

Examples:

SOURCE	LOCATIONS
STATEMENTS	RESERVED (OCTAL)
A=100	
B=200	
C: BLOCK 5	5
D: BLOCK B-A	1 <i>ø</i> Ø
BLOCK D-C+1	6

3.4 BYTE POINTERS

The LDC and STC instructions are available for byte manipulation. These instructions use the effective word as a character pointer to locate an 8 bit byte. The LBYTE and RBYTE pseudo ops are used to set up the pointer word, where LBYTE initializes to point to the left byte and RBYTE to the right byte.

Examples:

(assume BUFF to be location $3000_{\rm g}$)

LDC POINT1: LBYTE	4, POINT1 BUFF ;generates 6001 .	0 BUFF	14 15
LDC POINT2: RBYTE LDC	4, POINT2 BUFF ;generates 6000 4, [RBYTE BUFF]	0 BUFF	14 15
VFD STATEMENT			

3.5 VFD STATEMENT

To conserve memory, it is useful to store data in less than full 16 bit words. Bytes of any length, from Ø to 16 bits may be entered using the VFD statement written in the form:

The first operand n, which must be enclosed in angle brackets, is the byte size in bits. It is interpreted as a decimal number in the range of £-16. The operands following are separated by commas and are the data to be stored from left to right. If an operand is an expression, it is evaluated and if necessary truncated from the left to the byte size specified. The data may occupy only 1 word. (EXTERNAL symbols or relocatable symbols should not be used in VFD statements as results at object time may be erroneous.)

The byte size may be altered by inserting a new byte size, in angle brackets, immediately following any operand.

Examples:

	SOURCE	GENERATED	VALUE
A=500 · B=50 VFD VFD VFD VFD	<3>1,2<6>B <7>B, <9>A <3>1,2,3<7>A <3>1,2,3,4,5,6	025200 050500 024700 024712	error, too many bytes specified

3.6 TEXT HANDLING

Text handling enables the user to represent directly the 7-bit ASCII character set. The assembler will convert the desired character to its appropriate numerical equivalent. (See appendix A)

3.6.1 SINGLE WORD TEXT

The assembler translates up to 2 characters enclosed within single quote marks (') using the ASCII value of the characters and stores them with the first character in the right byte (8-15) and the second character, if any, in the left byte (\emptyset -7). Any legal ASCII characters may be used, except the single quote mark itself (').

Examples:

	•	GENERATE	D CODE	. ,	
TABLE=	Ø			•	
LDA	4,'A'+TABLE	Ø,4,Ø,1ØI			
LDAL	4, 'B' (1)	Ø,4,1,2ØØ	ØØØ1Ø2		
LDAL	4,['AB']	Ø,4,1,2ØØ	Ø411Ø1		
LDAL	4,['ABC']	Ø,4,1,2ØØ	Ø411Ø1	;error,	too many chars.
'AB'		Ø411Ø1			
VFD	$\langle \mathcal{D} \rangle \langle \mathcal{D} \rangle \langle$	Ø417Ø2			

3.6.2 MULTIPLE WORD TEXT

If the user desires more than 1 word of text, he may do so by using one of the following text pseudo ops;

1) ASCII	generate ASCII text.
2) ASCIP	generate ASCII text with odd parity in bit 8.
3) ASCIC	generate ASCII text followed with 2 words;
	WORD1 = number of bytes in text (2's complement)
	WORD2 = byte pointer to the first byte of the text
4) ASCIPC	generate ASCII text with parity, byte count and pointer.

The first non-blank or non-tab following one of the text pseudo ops will be interpreted as the text delimiter, which may be any ASCII character, except a left angle bracket (<). The text will be terminated by repeating the initial delimiter. The characters will be stored in the same manner as single word text, i.e., the first character in the right byte (8-15) and the second character in the left byte ($\emptyset-7$).

Examples:

ASCII	/THIS IS A MESSAGE/	ø2ø123	Ø51511 Ø2Ø1Ø1 Ø435Ø1	Ø42515
ASCIP	/PARITY/	14Ø72Ø	Ø44522	154524

ASCIC /COUNT + POINTER/ Ø475Ø3 Ø47125 Ø25524 (assume PC = 500) Ø4752Ø Ø47111 Ø42524 ØØØ122 byte count 177763 byte pointer ØØ12ØØ the user may reference the last 2 words in the following way; HDR = .-2LDA 4, HDR /DATA/ ASCIPC 140704 140524 (assume PC = 600) 177774 byte count byte pointer ØØ14ØØ ASCII .12/30/67. Ø31Ø61 Ø31*457* Ø2746Ø Ø33466 **ASCII** /E ND. Ø471Ø5 ØØ65Ø4 ØØØØ12

Expressions may be represented in text statements by enclosing them in angle brackets (<>). The angle brackets must appear external to the text delimiter, otherwise they will be interpreted as part of the text string. The value enclosed in angle brackets will be truncated to 7 bits.

