D. Vonada

CONFIDENTIAL

PDP-X Technical Memorandum # 35

TITLE:	PDP-X Assembler Initial Specifications
AUTHOR(S):	H. G. Bramson
INDEX KEYS:	Software Specifications XAP
DISTRIBUTION KEY:	B, C
OBSOLETE:	None
REVISION:	None
DATE:	November 1, 1967

PROGRAM SPECIFICATION

PDP-6/PDP-X ASSEMBLER

XAP-6

H. G. Bramson

11-5-67

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 5K of core memory. 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

"XAP"

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

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 I/O Monitor 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). XAP statements are virtually format free; i.e., elements 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.)

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

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:                 {Illegal, no spaces are allowed within the label or
TAGIT△:                  {between the last character and the colon.
```

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

Examples:

```
LABEL:  LDA  4,5

        LDA  4,LABEL#        ;Variable can not redefine a label. (See 3.3.1)

EOPDEF  LABEL,1Ø             ;EOPDEF can not redefine a label. (See 3.1)
LABEL=1Ø                     ;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)	↓	(carriage return)

Examples of Operators:

```
LDA     ;mnemonic machine instruction.
LOC     ;an assembler pseudo op.
FADD    ;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  FAD,25      ;defines FAD as a user defined operator
        LDA 3,LOC   ;"LDA" is an operator, "LOC" is an operand
        LDA+1       ;"LDA" is an operand
        LOC 5Ø      ;"LOC" is a pseudo op
        LOC+5Ø      ;"LOC" is an operand
        FAD TAG     ;"FAD" is an operator, "TAG" is an operand
        FAD+1       ;illegal statement (see 3.1)
```

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	Ø,4,Ø,5
+LDA	ØØØØØ5

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 (△), tab (→|), semicolon (;), or carriage return (↓).

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 never be used as operands.

Examples

```
LOC=5
STA: 1
     LDA  4,STA     ;STA is user defined
     STA  4,LOC     ;LOC is used defined
     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
        LDA     1,TAG     ;error, AC 1 can not be referenced
        LDA     25,TAG    ;AC5 referenced
        B       LOC       ;Correct form
        B       AC5,LOC   ;error, instruction does not require an AC
```

2.2.4 COMMENTS

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

Examples:

```
;this is a comment
A:  LDA  4,5     ;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	0-9
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

```
SYMBOL = value
```

where value can be a number or expression. Value can not be a machine instruction, pseudo op or EOPDEF defined symbol. (i.e., expression to right of = assumes the operand field).

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
```

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	A →I=5	
B=△1Ø	B△=△1Ø	
C=→I 2Ø	A=LDA△TAG	;operand field assumed and therefore there are too many operands.

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  4,4    ;same as LDA  4,5
```

A symbol may be defined after use.

Example:

```
B␣Y
Y=1
```

This is called a forward reference, and is resolved properly in Pass2. When first encountered in Pass 1, the B 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.

```
B␣Y
Y=Z
Z=1
```

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.

2.4.1 SINGLE PRECISION INTEGERS

The standard radix (base) used in all number interpretation by the assembler is octal (base 8). The radix may be changed for a single numeric term, by using the qualifier ↑ (up arrow) followed by the letter D (decimal).

Examples:

SOURCE STATEMENT	GENERATED VALUE (OCTAL)	RADIX IN EFFECT
256	ØØØ256	OCTAL
+135	ØØØ135	OCTAL
-75	1777Ø3	OCTAL (Twos complement)
↑D1ØØ	ØØØ144	DECIMAL
↑D-4Ø	17773Ø	DECIMAL (Twos complement)

2.4.2 DOUBLE PRECISION INTEGERS

Double precision integers are specified by the letter L 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 twos complement form. The radix change qualifier may also be used.

Examples:

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

2.4.3 BINARY SHIFTING

An integer may be logically shifted left by following it with the letter B, followed by a number, n, which represents the bit position in which the right hand bit of the number should be placed. "n" always represents a decimal number and may range from $0\text{-}31_{10}$. Bits leaving the left are lost and zeros enter the right end. Binary shifting may only be used with integers.

Examples:

SOURCE STATEMENT	GENERATED VALUE
125B12	001250
-15B7	171400
1B0	100000
↑D67B8	020600

2.4.4 FLOATING POINT NUMBERS

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

Examples:

SOURCE STATEMENT	GENERATED VALUE
+.19	Ø4ØØ6Ø 121727
-183.72	14127Ø Ø5Ø754
+23.279	Ø41Ø27 Ø43554

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 1Ø. The exponent will be treated as a decimal number.

Examples:

SOURCE STATEMENT	GENERATED VALUE
1.5E5	Ø42444 1174ØØ
1.5E+2	Ø41226 ØØØØØØ
1.5E-3	Ø37142 Ø46722

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 1Ø.

Examples:

SOURCE STATEMENT	GENERATED VALUE
-.36D-6	137623 Ø72274 Ø65176 174733

2.4.5 STRINGING OF NUMBERS

It was mentioned above that the user could locally change the standard radix by using the ↑ qualifier. The user can represent a string of numbers with the same radix without having to qualify each one, by preceding the string with either the pseudo op OCT (octal string) or DEC (decimal string). The forms of numbers in the string may be any of the above mentioned types. Floating point decimal numbers may not appear in an OCT string unless they are qualified as decimal.

Examples:

