MACRO XVM ASSEMBLER LANGUAGE MANUAL

DEC-XV-LMALA-A-D

XVM Systems

digital

MACRO XVM ASSEMBLER LANGUAGE MANUAL

DEC-XV-LMALA-A-D

digital equipment corporation · maynard. massachusetts

First Printing, December, 1975

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

Digital Equipment Corporation assumes no responsibility for the use or reliability of its software on equipment that is not supplied by DIGITAL.

The postage prepaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist us in preparing future documentation.

1/76-15

CONTENTS

Contents (Cont)

Contents (Cont)

LIST OF ALL XVM MANUALS

The following is a list of all XVM manuals and their DEC numbers, including the latest version available. Within this manual, other XVM manuals are referenced by title only. Refer to this list for the DEC numbers of these referenced manuals.

BOSS XVM USER'S MANUAL	DEC-XV-OBUAA-A-D
CHAIN XVM/EXECUTE XVM UTILITY MANUAL	DEC-XV-UCHNA-A-D
DDT XVM UTILITY MANUAL	DEC-XV-UDDTA-A-D
EDIT/EDITVP/EDITVT XVM UTILITY MANUAL	DEC-XV-UETUA-A-D
8TRAN XVM UTILITY MANUAL	DEC-XV-UTRNA-A-D
FOCAL XVM LANGUAGE MANUAL	DEC-XV-LFLGA-A-D
FORTRAN IV XVM LANGUAGE MANUAL	DEC-XV-LF4MA-A-D
FORTRAN IV XVM OPERATING ENVIRONMENT MANUAL	DEC-XV-LF4EA-A-D
LINKING LOADER XVM UTILITY MANUAL	DEC-XV-ULLUA-A-D
MAC11 XVM ASSEMBLER LANGUAGE MANUAL	DEC-XV-LMLAA-A-D
MACRO XVM ASSEMBLER LANGUAGE MANUAL	DEC-XV-LMALA-A-D
MTDUMP XVM UTILITY MANUAL	DEC-XV-UMTUA-A-D
PATCH XVM UTILITY MANUAL	DEC-XV-UPUMA-A-D
PIP XVM UTILITY MANUAL	DEC-XV-UPPUA-A-D
SGEN XVM UTILITY MANUAL	DEC-XV-USUTA-A-D
SRCCOM XVM UTILITY MANUAL	DEC-XV-USRCA-A-D
UPDATE XVM UTILITY MANUAL	DEC-XV-UUPDA-A-D
VP15A XVM GRAPHICS SOFTWARE MANUAL	DEC-XV-GVPAA-A-D
VT15 XVM GRAPHICS SOFTWARE MANUAL	DEC-XV-GVTAA-A-D
XVM/DOS KEYBOARD COMMAND GUIDE	DEC-XV-ODKBA-A-D
XVM/DOS READER'S GUIDE AND MASTER INDEX	DEC-XV-ODGIA-A-D
XVM/DOS SYSTEM MANUAL	DEC-XV-ODSAA-A-D
XVM/DOS USERS MANUAL	DEC-XV-ODMAA-A-D
XVM/DOS V1A SYSTEM INSTALLATION GUIDE	DEC-XV-ODSIA-A-D
XVM/RSX SYSTEM MANUAL	DEC-XV-IRSMA-A-D
XVM UNICHANNEL SOFTWARE MANUAL	DEC-XV-XUSMA-A-D

PREFACE

The DIGITAL XVM (XVM) MACRO Assembler program, MACRO XVM, provides the user with the symbolic programming capabilities of an assembler plus the added compiler capabilities of a many-for-one macro instruction generator. This manual describes the syntax, application and operations performed by the MACRO XVM assembler.

In the preparation of this manual it was assumed that the reader was familiar with the basic XVM symbolic instruction set.

The MACRO XVM program may be operated in:

a. Disk Operating System (XVM/DOS)
b. Batch Operating Software System, BOSS, a component of XVM/DOS
c. XVM/RSX Software System

It is assumed in this manual that the reader is familiar with the manual describing the software system under which MACRO is to be used.

The manuals involved are:

a. XVM/DOS Users Manual
b. BOSS XVM Users Manual
c. XVM/RSX System Manual

Differences in the use of MACRO in the available monitor systems are described, where applicable, in this manual.

CHAPTER 1
INTRODUCTION

1.1 MACRO XVM LANGUAGE

MACRO is a basic XVM symbolic assembler language which makes machine language programming on the XVM easier, faster and more efficient. It permits the programmer to use mnemonic symbols to represent instruction operation codes, locations, 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 standard output of the Assembler is a relocatable binary object program that can be loaded for debugging or execution by the Linking Loader. MACRO prepares the object program for relocation, and the Linking Loader, CHAIN or Task Builder, CHAIN (DOS), or Task Builder (RSX) provides relocation and sets up linkages to external subroutines. Optionally, the binary program may be output either with absolute addresses (non-relocatable) or in the full binary mode (see Chapter 3 for a description of the binary output modes).

The programmer directs the Assembler processing by using a powerful set of pseudo-operation (pseudo-op) instructions. These pseudo-ops are used to set the radix for numerical interpretation by the Assembler, to reserve blocks of storage locations, to repeat object code, to handle strings of text characters in 7-bit ASCII code or a special 6-bit code, to assemble certain coding elements if specific conditions are met, and to perform other functions which are explained in detail in Chapter 3.

The most advanced feature of the Assembler is its powerful macro instruction generator. This facility permits easy handling of recurring instruction sequences, changing only the arguments. Programmers can use macro instructions to create new language elements, adapting the Assembler to their specific programming applications. Macro instructions may be recursively called up to three levels, nested to any level, limited only by available memory, and redefined within the program. The technique of defining and calling macro instructions is discussed in Chapter 4.

An output listing, showing both the programmer's source code and the object program produced by the Assembler, 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 alphabetic error codes, which may be interpreted by referring to the error list in Chapter 5 of this manual.

Operating procedures for the MACRO XVM assembler are described in detail in Chapter 5.[*]

1.2 HARDWARE REQUIREMENTS

The MACRO XVM assembler program may be run in any configuration which meets the minimum hardware requirements for the following XVM software systems:

a. Disk Operating System (XVM/DOS)
b. Batch Operating Software System (BOSS XVM)
c. Resource Sharing Executive (XVM/RSX)

1.3 ASSEMBLER PROCESSING

The Assembler processes source programs in either a two-pass or three-pass operation. In the two-pass assembly operation the source program is read twice, with the object program and printed listed (both optional) being produced during the second pass. During PASS 1, the locations to be assigned the program symbols are resolved and a symbol table is constructed by the Assembler. PASS 2 uses the information computed during PASS 1 to produce the final object program.

In an optional three-pass assembly operation, PASS 2 calls in PASS 3, which performs a cross referencing operation during which a listing is produced that contains: (a) all user symbols, (b) where each symbol is defined, and (c) the number of each program line in which a symbol is referenced. On completion of its operation, PASS 3 calls the PASS 1 and PASS 2 portions of the assembler program back into core for further assembly operations.