Examples:

3.7 LOCATION DEFINING AND ADDRESS MODE

The normal output of the assembler is relocatable binary addresses. The user may also specify absolute binary addresses for the entire program or for selected portions. In addition to being able to set the address mode the user can alter the locations being assigned to instructions by explicitly defining the location.

The pseudo op LOC, control both the addressing mode and location defining.

3.7.1. LOC "n"

LOC sets the Location Counter (program counter) to "n", which must be a predefined value or expression. (The assembler will output a message, in Pass 1, if the value is not predefined).

If "n" is a relocatable expression, the assembler will assign relocatable locations for the instructions and data which follow. All labels encountered will be relocatable.

If "n" is an absolute expression, the assembler will assign absolute locations for the instructions and data which follow and all labels encountered will be absolute.

If no LOC statement appears, the assembler assumes an origin of relocatable \emptyset .

Examp	les:		•	LOCATION	TYPE
	LDA	4, A		Ø	REL
A:	LDA	4, B	•	. 1	REL
B:	LDA	4, C	• .	2	REL
X=.			-		
	LOC	200			
C:	LDA	4, D		2ØØ	ABS
D:	LDA	4, A	· •	201	ABS
•	LOC	Χ			
	LDA	4,B		3	REL

"A", "B", and "X" are relocatable, while "C" and "D" are absolute.

3.7.2 MODULAR ORIGIN

The statement MORG "n" causes the location counter to be set to the next highest multiple of "n" if it is not already at such a value. "n" is mainly useful when it is a power of 2, but it may be any value. It does not affect the existing address mode (absolute or relocatable). The assembler will output a message, in Pass 1, if "n" is not predefined.

Examples:					ATION ASS	
				TC	INSTRUCT	ION
LOC	5Ø .					
BR	4Ø				5Ø	
MORG .	2					
LDA	4,5				52	
LDA	4,5			•	53	
MORG	2		100			,
LDA	4,5	•			54	
MORG	1øø					
LDA	4,5				1øø	

The algorithm used by the assembler is as follows:

- 1. Divide current value of Location Counter by "n"
- 2. If no remainder, bypass step #3
- 3. ("n" remainder) + Location Counter ⇒ Location Counter

3.8 BINARY OUTPUT

The standard binary output of the assembler is in a format acceptable to the Linking Loader. The user may specify to the assembler to output the binary in "readin mode" by using the pseudo ops RIM of RIMNLD.

3.8.1 USAGE OF RIM AND RIMNLD

- 1. They must occur before any other statements, except the TITLE statement, otherwise they will be flagged and ignored.
- 2. All locations and operands in the program will be treated as absolute.
- 3. They force the implicit statement LOC 200 to occur.

 Note: The LOC 200 may be overridden by immediately following with a LOC statement.

3.8.2 RIM AND RIMNLD

The RIM pseudo op, in addition to specifying readin mode binary, causes the assembler to precede the binary output with a loader. If no address follows the RIM statement, the assembler assigns the program break address—as the starting address of the loader. If an address is specified, this value will be used as the starting address of the loader.

Examples:

Assume program break is 1000

RIM

- Loader address set to 1000

RIM 500

- Loader address set to 500

The RIMNLD pseudo op also specifies readin mode binary, but no loader precedes the binary output.

3.9 SUBROUTINE LINKAGE

Programs usually consist of subroutines which contain references to symbols in external programs. Since these subroutines may be assembled separately the Linking Loader must be able to identify "global" symbols.

For a given subroutine, a global symbol is either a symbol defined internally and available for reference by other subroutines, or a symbol used internally but defined in another subroutine.

Global symbols defined within a subroutine and available to others are called internal symbols. Global symbols defined by another routine and referenced by the current subroutine are called external symbols.

The linkages between internal and external symbols are set up by declaring global symbols through the INTERN and EXTERN pseudo ops.