```
DEC     -83,37,128L,+145B13,6.3E-2
OCT     -72,↑D-38,76,12L,133B5,↑D1.2
OCT      17,27,1.2      ;1.2 is illegal because floating point
                        ;numbers must be decimal
```

2.5 EXPRESSIONS

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

The following are the allowable operators.

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

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)
A	2
B	1Ø
C	3
D	5

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

EXPRESSION	EVALUATION (OCTAL)	
A/B+A*C	000006	Remainder of A/B is lost
B/A-2*A-5	177777	(-1)
C+A&D	000005	
A+B*C&D	000004	
1+A&C	000003	
↑D50-B	000052	Note that the decimal qualification applies only to 50 and not to "B".

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 ops LOC and RELOC, which will be described later on. He may reference the Location Counter directly by using the symbol, period(.).

Consider the following example:

LOCATION COUNTER	STATEMENT	FORM
100	B .+5	SHORT (1 word)
101	LDA 4,5000	LONG (2 words)
103	STA 4,6000	LONG
105	LDA 5,20	SHORT

The first statement, B .+5, refers to 5 locations away from the current instruction and therefore references location 1Ø5. If the B .+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 Ø, (either first or second word), 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			Assignments, no values generated.
TAG=20			
LDA 4,@TAG	Ø,4,Ø,2ØØ	100020	Two words generated
LDA 4,@LDA	Ø,4,Ø,2ØØ	100050	
@TAG	100020		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)
B    0(X3) or B   (X3)
STA  5,TAG(2)
LDA  4,@ TAG(X2)          ;indexed, indirect
```

2.6.4 LITERALS

In assembler statements, a symbolic data reference may be replaced by a direct representation of the data enclosed within brackets ([ ]). This direct representation is called a literal. The reference may be a number, user defined symbol (exclusive of EOPDEF) or an expression. All symbols used within literals will always be treated as operands rather than operators. In addition only 1 word may be generated by the literal. The assembler will cause the immediate mode of addressing to occur with the value of the literal as the effective word.

STATEMENT	GENERATED VALUE	
A=50		
LDA 4,[A]	Ø,4,1,2ØØ ØØØØ5Ø	
LDA 4,[-5]	Ø,4,1,2ØØ 177773	
LDA 4,[A-3]	Ø,4,1,2ØØ ØØØØ45	
LDA 4,[2*A-1]	Ø,4,1,2ØØ ØØØ117	
LDA 4,[5ØL]	Ø,4,1,2ØØ ØØØØØØ	- error, too many words
LDA 4,[LDA A]	Ø,4,1,2ØØ ØØØØØØ	- error, LDA undefined

2.7 BASIC INSTRUCTION FORMS

Normally, the assembler allocates two memory locations for all instructions, including Basic Op Codes. The programmer can explicitly instruct the assembler to allocate only one memory location for basic ops on either an individual statement basis or a block basis.

Individual statements may be qualified by using an expanded set of index values. In addition, the index value may state the form of addressing to be used.

A block of statements may be made short by the usage of a pseudo op, BEGS (See 2.7.2).

2.7.1 INDIVIDUAL STATEMENT QUALIFICATION

It was mentioned previously that the method of indicating an index value was to suffix the operand with a value enclosed in parenthesis. The following are the legal index values and their meanings. (Any other values are flagged as errors and ignored. If the form indicated is illegal for the instruction, it will also be flagged, and the indicated length will be generated for the instruction.)