[*] These procedures are also described in the XVM/DOS Keyboard Command Guide and in the On-Line Task Development section of the XVM/RSX System Manual.

Introduction

The standard object code produced by the Assembler is in a relocatable format which is acceptable to the Linking Loader, CHAIN, PATCH and TKB Utility programs. Relocatable programs that are assembled separately and use global symbols* where applicable, can be combined by the Linking Loader, CHAIN, and TKB into an executable object program. MACRO XVM reserves one additional word in a program for every external symbol**. This additional word is used as a pointer (called a transfer vector) to the actual data word in another program. The Linking Loader CHAIN or task builder sets up these transfer vectors when the programs are loaded with the actual address of the global symbol.

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

a. Reassembly of one program, which at object time was linked 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.

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

**Symbols which are referenced in the program currently being assembled but which are defined in another program.

CHAPTER 2

ASSEMBLY LANGUAGE ELEMENTS

2.1 PROGRAM STATEMENTS

One or more statements may be written on a line of up to 75 characters where the last character is a carriage-return. Since the carriage return is a non-printing character, it is graphically represented as ↵ in this manual, e.g.,

```
    STATEMENT ↵
```

Several statements may be written on a single line, separated by semicolons

```
    STATEMENT;STATEMENT;STATEMENT ↵
```

Only the last statement may have a comments field, since semicolons are allowed in and do not delimit comments. Also, macro calls (a type of statement described in a later chapter) should not appear in a multi-statement line since they cause subsequent statements to be ignored.

Normally, a single statement must fit on one line. The exception to this rule is a macro call whose arguments may be continued on a subsequent line. This is described in the chapter on macros.

2.1.1 Basic Statement Format

A basic statement may contain up to four fields that are separated by a space, spaces, or a tab character. These four fields are the label (or tag) field, the operation field, the address field, and the comments field. Because the space and tab characters are not printed, the space is represented by␣, and the tab by ⇥ in this manual. Tabs are set 8 spaces apart on DEC-supplied teleprinter machines, and are used to line up the fields in columns in the source program listing.

This is the basic statement format:

```
    LABEL ⇥ OPERATION ⇥ ADDRESS ⇥ /COMMENTS ↵
```

where each field is delimited by a tab or space, and each statement is terminated by a semicolon or carriage-return. The comments field is preceded by a tab (or space) and a slash (/).

Note that a combination of a space and a tab will be interpreted by the Assembler as two field delimiters.

Example:

```
TAG   ⇥ OP ␣ ⇥ ADR↵ } both are
TAG ␣ ⇥ OP   ⇥ ADR↵ } incorrect
```

These errors will be flagged by the assembler, but will not show on the listing because the space is hidden by the tab.

A MACRO statement may have an entry in each of the four fields, or three, or two, or only one field. The following forms are acceptable (where the character(s) indicates one or more of the preceding character):

```
TAG ↵
TAG   ⇥ OP↵
TAG   ⇥ OP    ⇥ ADDR↵
TAG   ⇥ OP    ⇥ ADDR ␣  (s)/comments ↵
TAG   ⇥ OP ␣  (s)/comments ↵
TAG   ⇥       ⇥ ADDR ↵
TAG   ⇥       ⇥ ADDR ␣ (s)/comments ↵
TAG   ⇥  (s)/comments ↵
      ⇥ OP↵
      ⇥ OP    ⇥ ADDR ↵
      ⇥ OP    ⇥ ADDR ⇥ (s)/comments ↵
      ⇥ OP    ⇥ (s)/comments ↵
      ⇥       ⇥ ADDR↵
      ⇥       ⇥ ADDR ⇥ (s)/comments ↵
/comments ↵
      ⇥ (s)/comments↵
```

A label (or tag) is a symbolic address created by the programmer to identify the statement. When a label is processed by the Assembler, it is said to be defined. A label can be defined only once. The operation code field may contain a machine mnemonic instruction code, a pseudo-op code, a macro name, a number, or a symbol. The address field may contain a symbol, number, or expression which is evaluated by the assembler to form the address portion of a machine instruction. In some pseudo-operations, and in macro instructions, this field is used for other purposes, as will be explained in this manual. *Comments* are usually short explanatory notes which the programmer adds to a statement as an aid in analysis and debugging. Comments do not affect the object program or assembly processing. They are merely printed in the program listing. Comments must be preceded by a slash (/). The slash (/) may be the first character in a line or may be preceded by:

a. Space (␣).
b. Tab (⇥)
c. Semicolon (;)

2.1.2 Direct Assignment Statement

The Direct Assignment Statement causes no object code to be generated by the assembler, but rather equates a value to a symbol at assembly time. The format of this statement is:

```
symbol=expression /comments
```

The *symbol* is the symbolic name specified to receive the value of the expression. The *expression* is any legal combination of symbols and/or constants connected by operators as described in Section 2.3.2. *Comments* are optional, as described in Section 2.1.1.

The direct assignment statement is useful for assigning a symbolic name to a constant and controlling conditional assembly. These features are explained in detail later on. Unlike *labels* defined in basic statements, which must be defined only once, the symbol defined in a direct assignment may be redefined at will.

2.2 SYMBOLS

The programmer creates symbols for use in statements to represent addresses, operation codes and numeric values. A symbol contains one to six characters from the following set:

The letters A through Z
The digits 0 through 9
Two special characters, period (.) and the percent sign (%).

The first character of a symbol must be a letter, a period, or percent sign. A period may not be used alone as a symbol. The letter 'X' alone may not be a symbol. ('X' and period alone have a special meaning to the Assembler, as explained later.)

The following symbols are legal:

MARK1	..1234	.A
A%	%50.99	.%
P9.3	INPUT	

The following symbols are illegal:

TAG:1	: is not a legal symbol character.
5ABC	First character may not be a digit.
X	Letter 'X' alone is illegal.
.	'.' alone is illegal as 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 will print "M" error flags on the lines containing SYMBOL1, SYMBOL2 and SYMBOL3. To the Assembler they are duplicates of SYMBOL. Note that "M" errors are not produced if the duplicate symbols appear in direct assignment statements.

2.2.1 Evaluation of Symbols and Globals

When the Assembler encounters a symbol during processing of a source language statement, it evaluates the symbol by referring to two tables: the user's symbol table and the permanent symbol table. The user's symbol table contains all symbols defined by the user. The user defines symbols by using them as labels, as variables, as macro names and globals, and by direct assignment statements. A label is defined when first used, and cannot be redefined. (When a label is defined by the user, it is given the current value of the location counter, as will be explained later in this chapter.)

Assembly Language Elements

All permanently defined system symbols (excluding the index register symbol, X), including system macros (except for XVM/RSX) and all Assembler pseudo-instructions use a period (.) as their first character. The Assembler also has, in its permanent symbol table, definitions of the symbols for all of the XVM memory reference instructions, operate instructions, the basic EAE instructions, and some input/output transfer instructions. (See Appendix B for a complete list of these instructions.)

XVM instruction mnemonic symbols may be used in the operation field of a statement without prior definition by the user.

Example:

```
⇥ LAC␣A↵
```

LAC is a symbol whose appearance in the operation field of a statement causes the Assembler to treat it as an op-code rather than a symbolic address. It has a value of 200000_8 which is taken from the operation code definition in the permanent symbol table.