3.9.1 INTERN \$1,52,53,...\$n

The INTERN statement defines the symbol or symbols in the string as internal to the currently being assembled subroutine and they may be referenced by other subroutines. Internal symbols may be defined in the program as either a label or direct assignment.

Examples:

INTERN

INTI, INT2

INT1: LDA 4,5

STA 4, INT2

INT2=.

3.9.2 EXTERN \$1,\$2,\$3,...\$n

The EXTERN statement defines the symbol or symbols in the string as external to the current subroutine. The symbols defined as external must not be defined in the current program.

If an external symbol is an operand on a basic op, long form must be specified.

Examples:

EXTERN

SQRT, CUBE

PUL

SQRT

PUL

CUBE

3.10 CONDITIONAL ASSEMBLY

It is often useful to have the assembler test the value of an expression and to assemble conditionally portions of the program based upon the results of the test. For this purpose, two pseudo ops are provided.

1) IF...

To initiate the condition

2) ENDC

To terminate the condition

The general form is as follows:

IF...

EXP

ENDC

The body of coding following the IF statement is assembled only if the expression "EXP" satisfies the IF condition. If not satisfied, all coding up to and including ENDC is bypassed.

The IF statements allowed are as follows:

CONDITION	ASSEMBLE IF "EXP" IS
IFLS IFN IFGE IFE IFDEF IFUND IF1 IF2	<pre> <pre> <pre> <pre> <pre></pre></pre></pre></pre></pre>
Examples:	
$A = \emptyset$ $B = 1$ $IFDEF \qquad A$ $LDA \qquad 4,5$ $STA \qquad 4,6$	these statements will be assembled

STA 4,6 **ENDC IFZER** В 4,5 these statements will not be LDA assembled, but treated as comments STA 4,6 **ENDC**

Conditional statements may be nested, that is, within the limits of a conditional statement there may be other conditional statements. Each nested conditional statement requires its own ENDC pseudo op to terminate it.

Examples:

ENDC

3.11 BEGINNING AND END OF PROGRAM STATEMENTS

3.11.1 TITLE NAME

The name appearing after the TITLE statement (up to 6 characters) will appear on the top of each page of the assembly listing. It also will be used to identify the program for DDT (debugging) and Linking Loader operations. If no TITLE statement is present, the assembler inserts the assumed name %MAIN%.

3.11.2 END START

The END statement must be the last statement in every program. A single operand may follow the END operator to specify the address of the first instruction in the program to be executed.

3.12 RADIX

When a numerical value is encountered in a statement, the assembler converts the number to a binary representation reflecting the radix indicated by the user. The statement,

RADIX n , where n is a decimal number from 2 to $1\emptyset$, sets the radix to n for all numerical values that follow, unless another RADIX statement changes the prevailing radix or a local radix change to decimal occurs.

The implicit statement, RADIX 8, begins every program and if octal numbers are desired a radix statement is not necessary.

Examples:

SOURCE	GENERATED VALUE (8)	RADIX IN EFFECT
125 RADIX 10	ØØØ125	8
1ØØ * RADIX 2	ØØØ1 44	1Ø
1Ø1111Ø RADIX 8	ØØØ136	2 · ·
175 2ØØ.	ØØØ175 ØØØ31Ø	8 1 <i>0</i>

3.13 LPOOL

LPOOL causes literals that have previously been referenced in short form instructions to be assembled, starting at the current value of the location counter.

If n literals have been defined, the next free storage location will be at program counter plus n. Literals defined after the LPOOL statement are not affected. (See 2.6.4.1.2)

3.14 PASS2

The PASS2 pseudo op causes the assembler to switch to pass 2 processing for the remaining coding. Coding preceding this statement will have been processed by Pass 1 only. The PASS2 pseudo op is primarily used for debugging, such as testing macros.

3.15 SHFTA, SHFTC, SHFTR, SHFTL

The extended op code SHFT shifts the specified accumulator as indicated by the effective word. The right half (byte) of the effective word is used as a signed shift count (+ = left, - = right) and bits 6 and 7 of the effective word indicates the type of shift to be performed.