VALUE	MEANING	COMMENTS
2	Long, indexed with 2	If encountered in a short (BEGS) area it will be assembled short and is legal only if the operand is absolute and within ±Ø-177.
3	Long, indexed with 3	
2Ø	Short, direct	Legal only if operand is absolute and ≤377 and ≠ 2ØØ.
21	Short, relative to PC	Legal only if location and operand have same address form (abs/abs - rel/rel) and difference between operand and Location Counter is within ±177.
22	Short, indexed with 2	Legal only if operand is absolute and within ±Ø-177.
23	Short, indexed with 3	
24	Short, direct or relative whichever is possible	Legal only if operand complies to rules of either value 20 or 21.
3Ø	Long, direct	Instructions encountered in short (BEGS) area with any of these values will always be long.
31	Long, immediate	
32	Long, indexed with 2	
33	Long, indexed with 3	

All values may be used with all instructions, except 2Ø-24 which may only be used with basic ops.

Examples:

ASSUMED LOCATION$_8$	SOURCE	GENERATED VALUES	
100	LDA 4,5(24)	010005	
500	LDA 4,600(24)	010500	
100	LDA 4,500(24)	010000	error
100	CMP 4,5(24)	150111	error
	LDA 4,5Ø(20)	010050	
	LDA 4,2ØØ(20)	010000	error
	LDA 4,4ØØ(20)	010000	error
	CMP 4,5Ø(20)	150111	error
100	LDA 4,5Ø(21)	010750	
100	LDA 4,2ØØ(21)	010500	
100	LDA 4,3ØØ(21)	010000	error
100	CMP 4,5(21)	150111	error
100	LDA 4,5Ø(22)	011350	
100	LDA 4,2ØØ(22)	011100	
100	LDA 4,3ØØ(22)	011000	error
100	CMP 4,5(22)	151111	error
	LDA 4,2ØØ(30)	010200 000200	
	LDA 4,25ØØ(30)	010200 002500	
	CMP 4,3ØØØ(30)	150111 003000	
	LDA 4,150123(31)	010600 150123	(same as LDA 4,[150123]
	CMP 4,123(31)	150511 000123	(same as CMP 4,[123]
	LDA 4,↑D15(31)	010600 000017	(same as LDA 4,[↑D15]
	LDA 4,375(32)	011200 000375	
	CMP 4,2ØØ(32)	151111 000200	
100	LDA 4,5Ø(2)	011200 000050	
100	LDA 4,2Ø0(2)	011200 000200	
100	CMP 4,200(2)	151111 000200	

The following four types of operands will always cause long form to be created and can not be overidden by index values to create short form:

1)	Indirect references	-	LDA 4,@ TAG	
2)	Literal references	-	LDA 4, [1,2,3]	cause
3)	Text operands (See 3.6.1)	-	LDA 4, "AB"	immediate form
4)	External symbol references	-	BAL EXT	

With respect to the nature of the relocatability of an instruction and its operand at object time the following are the legitimate forms.

LOCATION	OPERAND	ONLY TYPE(S) ALLOWED FOR SHORT
RELOCATABLE	RELOCATABLE	RELATIVE ONLY
RELOCATABLE	ABSOLUTE	DIRECT ONLY
ABSOLUTE	RELOCATABLE	RELATIVE ONLY
ABSOLUTE	ABSOLUTE	DIRECT OR RELATIVE

2.7.2 BLOCK QUALIFICATION

If the user desired to have all basic ops in a specified area to be generated as short form (either direct or relative, whichever is possible), he may do so by the use of two pseudo ops.

1) BEGS - Beginning of short form area.
2) ENDS - End of short form area.

If the basic op can not be made short, only one word will be allocated for it and it will be flagged as an error. In addition, the user may specify individual forms within the area.

Example:

```
        BEGS            ;BEGINNING OF SHORT AREA
W:      Ø
B:      Ø
L:      Ø
E:      125
LOOP:   LDA  4,B        ;SHORT
        STA  4,L        ;SHORT
        LDA  4,         ;  FORCES LONG
        AND  4,         ;    FORCES LONG
        CMP  4,W        ;LONG - EOP
        B    P(3Ø)      ;LONG
        LDA  4,L        ;SHORT
        CMP  4,E        ;LONG - EOP
        BP   LOOP       ;SHORT

        ENDS            ;END OF SHORT AREA
P:      STA  4,5        ;LONG - NOT IN RANGE OF SHORT AREA
```

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.

The following pseudo ops, of necessity, were previously described in the indicated sections:

OCT, DEC	section 2.4.5
BEGS, ENDS	section 2.7.2

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. "DI" is the value to be assigned to the D_1 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).

The following are the specific forms that an EOPDEF may be written:

```
EOPDEF    NAME, DI
EOPDEF    NAME, DI,
EOPDEF    NAME, DI,R
EOPDEF    NAME, DI,R,
```

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

Examples:

```
EOPDEF    A,12Ø
EOPDEF    B,121,2
```

			OP,R,X,DI		
A	6,5	generates	6,6,Ø,12Ø	ØØØØØ5	
A	5	generates	6,4,Ø,12Ø	ØØØØØ5	and flagged (AC=4 assumed)
B	5	generates	6,2,Ø,121	ØØØØØ5	
B	4,5	generates	6,2,Ø,121	ØØØØØ5	and flagged

The following rules must be followed when using EOPDEF's, otherwise phase errors might occur at assembly time.