The user can use instruction mnemonics or the pseudo-instruction mnemonics code as symbol labels. For example,

```
DZM ⇥ DZM␣Y↵
```

where the label DZM is entered in the symbol table and is given the current value of the location counter, and the op-code DZM is given the value 140000 from the permanent symbol table. The user must be careful, however, in using these dual purpose (field dependent) symbols. Symbols in the operation field are interpreted as either instruction codes or pseudo-ops, not as symbolic labels, if they are in the permanent symbol table. Macro names cannot also be defined as labels or symbols by the user. In the following example, several symbols with values have been entered in the user's symbol table and the permanent symbol table. The sample coding shows how the Assembler uses these tables to form object program storage words.

Assembly Language Elements

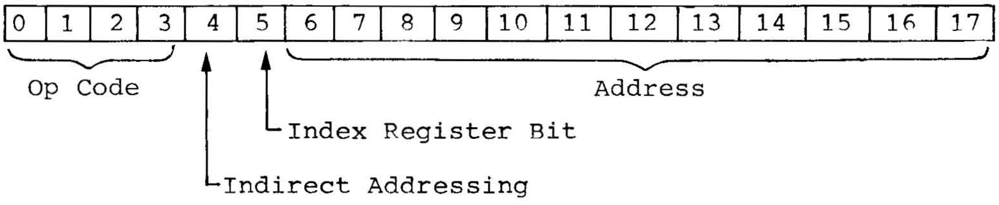
User Symbol Table		Permanent Symbol Table	
Symbol	Value	Symbol	Value
TAG1	100	LAC	200000
TAG2	200	DAC	040000
DAC	300	JMP	600000
		X	010000

The following statements generate the following code:

```
          Statement                         Code
               .
               .
TAG1  →| DAC    →| TAG2                     040200
               .
               .
TAG2  →| LAC    →| DAC                      200300
               .
               .
DAC   →| JMP    →| TAG1                     600100
      →| DAC    →| TAG1,X                   050100
      →| TAG1                               000100
               .
               .
```

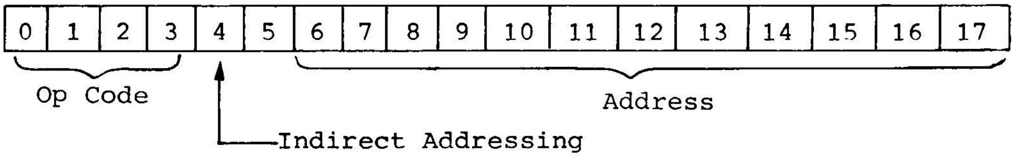
2.2.1.1 Special Symbols - The symbol X is used to denote index register usage. It is defined in the permanent symbol table as having the value of 10000. The symbol X cannot be redefined and can only be used in the address field.

2.2.1.2 Memory Referencing Instruction Format - When operating in page mode the XVM uses 12 bits for addressing, 1 bit to indicate index register usage, 1 bit to indicate indirect addressing, and 4 bits for the op-code.



PAGE MODE MEMORY REFERENCE INSTRUCTION

When operating in bànk mode on the XVM, 13 bits are used for addressing, there is no index register bit, 1 bit is for indirect addressing, and 4 bits are for the op-code.



BANK MODE MEMORY REFERENCE INSTRUCTION

2.2.2 Variables

A variable is a symbol that is defined in the user's symbol table by using it in an address field or operation field with the number sign (#). Symbols with the # may appear more than once in a program (see items 1, 3, 4, and 5 of example given below). A variable reserves a single storage word which may be referenced by using the symbol at other points in the program with or without the #. If the variable duplicates a user-defined label, it is multiply defined and is flagged as an error during assembly.

Variables are assigned memory locations at the end of the program. The initial contents of variable locations are unspecified. The # can appear any place within the symbol character string as in the example.

Example:

```
     1       00000 R 200005 R          LAC      [illegible]
     2       00001 R 040007 R          DAC      [illegible]
     3       00002 R 200005 R          LAC      TA3#
     4       00003 R 050007 R          DAC      [illegible]
     5       00004 R 200006 R          LAC      [illegible]
     6               000000 A          .END
SIZE=00010       NO ERROR LINES
```

2.2.3 Setting Storage Locations to Zero

Storage words can be set to zero as follows:

A ⇥ .0; ⇥ 0; ⇥ 0↲

In this way, three words are set to zero starting at A.

2.2.4 Redefining the Value of a Symbol

The programmer may define a symbol directly in the user's symbol table by means of a direct assignment statement written in the form:

```
SYMBOL=n ↵
   or
SYM1=SYM2 ↵
```

Where n is any number or expression. There should be no spaces between the symbol and the equal sign, or between the equal sign and the assigned value, or symbol. The Assembler enters the symbol in the symbol table, along with the assigned value. Symbols entered in this way way may be redefined. These are legal direct assignment statements:

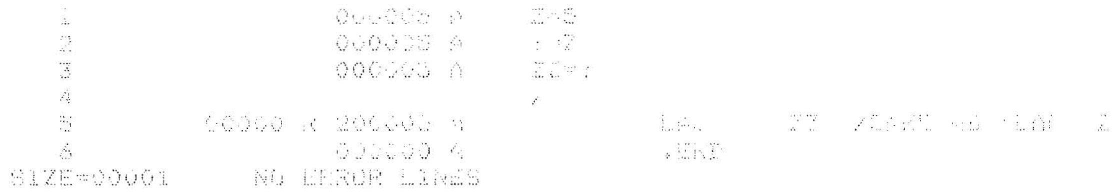
```
XX=28;A=1;B=2 ↵
```

A symbol can also be assigned a symbolic value; e.g., A=4, B=A, or

```
SET=ISZ␣SWITCH ↵
```

In the previous example, the symbol B is given the value 4, and when the symbol SET is detected during assembly the object code for the instruction ISZ␣SWITCH will be generated. This type of direct assignment cannot be used in a relocatable program. Direct assignment statements do not generate storage words in the object program.

In general, it is good programming practice to define symbols before using them in statements that generate storage words. The assembler will interpret the following sequence correctly.



```
    1                000005 A    Z=5
    2                000007 A    Y=7
    3                000007 A    ZZ=Y
    4                            /
    5        00000 R 200007 A                LAC     ZZ   /LOAD AC WITH  7
    6                000000 A                .END
 SIZE=00001      NO ERROR LINES
```

2.2.5 Forward Reference

A symbol may be defined after use. For example,

```
   1           00000 R 200001 A                  LAC      Y
   2                                    /
   3                    000001 A        Y=1
   4                    000000 A                 .END
SIZE=00001       NO ERROR LINES
```

This is called a forward reference, and is resolved properly in PASS 2. When first encountered in PASS 1, the LAC 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 storage word.