In order to facilitate specifying the type of shift the pseudo ops SHFTA,

SHFIC, SHFIR and SHF	IL are used.	10 718 15
SHFTA ±nn generates	(arithmetic)	0 7 8 15 0 — 0 ±nn
SHFTC +nn generates	(rotate combined)	0 7 8 15 0 — 1 ±nn
SHFTR +nn generates	(rotate)	0 7 8 15 0 — 2 +nn
SHFTL ±nn generates	(logical)	0 7 8 15 0 —3 +nn

Examples:

SHFT	4, ISHFTA	+41	6,4,1,113 øøøøø4
SHFT	4, ISHFTC	-51	6,4,1,113 ØØØ773
SHFT	4, ISHFTR	<u>+</u> 71	6,4,1,113 ØØ1ØØ4
SHFT	4, ISHFTL	-41	6,5,1,113 ØØ1774

Note, that in the above examples the SHFT instructions are not indexable. Indexing the SHFT may be done as follows:

SHFT 4, TABLE(2)

4.0 RELOCATION

The normal output from the assembler is a relocatable binary program. The program may be loaded into any part of memory regardless of what locations were assigned at assembly time. To accomplish this, the address portion of some instruc-

tions must have a relocation constant added to it. This relocation constant, which is added at load time by the Linking Loader, is equal to the difference between the memory location an instruction is actually loaded into and the location that was assigned to it at assembly time.

Example:

ASSEMBLY ADDRESS LOAD ADDRESS RELOCATION CONSTANT 200_8 1000_8 1000_8

The rules for determining if operand is absolute or relocatable are as follows:

- 1. If operand is a number, it is absolute.
- 2. If operand is a direct assignment which was equated to a number, the operand is absolute.
- 3. If operand is a label which was defined within a block of absolute coding, it is absolute.
- 4. Point references (.) get current block relocation.
- 5. Undefined symbols, as operands, are relocatable if a block of relocatable code was encountered in the program, otherwise they are absolute.
- 6. All other operands are relocatable.

If an operand contains both absolute and relocatable elements, they are handled as follows:

(A = absolute, R = relocatable)

A + A = A

A - A = A

A + R = R

A - R = R flagged as possible relocation error

R + A = R

R - A = R

R - R = A

R + R = R flagged as possible relocation error

Multiplication, division, and boolean operations are not allowed on relocatable symbols.

5.0 ERROR FLAGS

The assembler will examine each source statement for possible errors. The statement which contained the error will be flagged with one or several letters in the left hand margin of the source line. The following table shows the possible error flags and their meanings.

FLAG

- A Error in address form. Assembler could not generate a requested form or indirect or external requested on a short form instruction.
- B Illegal character encountered. It is replaced with a ?
- D Statement contains a reference to a multiply defined symbol.
- E Statement contains a reference to an unresolved direct assignment.
- G Global symbol error. Will appear on an EXTERN statement if defined by user or defined as an Internal. Will appear on a direct assignment if right hand side is an external symbol.
- I Illegal redefinition of an EOPDEF or direct assignment. Value not entered into symbol table.
- M Multiply defined symbol.
- N Error in number usage.
- P Phase error. Pass 1 value of symbol does not equal Pass 2 value. (Usually fatal)
- Q . Questionable syntax. (Results may be erroneous.)
- R Possible relocation error.
- Truncation error Literal or direct assignment generated more than 1 word.
- U Undefined symbol in statement.

In addition to flagging erroneous statements, the assembler during Pass 1 will print out multiple definitions and all undefined symbols and the locations allocated to them.

6.0 ASSEMBLY OUTPUT LISTING

If the user requests it, the assembler will produce an output listing on the requested output device.

The top of each page will contain the name of the program (as supplied in the TITLE statement) and the page number.

The body of the listing will be formatted as follows:

If the source statement is a machine op code or an EOPDEF the OBJECT CODE will be formatted as follows:

All other statements will produce a 6 digit octal value.

In addition, if the object code is to be relocated by the Linking Loader it will be indicated by a single quote following the value. External symbol references will be indicated with an E.

Instructions which require more than "n" words of object code will be listed as "n" lines.

SAMPLE	PAGE 1	•	
ØØØØØ ØØØØ1' ØØØØ32' ØØØØØ3' ØØØØØ4'	Ø,4,1,ØØ6 A 1,4,1,ØØ7 1,4,1,ØØ5 6,4,1,111	TITLE : LDA STA LDA CMP	SAMPLE 4, G 4, G2 4, G1 4, ["OK"]
、	المراب أبد أبد أبد أبد أبد أبد	B 51: Ø 52: Ø END	Á

6.1 SYMBOL TABLE LISTING

After the assembly listing has been outputted, the assembler will output an alphabetically ordered symbol table which lists all user defined symbols. The symbol table listing is useful in tracing or cebugging a program for which the programmer does not have an assembly listing.