1. They must be defined prior to usage.

2. They must not be redefined.

3.2 EXP

The EXP pseudo op allows the user to express a string of numbers or expressions in one statement. They will be treated as operands. It is written in the following form:

```
EXP     n,n,...,n
```

Examples:

```
EXP     TAG+1
EXP     TAG,TAG1,25,LOC  (Causes 4 words to be generated)
```

3.3 RESERVING STORAGE

3.3.1 VARIABLES

The user may request the assembler to assign single storage registers. These registers, whose contents may be altered at object time, are called variables. A symbol which contains a number sign (#) as one of its characters and which is not explicitly defined elsewhere is a variable.

The symbol may contain a # any number of times that it is used, but the symbol need only be defined once as a variable, but not necessarily the first time it is referenced.

Examples:

```
LDA     4,TAG#
LDA     4,TAG1
LDA     4,TAG1#
LDA     4,#T#A#G#
```

Variables are assigned memory locations at the end of the program, one memory location will be reserved for each variable. The initial contents of variable locations is unspecified.

3.3.2 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 block reserved for variables 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.

3.3.3 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 initial contents of the reserved location are unspecified.

Examples:

SOURCE STATEMENTS	LOCATIONS RESERVED (OCTAL)
A=100	
B=200	
C: BLOCK 5	5
D: BLOCK B-A	100
BLOCK D-C+1	150

3.3.4 RESERVING A BLOCK FOR A PUSHDOWN LIST

The pseudo op PBLOCK is functionally equivalent to BLOCK. In addition, it will cause a pointer and counter to be generated as the first two words of the reserved block. The amount of memory reserved is still "n" words and therefore the pseudo op causes n+2 words to be reserved.

Example:

ASSUMED LOCATION	SOURCE STATEMENT	GENERATED CODE	
		location	contents
100	PBLOCK 5	100	000102 (pointer)
		101	000005 (count)
	LDA 0,0	107	000000

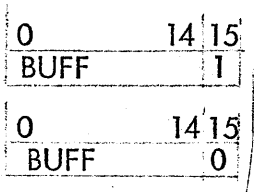
3.4 BYTE POINTERS

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_8)

```
        LDC   4,POINT1
POINT1: LBYTE BUFF        ;generates 6001
        LDC   4,POINT2
POINT2: RBYTE BUFF        ;generates 6000
```



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 1 to 16 bits may be entered using the VFD statement written in the form:

```
VFD    (N)X,X(N)Y
```

The first operand n, which must be enclosed in parenthesis, is the byte size in bits. It is interpreted as a decimal number in the range of 1-16. 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 parenthesis, immediately following any operand.

Examples:

SOURCE	GENERATED VALUE	
A=500		
B=50		
VFD (3) 1,2(6)B	025200	
VFD (7) B,(9)A	050500	
VFD (3) 1,2,3(7)A	024700	
VFD (3) 1,2,3,4,5,6	024713	error, too many bytes specified

3.6 TEXT HANDLING

Text handling enables the user to directly represent 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

If a single word of text (1 or 2 ASCII characters) is desired it can be expressed by enclosing the desired character(s) in quote marks ("). They will be stored with the first character in the right half of the word (bits 8-15) and the second character in the left half of the word (bits 0-7).

If text is used as an operand of a basic instruction it will cause long form immediate to be generated. Single word text may be used as operands of instructions, expressions, or VFD statements. Only the 64 printing characters, including space may be used for single word text. The quote sign (") itself may not be used in single word text.

Examples:

SOURCE STATEMENT	GENERATED CODE		
LDA 4,"A"	0,4,1,200	000101	
LDA 4,"AB"	0,4,1,200	041101	
LDA 4,"""	0,4,1,200	000000	error, quote may not be used in single word text.
LDA 4,"ABC"	100600	041101	error, too many characters.
"AB"	041101		
VFD (1) 1 (7)"B"(1)1(7)"A"	141301		

3.6.2 MULTIPLE WORD TEXT

If the user desires more than one word of text, he may do so by using the ASCII pseudo op. The first non blank or tab following the ASCII pseudo op will be interpreted as the text delimiter, which may be any printing character except angle brackets (< >). The text will be terminated by repeating the initial delimiter or on the occurance of a carriage return. The characters will be stored in the same manner as single word text. The character quote (") may be used in multiple word text.