Since basic assembly operations are performed in two passes, only one-step forward references are allowed. The following example is illegal because the symbol Y is not defined during PASS 2.

```
   1     F    00000 R 200000 A                  LAC      Y
   2                                   /
   3                   000001 A        Y=Z
   4                   000001 A        Z=1
   5                   000000 A                 .END
SIZE=00001        1 ERROR LINES
```

Forward references to internal .GLOBL symbols (see Paragraph 3.9) are illegal because the internal globals are output at the beginning of PASS 2 for library searching. Globals must be defined during PASS 1, otherwise they will be flagged. The following example is illegal:

```
   1     E                                        .GL[illegible]    A,B,C
   2     F    00000 R 200000 A                    [illegible]       A
   3                   00000[illegible] A    A=[illegible]
   4          00001 R 2000[illegible] [illegible]             [illegible]       [illegible]
   5          00002 R 1[illegible]04 [illegible]      [illegible]          [illegible]       [illegible]
   6          00003 R 120[illegible] [illegible]      [illegible]          [illegible]       [illegible]
   7                   [illegible] [illegible]                 [illegible]
              00004 R 0[illegible]0004 E [illegible]
              0[illegible]005 R [illegible] E [illegible]
SIZE=00006         2 ERROR LINES
```

2.2.6 Undefined Symbols

If any symbols, except global symbols, 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 statements that referenced the undefined symbol are flagged as undefined. One memory location is reserved for each undefined symbol with the initial contents of the reserved location being unspecified.

```
     1    U    00000 R 200006 R               LAC     UNDEF1
     2         00001 R 200004 R               LAC     TAG#
     3    U    00002 R 200005 R               LAC     TAG1
     4    U    00003 R 200007 R       TAG2    LAC     UNDEF2
     5                 000000 A               .END
   SIZE=00010          3 ERROR LINES
```

2.3 NUMBERS

The initial radix (base) used in all number interpretation by the Assembler is octal (base 8). To allow the user to express decimal values and then restore to octal values, two radix-setting pseudo-ops (.OCT and .DEC) are provided. These pseudo-ops, described in Chapter 3, must be coded in the operation field of a statement. If any other information is written in the same statement, the Assembler treats the other information as a comment and flags it as a questionable line. All numbers are decoded in the current radix until a new radix control pseudo-op is encountered. The programmer may change the radix at any point in the program.

Examples:

```
   1        00000 R 200100 A            LAC     100     /INITIAL RADIX
   2        00001 R 000025 A            25              /IS OCTAL (8)
   3                                    .DEC
   4        00002 R 200144 A            LAC     100     /NOW RADIX IS
   5        00003 R 000031 A            25              /DECIMAL (10)
   6                                    .OCT
   7        00004 R 000076 A            .DSA    76      /BACK TO OCTAL
   8    N   00005 R 000143 A            99              /NOTE THAT A
   9                                                    /NON-OCTAL DIGIT
  10                                                    /CAUSES THIS LINE
  11                                                    /TO DECIMAL
  12        00006 R 000033 A            33              /STILL OCTAL
  13                000000 A            .END
SIZE=00007          1 ERROR LINES
```

2.3.1 Integer Values

An integer is a string of digits, with or without a leading sign. Negative numbers are represented in two's complement form. The range of integers is as follows:

Unsigned	$0 \rightarrow 262143_{10}$	(777777_8) or $2^{18}-1$
Signed	$0 \rightarrow 131071_{10}$	(377777_8) or $2^{17}-1$
	$0 \rightarrow -131072_{10}$	(400000_8) or -2^{17}

An octal integer* is a string of digits (0-7), signed or unsigned. If a non-octal digit (8 or 9) is encountered the string of digits will be assembled as if the decimal radix were in effect and it will be flagged as a possible error.

Example:

```
   1                                            .DEC
   2          00000 R 007303 A                  3779              /DECIMAL
   3                                            .OCT
   4          00001 R 777773 A                  -5                /TWO'S COMPL.
   5          00002 R 003347 A                  3347
   6     N    00003 R 007303 A                  3779              /DECIMAL ASSUMED
   7                  000000 A                  .END
SIZE=00004          1 ERROR LINES
```

A decimal integer** is a string of digits (0-9), signed or unsigned.

Examples:

```
   1                                            .DEC
   2          00000 R 777770 A                  -8                /TWO'S COMPL.
   3          00001 R 000400 A                  +256
   4     N    00002 R 714632 A                  -262144           /ERROR, < -2**18-1
   5     N    00003 R 303237 A                  999999            /ERROR, > 2**18-1
   6                  000000 A                  .END
SIZE=00004          2 ERROR LINES
```

[*] Preceded at some point by an .OCT pseudo-op and is also the initial assumption if no radix control pseudo-op is encountered.

[**] Preceded at some point by a .DEC pseudo-op.

2.3.2 Expressions

Expressions are strings of symbols and numbers separated by arithmetic or Boolean operators. Expressions represent unsigned numeric values ranging from 0 to $2^{18}-1$. All arithmetic is performed in unsigned integer arithmetic (two's complement), modulo 2^{18}. Division by zero is regarded as division by one and results in the original dividend. Fractional remainders are ignored; this condition is not regarded as an error. The value of an expression is calculated by substituting the numeric values for each symbol of the expression and performing the specified operations.

The following are the allowable operators to be used with expressions:

Character		Function	
Name	Symbol		
Plus	+	Addition (two's complement)	
Minus	-	Subtraction (convert to two's complement and add)	
Asterisk	*	Multiplication (unsigned)	
Slash	/	Division (unsigned)	
Ampersand	&	Logical AND	Boolean
Exclamation point	!	Inclusive OR	
Back slash	\	Exclusive OR	
Comma	,	Exclusive OR	

Operations are performed from left to right (i.e., in the order in which they are encountered). For example, the assembly language statement 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)	Comments
A	000002	
B	000010	
C	000003	
D	000005	
X	010000	Index Register Value

The following expressions to be evaluated:

Expression	Evaluation (Octal)	Comments
A+B-C,X	010007	Index Register Usage
A/B+A*C	000006	(The remainder of A/B is lost)
B/A-2*A-1+X	010003	Index Register Usage
A & B	000000	
C+A&D	000005	
B*D/A	000024	
B*C/A*D	000074	
A,X+D,X	010007	Index Register Usage Error

In the last example the expression is evaluated as follows:

Sequence of arithmetic

a. A,X = 000002 XORed with 010000 = 010002

b. A,X+D = 010002 + 000005 = 010007

c. A,X+D,X = 010007 XORed with 010000 = 000007

Note that arithmetic produces 000007 yet the value given in the example is 010007. Regardless of how the index register is used in the address field, the index register bit will always be turned on by the Assembler. In the sequence of address arithmetic above, the line would be flagged with an X because of the illegal use of the index register symbol (X).

Using the symbol X to denote index register usage causes the following restrictions:

a.	X cannot appear in the TAG field	X ⇥ LAC ⇥ A
b.	X cannot be used in a .DSA statement	⇥ .DSA ⇥ A,X
c.	X can be used only once in an expression (see 2.4.3)	⇥ LAC ⇥ A,X+D,X

2.4 ADDRESS 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 and is incremented by one each time a storage word is formed in the object program. Some statements, such as machine instructions, cause only one storage word to be generated, incrementing the location counter by one. 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.