ØØØØØ'

Α

G ØØØØ6' G1 ØØØØ7' G2 ØØØ1Ø'

APPENDIX A

7-BIT ASCII CHARACTER SET

CHARACTER	ASCII	CHARACTER	ASCII
@ A	1ØØ 1Ø1	△(space)	Ø4Ø Ø41
В	1ø2	e u	Ø42
C	1ø3	#	ø43
D	1 ø4	\$	OLA A
E	1Ø5	%	Ø45
F	1ø6	&	Ø46
G	1/97	t .	Ø47
Н	1 1ø	(Ø5ø
	111)	Ø51
J	112	*	Ø52
K	113	· · · · · · · · · · · · · · · · · · ·	Ø53
L	114	,	Ø54
M	115	· · · · · · · · · · · · · · · · · · ·	Ø55
N	116	•	ø56
0	117		Ø57
P	120	Ø	Ø5Ø
Q	121	1	Ø61
R .	122	2	Ø62
S T	123	3	Ø63
T	124	4	Ø64
U	125	• 5	Ø45
V	126	6	Ø65
W	127	7	Ø57
X	13ø	8	Ø7Ø
Y	131	9	Ø71
Z [132	•	Ø72
L	133	;	Ø73
1	134	<	Ø74
1	135	=	Ø75
 ←	136	>	Ø76
NULL	137	?	Ø77
HORIZONTAL	ØØØ	FORM FEED	Ø14
TAB	ØIJ	CARRIAGE	Ø15
LINE FEED		RETURN	
VERTICAL TAB	Ø12	CODE DELETE	1 <i>77</i>
A TIVITCAL TAD	Ø13		

APPENDIX B

PERMANENT SYMBOL TABLE

SYMBOL	<u>OP</u>	<u>R</u>	DI	<u>VALUE</u>	SYMBOL	<u>OP</u>	<u>R</u>	DI	VALUE
VDD(1)				0/0000	14111			***	. (0
ADD(L)	3		*	060000	MUL	• 6		101	140101
AND(L)	2	^		040000	NEG(L)	5	3		126000
BR(L)	4	3		106000	POB	6	6	116	154116
BSR (L)	4	7		116000	POC	6	4	116	150116
BCN(L)	4	0		100000	POL	6	7	116	156116
BCZ(L)	4	4		110000	POP	6	5	a 116	152116
BLS(L)	4	1		102000	PSC	7	7	006	176006
BN(L)	4	2		104000	PSD	7	7	007	176007
BGE(L)	4	5		112000	PSI	7	7	004	176004
BE(L)	4	6		114000	PSR	7	7	005	176005
CLR(L)	5	7		136000	PUB	6	2	116	144116
CMP	6	_	111	140111	PUC	6	0	116	140116
COM(L)	5	1		122000	PUL	6	3	116	146116
DIV .	6		103	140103	PUSH	6	1	116	1 42116
HLT	7	7	001	176001	RCS /	7 · ·	7	002	176002
INC(L)	5	2		124000	RIO	7	7	0 00	176000
IOC	7	5		172000	RR (L)	5	4		130000
IOD	7	7		176000	SHFT	6		113	140113
IOR	7	0		160000	STA (L)	1			0 20000
IORC	7 ·	1		162000	STC	6		115	140115
IOS	7 .	4		170000	SUB	6		112	140112
IOT	7	6		174000	SWP(L)	5	6		134000
IOW	7	2		164000	•TST(L)	5	0		120000
IOWC	7 .	3		166000	TSTC	6		107	140107
LCMP	6		110	140111	TSTN	6		104	140104
LDA(L)	0			000000	TSTO	6		106	140106
LDC	6		114	140114	TSTZ	6		105	140105
LDIV	6		102	140102	WCI .	7	7	003	176003
LMH	7 '	7	011	176011					
LML	7	7	010	176010	*				
LMUL	6		100	140100					
LR(L)	5	5		132000					
LUH	7	7	013	176013					
LUL	7	7	012	176012					

By suffixing the indicated symbols with \boldsymbol{L} , the assembler will instructions.

treat them as long form

APPENDIX C

SUMMARY OF PSEUDO OPS

ASCIC Seven bit ASCII text with byte count.