Example:

SOURCE	GENERATED CODE
ASCII/THIS IS A MESSAGE/	044524 051511 044440 020123 020101 042515 051523 043501 000105
ASCII .12/30/67.	031061 031457 027460 033466
ASCII /END " ASCII /AB/.CD.	047105 020104 000042 041101 042103

Note, that in the last example more than one delimiter was used. In order to do so there may not be spaces or tabs between the delimiters.

ASCII /AB/ /CD/ would be flagged as a questionable line.

The non printing characters may be represented in an ASCII statement by enclosing them in angle brackets, in their octal equivalence. They must appear external to the text delimiters, otherwise they will be interpreted as part of the text string.

Examples:

SOURCE	GENERATED VALUE
ASCII /TEXT/<15><12>	042524 052130 005015
ASCII <15><12>/< >/	005015 037074

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.

Two pseudo ops RELOC and LOC, control both the addressing mode and location defining.

3.7.1 RELOC "n"

RELOC sets the location counter to "n", which may be a number or predefined expression, and it causes the assembler to assign relocatable locations for the instructions and data which follow. All user defined labels encountered will be relocatable. The operand relocatability depends upon the relocatability of the operand expression.

Since most relocatable programs start with the location counter set to Ø, the implicit statement, RELOC Ø, is assumed at the beginning of the program and need not be written by the user. If the user wishes to start at other than Ø, he must write a RELOC with the value desired.

Examples:

		LOCATIONS ASSIGNED	OPERAND
	BEGS RELOC 100		
A:	LDA 4,B	100 REL	REL

```
        STA     4,5Ø            1Ø1     REL             ABS

        RELOC   2ØØ
        LDA     4,C             2ØØ     REL             REL
        B       A               2Ø1     REL             REL
B:      1                       2Ø2     REL             ABS
C:      2                       2Ø3     REL             ABS
D:      A                       2Ø4     REL             REL
        ENDS
```

When a relocatable operand is associated with a short form basic op code, relocation is automatically performed by virtue of the operand being assembled relative to the location counter. However, if the relocatable operand is associated with any length 2 instruction, or if no operator is present, the operand assembles as a 15 bit relocatable expression and must be relocated via the object time Linking Loader.

Examples:

```
                                        LOCATIONS       OPERAND
                                        OCCUPIED        RELOCATED BY:
        BEGS
        RELOC   1ØØØ
A:      LDA     4,B                     1               ASSEMBLER
        STA     4,C (3Ø)                2               LOADER
        B       A                       1               ASSEMBLER
B:      D                               1               LOADER
        RELOC   2ØØØ
D:      LDA     4,B (3Ø)                2               LOADER
        STA     4,C                     1               ASSEMBLER
        B       D                       1               ASSEMBLER
C:      Ø                               1               NOT RELOCATABLE
        ENDS
```

3.7.2 LOC "n"

LOC set the location counter to "n", which may be a number or predefined expression, and it caused the assembler to assign absolute locations for the instructions and data which follow. All user defined labels encountered in a "LOC" area will be treated as absolute. The operand relocatability depends upon the relocatability of the operand expression. If the entire program is to be assembled as absolute a LOC n statement must be the first statement in the program. If it is desired to have only part of the program assemble as absolute, the LOC must be inserted where desired. Note that if a basic op which is assigned an absolute locations has a relocatable operand then long form should be stated for the statement.

Example:

```
                                        LOCATION        OPERAND
                                                        RELOCATED BY:
        BEGS
        RELOC   Ø
        LDA     4,A             Ø       REL             ASSEMBLER
```

```
A:      LDA     4,B             1   REL     NOT RELOCATABLE
R=.
        LOC     5Ø
B:      LDA     4,C (3Ø)        5Ø  ABS     LOADER
        LDA     4,B             52  ABS     NOT RELOCATABLE
        RELOC   R
C:      LDA     4,B             2   REL     NOT RELOCATABLE
        LDA     4,A             3   REL     ASSEMBLER
        LDA     4,C             4   REL     ASSEMBLER
        ENDS
```

3.7.3 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).

Examples:

```
                        LOCATION ASSIGNED
                        TO INSTRUCTION
BEGS
RELOC   5Ø
B       4Ø                  5Ø
MORG    2
LDA     4,5                 52
LDA     4,5                 53
MORG    2
LDA     4,5                 54
MORG    1ØØ
LDA     4,5                 1ØØ
ENDS
```

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 op RIM.