2.4.1 Referencing the Location Counter

The programmer may directly reference the location counter by using the symbol period (.) in the address field. He can write,

```
⇥ JMP␣.-1
```

which will cause the program to jump to the storage word whose address was previously assigned by the location counter. The location counter may be set to another value by using the .LOC pseudo-op, described in Chapter 3.

2.4.2 Indirect Addressing

To specify an indirect address, which may be used in memory reference instructions, the programmer writes an asterisk immediately following the operation field symbol. This sets the defer bit (bit 4) of the storage word.

If an asterisk suffixes either a non-memory reference instruction, or appears with a symbol in the address field, an error will result.

Two examples of legal indirect addressing follow.

```
⇥ TAD* ⇥ A
⇥ LAC* ⇥ B
```

The following examples are illegal.

```
⇥ CLA*            Indirect addressing may not be specified
⇥ LAW*␣17777      in non-memory reference instructions.
```

2.4.3 Indexed Addressing

To specify indexed addressing an X is used with an operator directly after the address. No spaces or tabs may appear before the operator. The Assembler will perform whatever operation is specified with the index register symbol, and then continue to evaluate the expression. At completion of the expression evaluation, if the index bit (bit 5) is not on and the location counter is pointing to page 0 of any bank, the line is flagged with a B for bank error because the address (aside from indexing modifications) must have been greater than 7777_8 (i.e., it pointed to another page). The standard code used to indicate indexing is:

```
    ⇥LAC  ⇥A,X
```

The indexed addressing operation is illustrated in the following example.

```
  1        00000 R 210000 A    A      LAC     X        /SAME AS 'LAC  0,X'
  2        00001 R 050005 R    B      DAC     A,X+1,7-1
  3        00002 R 210001 R           LAC     B+X
  4        10000 A                    .LOC    10000    /SET TO PAGE 1
  5        10000 A 210001 A    C      LAC     X,D
  6        10001 A 210000 A    D      LAC     C,X
  7        10002 A 210000 A           LAC     C
  8                000000 A           .END
SIZE=10003     NO ERROR LINES
```

Expression evaluation where A = 000000, B = 000001, C = 010000, D = 010001, X = 010000

NOTE: $\oplus$ = exclusive OR

Location	Address Field	Discussion
0	X	The value of X is added to 0. Absence of an operator always implies addition.
1	A,X+1,7-1	000000 $\oplus$ 010000 = 010000 010000 $\oplus$ 000001 = 010001 010001 $\oplus$ 000007 = 010006 010006 - 000001 = 010005
2	B+X	000001 $\oplus$ 010000 = 010001

Location	Address Field	Discussion
10000	X,D	010000 ⊕ 010001 = 000001 The index bit has been turned off during expression evaluation. Because the location counter (10000) is pointing to Page 1, this line is not flagged, and the index register bit is turned on.
10001	C,X	010000 ⊕ 010000 = 000000 Same as example at Location 10000.

2.4.4 Literals

Symbolic data references in the operation and address fields may be replaced with direct representation of the data enclosed in parentheses. This inserted data is called a literal. The Assembler sets up the address link, so one less statement is needed in the source program. The following examples show how literals may be used, and their equivalent statements. The information contained within the parentheses, whether a number, symbol, expression, or machine instruction, is assembled and assigned consecutive memory locations after the locations used by the program, unless a .LTORG pseudo-instruction appears in the program. (See section 3.2.5.) The address of the generated word will appear in the statement that referenced the literal.

Duplicate literals, completely defined when scanned in the source program during PASS 1, are stored only once so that many uses of the same literal in a given program result in the allocation of only one memory location for that literal. Nested literals, that is, literals within literals, are illegal and will be flagged as a literal (L) error. The following is an example of a nested literal.

```
LAC␣(ADD␣(3))
```

Usage of Literal	Equivalent Statements
⇥ADD␣(1)	⇥ADD␣ONE One ⇥1
⇥LAC␣(TAG)	⇥ LAC␣TAGAD TAGAD⇥ TAG
⇥LAC␣(DAC ⇥ TAG)	⇥ LAC ␣ INST INST ⇥ DAC ⇥ TAG
⇥LAC␣(JMP ⇥ .+2)	HERE ⇥ LAC ␣ INST INST ⇥ JMP ␣ HERE+2

The following sample program illustrates how the Assembler handles literals.

```
     1       00000 R 200010 R     TAG1      LAC       (100)
     2       00001 R 040100 A     IU        DAC       100
     3       00002 R 200011 R               LAC       (JMP   .+5)
     4       00003 R 200012 R               LAC       (TAG1)
     5       00004 R 200013 R               LAC       (JMP   TAG1)
     6       00005 R 200013 R               LAC       (JMP   TAG2)
     7               000000 R     TAG2=TAG1
     8       00006 R 200014 R               LAC       (JMP   0)
     9       00007 R 200015 R     DAC       LAC       (DAC   DAC)
    10               000000 A               .END
             00010 R 000100 A *L
             00011 R 600007 R *L
             00012 R 000000 R *L
             00013 R 600000 R *L
             00014 R 600000 A *L
             00015 R 040007 R *L
  SIZE=00017      NO ERROR LINES
```

2.5 STATEMENT FIELDS

The following paragraphs provide a detailed explanation of statement fields, including how symbols and numbers may be used in each field.

2.5.1 Label Field

If the user wishes to assign a symbolic label to a statement in order to facilitate references to the storage word generated by the Assembler, he may do so by beginning the source statement with any desired symbol. The symbol must not duplicate a system or user defined macro symbol and must be terminated by a space or tab, or a statement terminating semicolon or carriage-return.

Examples:

```
TAG⇥Ø;TAG2⇥Ø;TAG3⇥Ø;TAG4⇥Ø
```

A new logical line starts after each semicolon. This line is equivalent to

```
TAG1 ⇥ 0↵
TAG2 ⇥ 0↵
TAG3 ⇥ 0↵
TAG4 ⇥ 0↵
```

If there were a tab or a space after the semicolon the symbol would be evaluated as an operation instead of a label. The sequence:

```
TAG 1⇥ 0; ⇥ TAG2;TAG3 ⇥ 0; ⇥ TAG4
```

is evaluated as follows:

```
TAG1 ⇥ 0↵
     ⇥ TAG2↵
TAG3 ⇥ 0↵
     ⇥ TAG4↵
```

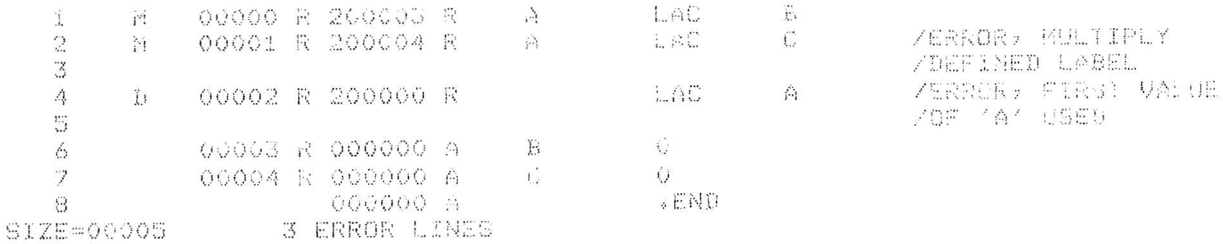
When writing numbers separated by semicolons, the first number must be preceded by a tab (⇥) or a space (␣). The sequence

```
TABLE␣1;2;3;4;5
```

produces symbol (S) errors because the first symbol of a tag cannot be numeric. The correct way to write the table sequence is as follows:

```
TABLE␣1;␣2;␣3;␣4;␣5
```