ASCII Seven bit ASCII text.

ASCIP Seven bit ASCII text with parity.

ASCIPC Seven bit ASCII text with parity and byte count.

BLOCK Reserve block of memory.

END End of program.

ENDC End of conditional section.

EOPDEF Define user created operator.

EXTERN External symbol declaration.

IFDEF Conditionally assemble if defined.

IFE Conditionally assemble if $= \emptyset$.

IFGE Conditionally assemble if $\geq \emptyset$.

IFLS Conditionally assemble if $\leq \emptyset$.

IFN Conditionally assemble if $\neq \emptyset$.

IFUND Conditionally assemble if undefined.

IF1 Conditionally assemble if pass 1.

IF2 Conditionally assemble if pass 2.

INTERN Internal symbol declaration.

LBYTE Left byte pointer.

LOC Set location counter.

LPOOL Output literals.

MORG Modular origin.

PASS2 Switch to pass2 processing.

RADIX Interpret number in declared radix.

RBYTE Right byte pointer.

RIM Output readin binary with loader.

RIMNLD Output readin binary with no loader.

SHFTA Create arithmetic shift words.

SHFTC Create rotate combined shift word.

SHFTL Create logical shift word.

SHFTR Create rotate shift word.

TITLE Name of program.

VFD Variable length byte statement.

APPENDIX D

SUMMARY OF SPECIAL CHARACTER INTERPRETATIONS

The characters listed below have special meaning in the context indicated. These interpretations do not apply when the characters appear in text strings or in comments.

CH	ARAC	TER	MEANING	EXAMPLE
	#		Follows number to be shifted and precedes binary shift count	1 5 [#] 3
	D .		Specifies double precision number	1.5D 7243653D
	E		Exponent indicator. Precedes decimal exponent in floating point numbers.	25.43E5
	+	(plus)	bbA	
	ės	(minus)	Subtract Arithmetic operations	
	*	(asterisk)	Multiply	
	/ /	(slash)	Divide	
	&	(ampersand)	AND	
	1.	(exclamation)	Inclusive OR Boolean operations	
	\	(back slash)	Exclusive OR	
	\$	(dollar sign)	Legal character if encountered	\$TAG%
	%	(percent sign)	in a label or symbol	
	()	(parenthesis)	Used to enclose index field.	LDA 4, Ø(2)
	1	(up arrow)	Indicates half word generation for long form instructions	CMP ⁴ 4, (1)

. <u>CH</u>	ARACTER	MEANING	EXAMPLE
:	(colon)	Immediately follows all labels	LABEL: LDA 4,5
;	(semicolon)	Precedes all comments	; this is a comment
•	(point)	1) Has current value of the location	B .+5
		counter 2) Local radix charge to decimal	187.
•	(comma)	General operand or argument delimiter	INTERN SYM1, SYM2, SYM3
		2) Accumulator field delimiter	LDA 4,5
	(square brackets)	Delimits a literal	LDA 4, [123]
= = .;;	(equal sign)	Indicates a direct assignment	A = 1
@	(at sign)	Indicates indirect addressing	BRL @TAG
· , ′ .	(single quote marks)	Enclose 7-bit ASCII text symbol, one or two characters.	'AB'
<>	(angle brackets)	1) Enclose an expression within ASCII text	ASCII/ABC/< 15 ><12> <ff></ff>
		 Enclose byte size in VFD statement 	VFD <4> A, 1, <3>2,2<2> 1

APPENDIX E

Operand Formats

The instruction set of the PDP-X in addition to using the op code bits $(\emptyset-2)$ for identification sometimes uses the R bits (3-5) and/or D I (8-15) to identify the instruction. Because of this condition, the user should not use operands in a format that will alter the instruction.

The following table shows the legal and illegal formats for the PDP-X instruction repertoire.