3.8.1 USAGE OF RIM

1. It must occur before any other statements, except the TITLE statement, otherwise it will be flagged and ignored.

2. All locations and operands in the program will be treated as absolute.

3. It forces the implicit statement LOC Ø to occur.

4. If any RELOC statements are encountered in the program they will be treated as a LOC statement, i.e., they will not force any values to be relocatable.

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 S1,S2,S3, ...Sn

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, direct assignment, or variable.

Example:

```
        INTERN      INT1,INT2,INT3
INT1:   LDA    4,5

        STA    4,INT2#
INT3=.
```

3.9.2 EXTERN S1,S2,S3, ...Sn

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.

The EXTERN statement must occur prior to usage of the external symbols in the program.

If an external symbol is an operand on a basic op, it will cause long form to be generated.

Example:

```
EXTERN      SQRT,CUBE
PUL         SQRT
PUL         CUBE
```

3.1Ø CONDITIONAL ASSEMBLY

It is often useful to have the assembler test the value of an expression and to conditionally assemble 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
IFNEG	NEGATIVE
IFNZR	NON ZERO
IFPOS	POSITIVE
IFZER	ZERO
IFDEF	DEFINED
IFUND	UNDEFINED

Examples:

A=Ø
B=1

```
        IFDEF   A
        LDA     4,5     }
        STA     4,6     }   these statements will be assembled
        ENDC
        IFZER   B
        LDA     4,5     }
        STA     4,6     }   these statements will not be assembled, but treated as comments
        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.

Example:

```
A=Ø
B=1
        IFDEF   A
        LDA     4,5                         this statement will be assembled
                IFDEF   B
                LDA     4,5                 this statement will be assembled
                ENDC
                        IFDEF   C
                        LDA     4,5         this statement will not be assembled
                        ENDC
        ENDC

        IFNEG   B-A
        LDA     4,5                         } none of these
                IFPOS   B-A                 } statements
                LDA     4,5                 } will be
                ENDC                        } assembled
        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.

4.Ø 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 instructions 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	1200_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. Variables and undefined symbols, as operands, are relocatable if a block of relocatable code was encountered in the program, otherwise they are absolute.

   In addition, they are assigned after the highest relocatable location encountered, or highest absolute location encountered if no relocatable coding was encountered.
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 and division are not allowed on relocatable symbols; however, boolean operations are allowable.

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 direct assignment, assignment ignored. (Illegal redefinition)
D	Statement contains a reference to a multiply defined symbol.
E	Statement contains a reference to an unresolved direct assignment.
F	Error in address form. Assembler could not legally generate an explicitly requested address form.
G	Will appear on EXTERN statement line if an external symbol is defined by the user, appears in an INTERN string or appears on right hand side of a direct assignment.
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.
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.Ø 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:

ERROR FLAGS	LOCATION	OBJECT CODE	SOURCE STATEMENT
XXXX	XXXXX	XXXXXX	X————X

If the location is relocatable it will be indicated by a single quote (') following the assigned location.

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

OP,R,X,DI

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
                                    TITLE     SAMPLE
                                    BEGS
ØØØØØ'    Ø,4,1,ØØ6     A:        LDA       4,G
ØØØØ1'    1,4,1,ØØ7               STA       4,G2
ØØØØ2'    1,4,1,ØØ5               LDA       4,G1
ØØØØ3'    6,4,1,111               CMP       4,"OK"
ØØØØ4'    Ø45517
ØØØØ5'    4,3,1,373               B         A
ØØØØ6'    ØØØØØØ        G:        Ø
ØØØØ7'    ØØØØØØ'       G1:       A
ØØØ1Ø'    ØØØØØØ        G2:       Ø
                                    ENDS
                                    END
```

6.1 SYMBOL TABLE LISTING

After the assembly listing has been outputted, the assembler will output a symbol table, if requested, which lists all user defined symbols. There will be two symbol lists. The first will be an alphabetically ordered list of the symbols and the second will be a list in numerical value order. The symbol table listing is useful in tracing or debugging a program for which the programmer does not have an assembly listing.