Symbols used as labels are defined in the symbol table with a numerical value equal to the present value of the location counter. A label is defined only once. If it was previously defined by the user, the current definition of the symbol will be flagged in error as a multiple definition. All references to a multiply defined symbol will be converted to the first value encountered by the Assembler.



```
    1    M   00000 R 200003 R    A       LAC     B
    2    M   00001 R 200004 R    A       LAC     C        /ERROR, MULTIPLY
    3                                                     /DEFINED LABEL
    4    D   00002 R 200000 R            LAC     A        /ERROR, FIRST VALUE
    5                                                     /OF 'A' USED
    6        00003 R 000000 A    B       0
    7        00004 R 000000 A    C       0
    8                000000 A            .END
SIZE=00005       3 ERROR LINES
```

Anything more than a single symbol to the left of the label-field delimiter is an error; it will be flagged and ignored. The following statements are illegal.

```
TAG+1 →| LAS↵
LOC*2 →| RAR↵
```

The line will be flagged with an S for symbol error. The label will be ignored but the rest of the line will continue to be processed. The only time that an error tag is not ignored is when the error occurs after the sixth character.

The statement:

```
TAGERROR*1␣NOP
```

will be assembled as:

```
TAGERR →| NOP
```

and the line will be printed and flagged with an S.

Redefinition of certain symbols can be accomplished by using direct assignments; that is, the value of a symbol can be modified. If an Assembler permanent symbol or user symbol (which was defined by a direct assignment) is redefined, the value of the symbol can be changed without causing an error message. If a user symbol, which was first defined as a label, is redefined by either a direct assignment or by using it again in the label field, it will cause an error. Variables also cannot be redefined by a direct assignment.

Examples:

```
  1               000003 A   A=3                    /SETS CURRENT VALUE
  2                                                 /OF 'A' TO '3'
  3        00000 R 200003 A           LAC     A
  4        00001 R 040003 A           DAC     A
  5               000004 A   A=4                    /REDEFINES VALUE OF
  6                                                 /'A' TO '4'
  7        00002 R 200004 A           LAC     A
  8        00003 R 040004 A   B       DAC     A
  9    A          000004 A   B=A                    /ERROR, A LABEL CAN
 10                                                 /NOT BE REDEFINED
 11               000003 R   A=B                    /BUT 'A' CAN BE RE-
 12                                                 /DEFINED TO THE VALUE 'B'
 13               123456 A   PSF=123456             /CAN REDEFINE A PERMA-
 14                                                 /NENT SYMBOL
 15        00004 R 040003 R           DAC     B
 16        00005 R 123456 A           PSF               /NOTE NEW VALUE
 17               000000 A            .END
SIZE=00006        1 ERROR LINES
```

2.5.2 Operation Field

Whether or not a symbol label is associated with the statement, the operation field must be delimited on its left by a space(s) or tab. If it is not delimited on its left, it will be interpreted as the label field. The operation field may contain any symbol, number, or expression which will be evaluated as an 18-bit quantity using unsigned arithmetic modulo 2^{18}. In the operation field, machine instruction op codes and pseudo-op mnemonic symbols take precedence over identically named user defined symbols. The operation field must be terminated by one of the following characters:

→| or ␣(s) (field delimiters)

↵ or ; (statement delimiters)

Examples:

```
     1          00000 R 440000 A     TAD       ISZ
     2          00001 R 000004 R               .+3
     3          00002 R 740007 A       CMA!CML
     4     RU   00003 R 000006 R               TAD/CMA!CCL        /ARG
           U    00004 R 000007 R
     5                  000000 A               .END
  SIZE=00010        2 ERROR LINES
```

The asterisk (*) character appended to a memory reference instruction symbol, in the operation field, causes the defer bit (bit 4) of the instruction word to be set; that is, the reference will be an indirect reference. If the asterisk (*) is appended on either a non-memory reference instruction or any symbol in the address field, it will cause an error condition which will be flagged as a symbol error (S-flag). The asterisk will be ignored and the assembly process will continue.

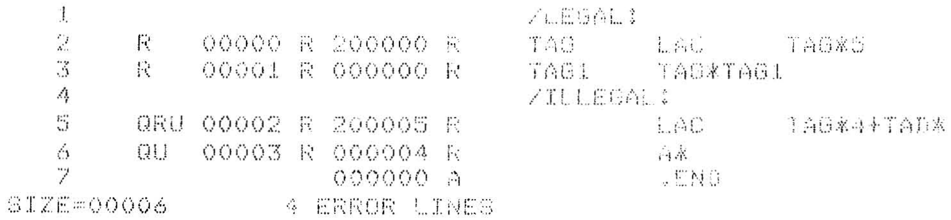
Examples:

```
     1          00000 R 360004 R               TAD*     A
     2          00001 R 220005 R               LAC*     B
     3                                   /
     4                                   /THE FOLLOWING ARE ILLEGAL
     5     G    00002 R 200004 R               LAC      A*
     6     S    00003 R 750000 A               CLA*
     7          00004 R 000001 A         A     1
     8          00005 R 000002 A         B     2
     9                  000000 A               .END
  SIZE=00006        2 ERROR LINES
```

where A = 1 and B = 2

However, the asterisk (*) may be used anywhere as a multiplication operator.

Examples:



```
      1                                 /LEGAL:
      2    R    00000 R 200000 R        TAG       LAC       TAG*5
      3    R    00001 R 000000 R        TAG1      TAG*TAG1
      4                                 /ILLEGAL:
      5    QRU  00002 R 200005 R                  LAC       TAG*4+TAD*
      6    QU   00003 R 000004 R                  A*
      7                 000000 A                  .END
 SIZE=00006          4 ERROR LINES
```

2.5.3 Address Field

The address field, if used in a statement, must be separated from the operation field by a tab, or space(s). The address field may contain any symbol, number, or expression which will be evaluated as an 18-bit quantity using unsigned arithmetic, modulo 2^{18}. If op code or pseudo-op code symbols are used in the address field, they must be user defined, otherwise they will be undefined by the Assembler and will cause an error message. The address field must be terminated by one of the following characters:

```
⇥ or ␣(s)       (field delimiters)
↵ or ;          (statement delimiters)
```

Examples:

```
LAW -1          /Correctly assembled as 777777
LAW-1           /No separation from the operation field; assembled
                as 757777 since -1 is treated as part of the oper-
                ation field.

TAG2 ⇥ DAC  ⇥ .+3
     ⇥      ⇥ TAG2/5+3␣(s)
```

The address field may also be terminated by a semicolon or a carriage-return.