BASIC INST	ructions			
CLASS 1 - 0	OP bits: only			. •
	LEGAL			ILLEGAL
OP OP	,OPERAND AC, OPERAND		OP	OPERAND (implies AC=4)
CLASS 2 - G	OP & R bits			
	LEGAL			ILLEGAL
OP	OPERAND	•	OP OP	, OPERAND AC, OPERAND
EXTENDED	INSTRUCTIONS			
CLASS 1 - G	OP & D]			
	LEGAL			ILLEGAL
OP OP	, OPERAND AC, OPERAND		ОР	OPERAND (implies AC=4)
CLASS 2 - C	OP, D1 & R			
	LEGAL			ILLEGAL
OP	OPERAND		OP OP	, OPERAND AC, OPERAND

I/O INSTRUCTIONS

CLASS 1 - OP & R

LEGAL ILLEGAL

OP DEV, OPERAND
OP OPERAND
OP OPERAND
implies DEV = Ø

CLASS 2 - OP, DI & R

LEGAL

OP OPERAND OP ,OPERAND OP DEV,OPERAND

(END OF MANUAL)

2.2.2 CHARACTER SET

The input to XAP-6 is prepared in 7-bit ASCII. Refer to Appendices A and D of the user's manual for a description of the acceptable ASCII characters and a summary of special character interpretations.

2.2.3 EXAMPLES

Examples of all language features may be found throughout the user's manual.

2.3 OUTPUT

2.3.1 OUTPUT FORMAT

The listing output format of XAP-6 is described in the user's manual, section 6.9.

The binary formats of the object program is absolute.

2.3.2 CHARACTER SET

The XAP-6 listing is in ASCII.

2.3.3 EXAMPLES

A sample XAP-6 output listing may be found in section 6.6 of the user's manual.

2.4. ORGANIZATION

2.4.1 OPERATIONAL ORGANIZATION

XAP-6 is a two pass assembler which requires that the source be read in twice.

2.4.2 INTERNAL ORGANIZATION

The entire assembler is resident in core at all times.

3. OPERATING PROCEDURE

LOADING PROCEDURE

XAP-6 relocatable binary is loaded in the following manner:

- A. .R LOADER n requests loader with core required.
- B. *DEV:XAP6, EXEC device which contains XAP6.

At this stage XAP6 is loaded.

SAVE DEV:XAP6

puts it on the specified device in dump mode.

3.1.1 CONDITIONAL LOAD

NOT APPLICABLE.

3.2 SWITCH SETTINGS

Switches are specified in the command string by typing "/" followed by a single switch letter. Any number of switch settings may be indicated at one time by enclosing a string of switch letters in parenthesis; for example, (WAAP) rewinds the magnetic tape, advances two files and increases the pushdown list size by 80 locations.

- All Advance magnetic tape by one file.
- B¹ Backspace magnetic tape by one file.
- Produce listing file in a format acceptable to CREF. If no listing device is specified, DSK: is assumed. If no filename is specified, CREF.TMP is assigned; if no extension is specified, TMP is assumed. In addition, the listing-device must be a retrievable device (e.g. DTA, DSK, MTA). If it is not, a command error results.
- D No symbol table in binary output. .
- Continue on source device data error. When I is not used, this type of error terminates the assembly.
- L² Force long form for all basic ops.
- N Suppress teletype error printouts
- P² Increase size of pushdown list by 80₁₀ locations for each appearance of P.
- T Skip to logical end of magnetic tape.
- W Rewind magnetic tape.
- Z^{1,2} Zero the DECtape directory.

NOTES:

- 1. This switch must immediately follow the device to which it refers.
- 2. This switch must not appear on source device specifications. If it appears on a source device specification, a command error results.

3.3 START UP PROCEDURE

After the user has logged into the system he types:

R XAP6

When XAP6 has been loaded, it responds with * (asterisk) and waits for the command string to be typed.

3.4 COMMAND LANGUAGE

The general command format is as follows:

BIN: FILENAME. EXT, LST: FILENAME. EXT -SRC: FILENAME. EXT,...

where: BIN is the object program device.

LST is the listing device.

SRC is the source device.

NOTE: A command string must be typed on one line because a line feed or carriage return terminates the string.

3.4.1 EXAMPLES

MTA1:, DTA3:/C C-CDR:)

Assemble one source file from the card reader; write the object program on MTA1; write the assembly listing on DTA3 in cross reference format and call the file CREF .TMP.

,TTY: (-TTY:)

Assemble one source file from the teletype and list the program on the teletype. Do not output any object code.

3.5 OPERATION

Throughout the assembly when a source device is encountered in the command string which requires manual loading the operator, (e.g. PTR, CDR, TTY), XAP6 will type

LOAD THE NEXT FILE ON Dev

on the console. All other devices will be loaded automatically. At the end of pass 1 the message

END OF PASS 1

will be typed on the console.