```
SAMPLE    PAGE 2

A         ØØØØØ'
G         ØØØØ6'
G1        ØØØØ7'
G2        ØØØ1Ø'
```

APPENDIX A

7-BIT ASCII CHARACTER SET

CHARACTER	ASCII	CHARACTER	ASCII
@	1ØØ	△(space)	Ø4Ø
A	1Ø1	!	Ø41
B	1Ø2	"	Ø42
C	1Ø3	#	Ø43
D	1Ø4	$	Ø44
E	1Ø5	%	Ø45
F	1Ø6	&	Ø46
G	1Ø7	'	Ø47
H	11Ø	(	Ø5Ø
I	111	)	Ø51
J	112	*	Ø52
K	113	+	Ø53
L	114	,	Ø54
M	115	-	Ø55
N	116	.	Ø56
O	117	/	Ø57
P	12Ø	Ø	Ø6Ø
Q	121	1	Ø61
R	122	2	Ø62
S	123	3	Ø63
T	124	4	Ø64
U	125	5	Ø65
V	126	6	Ø66
W	127	7	Ø67
X	13Ø	8	Ø7Ø
Y	131	9	Ø71
Z	132	:	Ø72
[	133	;	Ø73
\	134	<	Ø74
]	135	=	Ø75
↑	136	>	Ø76
←	137	?	Ø77
NULL	ØØØ	FORM FEED	Ø14
HORIZONTAL TAB	Ø11	CARRIAGE RETURN	Ø15
LINE FEED	Ø12	CODE DELETE	177
VERTICAL TAB	Ø13		

APPENDIX B

PERMANENT SYMBOL TABLE

SYMBOL	OP	R	DI	VALUE	SYMBOL	OP	R	DI	VALUE
ADD	3			060000	MUL	6		101	140101
AND	2			040000	NEG	5	3		126000
B	4	3		106000	POB	6	6	116	154116
BAL	4	7		116000	POC	6	4	116	150116
BCN	4	0		100000	POL	6	7	116	156116
BCZ	4	4		110000	POP	6	5	116	152116
BM	4	1		102000	PSC	7	7	006	176006
BN	4	2		104000	PSD	7	7	007	176007
BP	4	5		112000	PSI	7	7	004	176004
BZ	4	6		114000	PSR	7	7	005	176005
CLR	5	7		136000	PUB	6	2	116	144116
CMP	6		111	140111	PUC	6	0	116	140116
COM	5	1		122000	PUL	6	3	116	146116
DIV	6		103	140103	PUSH	6	1	116	142116
HLT	7	7	001	176001	RCS	7	7	002	176002
INC	5	2		124000	RIO	7	7	000	176000
IOC	7	5		172000	RL	5	5		132000
IOD	7	7		176000	RR	5	4		130000
IOR	7	0		160000	SHFT	6		113	140113
IORC	7	1		162000	*SHFTA	6		113	140113
IOS	7	4		170000	*SHFTC	6		113	140113
IOT	7	6		174000	*SHFTL	6		113	140113
IOW	7	2		164000	*SHFTR	6		113	140113
IOWC	7	3		166000	STA	1			020000
LCMP	6		110	140111	STC	6		115	140115
LDA	0			000000	SUB	6		112	140112
LDC	6		114	140114	SWP	5	6		134000
LDIV	6		102	140102	TST	5	0		120000
LMH	7	7	011	176011	TSTC	6		107	140107
LML	7	7	010	176010	TSTN	6		104	140104
LMUL	6		100	140100	TSTO	6		106	140106
LUH	7	7	013	176013	TSTZ	6		105	140105
LUL	7	7	012	176012	WCI	7	7	003	176003

* Extensions to the SHFT instruction.

APPENDIX C

SUMMARY OF PSEUDO OPS

ASCII	Seven bit ASCII text.
BEGS	Beginning of short area.
BLOCK	Reserve block of memory.
DEC	Decimal number string.
END	End of program.
ENDC	End of conditional section.
ENDS	End of short area.
EOPDEF	Defines user created operator.
EXP	Expression string.
EXTERN	External symbol declaration.
IFDEF	Conditionally assemble if defined.
IFNEG	Conditionally assemble if negative.
IFNZR	Conditionally assemble if non zero.
IFPOS	Conditionally assemble if positive.
IFUND	Conditionally assemble if undefined.
IFZER	Conditionally assemble if zero.
INTERN	Internal symbol declaration.
LBYTE	Left byte pointer.
LOC	Absolute location assignment.
MORG	Module origin.
OCT	Octal number string.
PBLOCK	Reserve block of memory with pointer.
RBYTE	Right byte pointer.
RIM	Prepare output in readin mode.
RELOC	Relocatable location assignment.
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.

CHARACTER	MEANING	EXAMPLE
B	Follows number to be shifted and precedes binary shift count	15B13
D	Specifies double precision floating point number	1.5D
E	Exponent indicator. Precedes decimal exponent in floating point numbers.	25.43E5
L	Specifies double precision integer	2ØØL
+ (plus)	Add	
- (minus)	Subtract	
* (asterisk)	Multiply	
/ (slash)	Divide	
& (ampersand)	AND	
! (exclamation)	Inclusive OR	
\ (back slash)	Exclusive OR	
$ (dollar sign)	Legal character if encountered	$TAG%
% (percent sign)	in a label or symbol	
( ) (parenthesis)	1. Used to enclose index field. 2. Enclose the byte size in VFD statements.	LDA 4,Ø(2) VFD (8)4,(8)3
↑ (up arrow)	Indicates local radix range followed by D	↑D50
: (colon)	Immediately follows all labels	LABEL: LDA 4,5
; (semicolon)	Precedes all comments	; this is a comment
. (point)	Has current value of the location counter	B .+5
, (comma)	1. General operand or argument delimiter 2. Accumulator field delimiter	OCT 1,2,3 EXP A,B,C LDA 4,5
[ ] (square brackets)	Delimits a literal	LDA 4, [123]

Arithmetic operations

Boolean operations

CHARACTER	MEANING	EXAMPLE