Examples:

```
⇥ JMP  ⇥ BEGIN↵
⇥ TAD  ⇥ A;  ⇥ DAC  ⇥ B  ⇥ LAC
```

In the last example, a tab or space(s) is required after the semicolon in order to have the Assembler interpret DAC as being the operation field rather than the label field.

In the second line of the preceding example, the address field B is delimited by a tab. The LAC after the B →| is ignored and is treated as a comment; but the line is flagged as questionable because only a comment field may occur on a line after the address field. If the LAC had been preceded by a slash (/), the line would have been correct.

When the address field is a relocatable expression, an error condition may occur. If the program is being assembled to run in page mode, it could not execute properly if its size exceeded 4K (4096) words because it would have to load access a memory page or bank boundary. In practice, the binary loaders restrict the size to 4K-16 (4080) to avoid loading a program into the first 16 locations in a memory page or bank. This avoids a possible ambiguity where indirect memory references would be mistaken for autoincrement register references. Consequently, any relocatable address field whose value exceeds 4095 (7777_8) is meaningless in page mode and will be flagged by the Assembler as an error.

There is a similar size restriction for programs being assembled to operate in bank mode. The Assembler flags in error any relocatable address field whose value exceeds 8191 (17777_8). The binary loaders restrict the size of bank mode program to 8K-16 (8176) words.

When the address field is an absolute expression, an error condition will exist if the extended memory and page address bits (3, 4, and 5) do not match the corresponding bits of the address of the page currently being assembled into.

NOTE

> In absolute mode, the page bits do not have to be equal if the .ABS or .FULL pseudo-ops are used instead of the .ABSP or .FULLP pseudo-ops.

The Linking Loader will not relocate any absolute addresses; thus, absolute addresses within a relocatable program are relative to that page in memory in which the program is loaded.

Example:

Assume that the following source line is part of a relocatable program that was loaded into bank 1 ($20000_8 \rightarrow 37777_8$).

Source Statement	Effective Address
⇥ LAC␣300	20300

An exception to the above rule is the auto-index registers, which occupy locations 10_8 - 17_8 in page 0 of memory bank 0. The hardware will always ensure that indirect references to 10_8 - 17_8 in any page or bank will access 10_8 - 17_8 of bank 0.

2.5.4 Comments Field

Comments may appear anywhere in a statement. They must begin with a slash (/) that is immediately preceded by one of the following:

a. ␣ (s)	space(s)
b. ⇥	tab
c. ↵	carriage return/line feed (end of previous line)
d. ;	semicolon

Comments are terminated only by a carriage-return or when 74_{10} characters have been encountered in a line.

Examples:

```
␣(s)/THIS IS A COMMENT (rest of line is blank)
TAG1 ⇥ LAC␣
/THIS IS A COMMENT
  ⇥ RTR␣/COMMENT↵
  ⇥ RTR;  ⇥ RTR;/THIS IS A COMMENT
```

Observe that; ⇥ A/COMMENT↵ is not a comment, but rather an operation field expression. A line that is completely blank (containing 0 to 75 blanks/spaces) is treated as a comment by the Assembler.

A statement is terminated as follows:

↵ or; or rest of line is completely blank.

Examples:

```
→| LAC↵
→| DAC (the rest of the line is blank)
→| TAG+3
→| RTR;  →| RTR;  →| RTR↵
```

In the last example, the statement-terminating character, which is a semicolon (;) enables one source line to represent more than one word of object code. A tab or space is required after the semicolon in order to have the second and third RTRs interpreted as being in the operation field and not in the label field.

2.6 STATEMENT EVALUATION

When the Assembler evaluates a statement, it checks for symbols or numbers in each of the three evaluated fields: label, operation, and address. (Comment fields are not evaluated.)

2.6.1 Numbers

Numbers are not field dependent. When the Assembler encounters a number (or expression) in the operation or address fields (numbers are illegal in the label field), it uses those values to form the storage word. The following statements are equivalent:

```
1          00000 R 200010 P          200000   10
2          00001 R 200010 A          10+LAC
3          00002 R 200010 A          LAC      10
```

All three statements cause the Assembler to generate a storage word containing 200010. A statement may consist of a number or expression which generates a single 18-bit storage word; for example:

```
→| 23;␣45;␣357;␣62
```

This group of four statements generates four words interpreted under the current radix.

2.6.2 Word Evaluation

When the Assembler encounters a symbol in a statement field, it determines the value of the symbol by reference to the user's symbol table and the permanent symbol table, according to the priority list shown in paragraph 2.6.4.

The operation field is scanned for the following special cases:

Mnemonic	Operation Field Value
LAW	760000
AAC	723000
AXR	737000
AXS	725000
EAE instructions	64xxxx

If the operation field is not one of the special cases, the object word value is computed as follows:

If assembling for page mode:

(Operation Field +(Address Field & 7777))=Word Value

If assembling for bank mode:

(Operation Field +(Address Field & 17777))=Word Value

If the index register is used anywhere in the address field, the index register bit is set to one in the word value. If it is not used, and you are assembling for page mode then the index register bit is set to zero in the word value regardless of the address field value.

a. If index register usage is specified, the result of XORing bit 5 of the location counter and bit 5 of the address field value must be non-zero. (Otherwise the address without index modification was in a different page than the location counter, and the line is flagged with a B for bank error).

Example:

```
     1                                                .ABSP
     2       00000    210001                          LAC      A,X      /PAGE 0
     3       00001    740000        A                 NOP
     4       10000                                    .LOC     10000    /PAGE 1
     5       10000    210001                          LAC      B,X
     6       10001    210001        B                 LAC      A,X
     7                000000                          .END
  SIZE=10002       NO ERROR LINES
```

The result of statement evaluation has produced the following results:

A,X = 10001   A = 00001

B,X = 00001   B = 10001

Note that when index register usage is specified, the index register bit may or may not be on. For B,X above, the index register bit was turned off during statement evaluation. The Assembler turns this bit on after the word is evaluated, not at statement evaluation time.

At location 10001, the result of XORing bit 5 of A,X and bit 5 of the location counter is 0. This signals the Assembler that the address reference (A) is in a different page.

b. If index register usage is not specified and the program is not assembled for bank mode*, the result of XORing bit 5 of the location counter and the address field value must be 0, otherwise the line is flagged with a B for bank error.

[*] See pseudo-ops .ABS, .ABSP, .FULL, .FULLP, .EBREL, .DBREL.

Example:

```
      1                                                  .ABSP
      2     B     00000     210500                       LAC       A
      3           10500                                  .LOC      10500
      4           10500     740000         A             NOP
      5                     000000                       .END
   SIZE=10501          1 ERROR LINES
```

c. The bank bits (3,4) of the address field value in a relocatable program must never be on. The bank bits are always lost when the address field value and the operation are combined to form the object word value.

Example:

```
      1     B     00000 R 200000 R     C          LAC       A
      2           17777 R                         .LOC      C+17777
      3           17777 R 740000 A                NOP
      4           20000 R 740000 A     A          NOP
      5                   000000 A                .END
   SIZE=20001          1 ERROR LINES
```

d. The bank bits of an absolute program must equal the bank bits of the location counter. If not, the B flag alerts the programmer that he is referencing another bank.