3.6 ERROR RECOVERY

3.6.1 INPUT ERRORS

XAP6 examines each source statement for possible errors and flags them with one or several letter codes. (See section 5.0 of the user manual).

3.6.2 OPERATOR ERRORS

If the command string to the assembler is typed improperly, XAP6 responds with "COMMAND ERROR" and returns an "*". The user may then retype the command string.

The following are additional messages which may occur.

Message	Meaning
CAN NOT ENTER FILE File name. ext	DTA or DSK directory is full; file can not be entered.
CAN NOT FIND filename.ext	The file can not be found on the specified device.
DATA ERROR ON DEVICE dev	Output error has occurred on the device.
INPUT ERROR ON DEVICE dev	Input error has occurred on the device.
INSUFFICIENT CORE	An insufficient amount of core is available for assembly.

Message

Meaning

dev NOT AVAILABLE	The device is assigned to another user or does not exist.
NO END STATEMENT ON INPUT FILE END STATEMENT FORCED	The END statement is missing at the end of the source program file. An END is forced and pass 2 begins.
PDP OVERFLOW, TRY /P	Pushdown list overflow. Use /P in command string to lengthen the pushdown list.

3.6.3 SOFTWARE ERRORS

There are no error halts nor are there any conditions which will cause the assembler to go into a loop.

3.6.4 HARDWARE ERRORS

If hardware failures (which are undetected by the monitor) occur, they will usually be detected and indicated on the listing as phase errors.

Peripheral errors will be indicated by an appropriate message (see 3.6.2) and control is returned to the command string.

4. INTERNAL ENVIRONMENT

4.1 TRADE-OFFS

Because XAP6 is intended to be downwared compatible with XAP (the PDP-X assembler) some features were considered and will not be implemented because of size problems.

Some of these features are:

- a) Hexadecimal numbers
- b) Radix 50
- c) Automatic optimization

The assembler was designed as a two pass assembler mainly for its capability of phase checking of labels. Some features may be conditionalized depending on the size of the computer.

4.2 SOFTWARE INTERFACES

XAP6 performs all of its Input/Output functions through calls to the monitor. (See DEC-10-MTBO-D, PDP-10/40, 10/50 Time Sharing Monitors).

All subroutines are called using the PUSHJ instruction. Arguments of subroutines which require a calling sequence will be contained in designated accumulators.

4.3 CONVENTIONS

XAP6 is designed to be re-entrant; thus, in the event that reentrant programs are recognized by the multiprogramming system on the PDP-6, only one copy of XAP6 need be resident in core for many users.

The accumulators are allocated according to major functions, namely:

- 1) Utility
- 2) Pointers
- 3) Calling sequences

4.4 LANGUAGE

XAP6 is written in MACROX language. It does not use the macro capability of MACROX.

5. EXTERNAL ENVIRONMENT

5.1 EXECUTION SPEED

5.2 USE

XAP6 is used to provide for PDP-X assemblies on the PDP-6, because of a higher availability of time on the PDP-6 as compared with other in-house computers.

5.3 INTERFACE

XAP6 is intended to be used by system programmers and diagnostic programmers for development of PDP-X software. Although it is not part of the final PDP-X software system, it will be a means of developing the PDP-X software system.

5.4 EXAMPLES OF USAGE

FORTRAN IV, DDT, MAINDEC

DOCUMENTATION

6.1 MAJOR ASPECTS

The maintenance of XAP6 will be facilitated by the following documents:

- 1) Macro flowcharts
- 2) Table formats
- 3) Heavily documented listing

6.2 CHECKOUT

XAP6 will be checked out by the implementor in the following manner.

- 1) All subroutines will be debugged.
- 2) Checkout of simple assembly statement.
- 3) Extensive checkout of syntax rules.
- 4) Comprehensive checking of the complete system.

When the above checkout has been completed to the satisfaction of the implementor, the program will be truned over to QC for further checkout.

6.3 MARKETING

Because of some of the features in XAP6, it compares favorably or even better than all assemblers for existing 16 bit computers.

Some of these features are:

- 1) MACROS
- 2) CONDITIONALS
- 3) EOPDEFS
- 4) Expressions
- 5) Variable length byte operations
- 6) Automatic relative addressing
- 7) Literal pooling
- 8) Booleans
- 9) Relocatable
- 10) Global sections
- 11) Extensive error checking
- 12) Device independence
- 13) Free form
- 14) Variable length, symbol, literal and macro storage
- 15) Takes advantage of more core