= (equal sign)	Indicates a direct assignment	A=1
@ (at sign)	Indicates indirect addressing	B @TAG
# (number sign)	Used to indicate a variable symbol	LDA 4,VAR#
". ." (quote marks)	Enclose 7-bit ASCII text, one or two characters.	"AB"
< > (angle brackets)	Enclose a numeric quantity within ASCII text	ASCII/ABC/<15><12>

APPENDIX E

Operand Formats

The instruction set of the PDP-X in addition to using the op code bits (Ø-2) for identification sometimes uses the R bits (3-5) and/or D_1 (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 INSTRUCTIONS

CLASS 1 - OP bits only

LEGAL	ILLEGAL
OP ,OPERAND	OP OPERAND (implies AC=4)
OP AC,OPERAND	

CLASS 2 - OP & R bits

LEGAL	ILLEGAL
OP OPERAND	OP ,OPERAND
	OP AC,OPERAND

EXTENDED INSTRUCTIONS

CLASS 1 - OP & D_1

LEGAL	ILLEGAL
OP ,OPERAND	OP OPERAND (implies AC=4)
OP AC,OPERAND	

CLASS 2 - OP, D_1 & R

LEGAL	ILLEGAL
OP OPERAND	OP ,OPERAND
	OP AC,OPERAND

I/O INSTRUCTIONS

CLASS 1 - OP & R

LEGAL	ILLEGAL
OP DEV,OPERAND	OP ,OPERAND OP OPERAND } implies DEV=Ø

CLASS 2 - OP,D_1 & R

LEGAL	ILLEGAL
OP OPERAND	OP ,OPERAND
	OP DEV,OPERAND

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.Ø.

The binary formats of the object program - NOT YET AVAILABLE.

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

3.1 LOADING PROCEDURE

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

A. .R LOADER n - requests loader with core required.
B. *DEV:XAP6) - 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

A - Advance magnetic tape reel by one file.
B - Backspace magnetic tape reel by one file.
C - Produce listing file in a format acceptable as input to CREF.
N - Suppress teletype error printouts.
S - Force short form for basic ops.
T - Skip to logical end of magnetic tape.
W - Rewind magnetic tape.
Z - Zero the DECtape directory.

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:

OBJPROG-DEV:FILENAME.EXT,LIST-DEV:FILENAME.EXT ←
SOURCE-DEV:FILENAME.EXT,....SOURCE-n)

where: OBJPROG-DEV is the object program device.
LIST-DEV is the listing device.
SOURCE-DEV is the source input device.

3.4.1 EXAMPLES

MTA1: ,DTA3:/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 coding.

3.5 OPERATION

If the source file is on a medium which must be manually re-entered by the operator (PTR: CDR: TTY), XAP6 will indicate this by either of the following messages.

1) (Bell) end of Pass 1
or 2) (Bell) Load the next file

All other devices used for input will automatically proceed into Pass 2 of assembly or be loaded automatically.

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.Ø 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	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.
IMPROPER INPUT DATA	The input data is not in the proper format.
INPUT ERROR ON DEVICE dev:	Input error has occurred on the device.

Message	Meaning
INSUFFICIENT CORE	An insufficient amount of core is available for assembly.
dev: NOT AVAILABLE	The device is assigned to another user or does not exist.
NO END STATEMENT ENCOUNTERED ON INPUT FILE	The END statement is missing at the end of the source program file.

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 downward 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) Syntax restrictions of the source statements
d) Automatic optimization

The assembler was designed as a two pass assembler mainly for its phasing capabilities. 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 a multiprogramming system is implemented for the PDP-6, only one copy of XAP6 need be resident in core for many users.

The accumulators will be allocated to 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

NOT YET AVAILABLE.

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

6. 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 turned over to QC for further checkout.

6.3 MARKETING

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

Some of these features are:

1) MACROS
2) CONDITIONALS
3) EOPDEF'S
4) Expressions
5) Variable length byte operations