Example:

```
      1                                          .ABSP
      2           20000                          .LOC      20000
      3     B     20000     200001               LAC       1
      4           20001     200001               LAC       20001
      5     B     20002     200001               LAC       40001
      6                     000000               .END
   SIZE=20003          2 ERROR LINES
```

e. The bank bits of lines 3 and 5 do not match those of the location counter, therefore, the lines are flagged.

2.6.3 Word Evaluation of the Special Cases

a. LAW - The operation field value and the address field value are combined as follows:

Operation Value + (Address Field Value & 17777) = Word Value

A validity check is then performed on the address field value as follows:

Address Field Value & 760000 = Validity Bits

If the validity bits are not equal to 760000 or 0, the line is flagged with an E to signal erroneous results.

b. AAC, AXR, AXS - The operation field value and the address field value are combined as follows:

Operation Value + (Address Field Value & 000777) = Word Value

The validity check:

Address Field Value & 777000 = Validity Bits

If the validity bits are not equal to 777000 or 0, the line is flagged with an E to signal erroneous results. The address field value for this type of instruction cannot be relocated. The line is flagged with an R if the address field value is relocatable.

c. EAE class instructions - The operation field value and the address field value are combined as follows:

Operation Value + Address Field Value = Word Value

The validity check:

word value and 640000 = Validity Bits

A validity check is then performed on the word value. If the validity bits differ from 640000, the line is flagged with an E error to signal erroneous results.

```
  1        00000 R 777777 A                LAW     17777
  2        00001 R 777777 A                LAW     -1
  3    E   00002 R 777777 A                LAW     677777
  4        00003 R 760000 A      A         LAW
  5        00004 R 723776 A                AAC     -2
  6        00005 R 723123 A                AAC     123
  7    E   00006 R 723000 A                AAC     -2000
  8        00007 R 750000 A                CLA     40000
  9    E   00010 R 753323 A                IDIV    100000
 10        00011 R 000023 A                23
 11                000000 A                .END
SIZE=00012        3 ERROR LINES
```

If numbers are found in the operation and address fields, they are combined in the same manner as defined symbols. For example,

```
    →| 2  →| 5  →| /GENERATES 000007
```

The value of a symbol depends on whether it is in the label field, the operation field, or the address field. The Assembler attempts to evaluate each symbol by running down a priority list, depending on the field, as shown below.

2.6.4 Assembler Priority List

Operation Field

1. Pseudo-op
2. User or System macro in macro table
3. Direct assignment in user symbol table
4. Permanent symbol table
5. User symbol table
6. Undefined

Address Field

1. The indexing symbol, X
2. User symbol table (including direct assignments)
3. Undefined

This means that if a symbol is used in the address field, it must be defined in the user's symbol table before the word is formed during PASS 2; otherwise, it is undefined. (See section 2.2.4)

In the operation field, pseudo-ops names take precedence. Direct assignments allow the user to redefine machine op codes, as shown in the example below.

Example:

```
     DPOSIT = DAC
```

System macros may be redefined by user macros, but may not be redefined as user symbols by direct assignment or by use as statement labels.

The user may use machine instruction codes and pseudo-op codes in the label field and refer to them later in the address field.

CHAPTER 3
PSEUDO OPERATIONS

The Assembler has definitions in its permanent symbol table of the symbols for all the XVM memory reference instructions, operate instructions, the basic EAE instructions, and many commonly used IOT instructions which may be used in the operation field without prior definition by the user. Also contained in the permanent symbol table are a class of symbols called pseudo-operations (pseudo-ops) which, instead of generating instructions, generate data or direct the Assembler on how to proceed with the assembly.

By convention, the first character of every pseudo-op symbol is a period (.). This convention is used in an attempt to prevent the programmer from inadvertently using, in the operation field, a pseudo-instruction symbol as one of his own.

The following is a summary of MACRO XVM Pseudo-ops.

Pseudo-op	Section	Function
.ABS	3.2.1	Object program is output in absolute, blocked, checksummed format for loading by the Absolute Binary Loader. (Neither is supported with XVM/RSX.)
.ABSP	3.2.1	
.ASCII	3.8.1	Input text strings in 7-bit ASCII code, with the first character serving as delimiter. Octal codes for non-printing control characters are enclosed in angle brackets.
.BLOCK	3.5	Reserves a block of storage words equal to the expression. If a label is used, it references the first word in the block.
.CBC	3.5.4	Initializes a word of a common block to a constant.
.CBD	3.18	Common Block Definition.
.CBDR	3.19	Common Block Definition Relative.
.CBE	3.5.5	End common block initialization section.
.CBS	3.5.3	Start common block initialization section.
.DBREL	3.2.3	Disable bank mode reloation.

Pseudo Operations

Pseudo-op	Section	Function
.DEC	3.4	Set prevailing radix to decimal.
.DEFIN	3.16	Macro definition.
.DSA	3.11	Generates a transfer vector for the specified symbol.
.EBREL	3.2.3	Enable bank mode relocation.
.EJECT	3.14	Skip to head of form on listing device.
.END	3.6	Must terminate every source program. The address field contains the address of the first instruction to be executed.
.ENDC	3.13	Terminates conditional coding resulting from .IF statements.
.ENDM	3.16	Terminates the body of a macro definition.
.EOT	3.7	Must terminate physical program segments, except the last, which is terminated by .END.
.ETC	3.16	Used in macro definitions to continue the list of dummy arguments on succeeding lines.
.FULL .FULLP	3.2.2	Produces absolute, unblocked, unchecksummed binary object programs. Used only for paper tape output. (Neither supported with XVM/RSX.)
.GLOBL	3.9	Used to declare all internal and external symbols which reference other programs.
.IFxxx	3.13	If a condition is satisfied, the source coding following the .IF statement and terminating with an .ENDC statement is assembled.
.IODEV	3.10	Specifies .DAT slots and associated I/O handlers required by this program. (Not supported with XVM/RSX.)
.LOC	3.3	Sets the location counter to the value of the expression.
.LOCAL	3.2.4	Allows deletion of certain symbols from the user symbol table.
.LST	3.17	Continues requested assembly listing output of source lines. Lines between .NOLST and .LST are not listed.
.LTORG	3.2.5	Allows the user to specifically state where literals are to be stored.
.NDLOC	3.2.4	Terminates deletion of certain symbols from the user symbol table contained between .LOCAL and .NDLOC.

Pseudo-op	Section	Function
.NOLST	3.17	Terminates requested assembly listing output of source lines of code contained between .NOLST and .LST.
.OCT	3.4	Sets the prevailing radix to octal. Assumed at start of every program.
.REPT	3.12	Repeats the object code of the next object code generating instruction.
.SIXBT	3.8.2	Input text strings in 6-bit trimmed ASCII with first character as delimiter.
.SIZE	3.15	Outputs the address of last location plus the one occupied by the object program.
.TITLE	3.1	Causes the assembler to accept characters to be printed at the top of each page of assembly listing and in the Table of Contents.

3.1 LISTING CONTROL PSEUDO-OPERATIONS

3.1.1 Program Segment Identification (.TITLE)

The program name (or any text) may be written in a .TITLE statement as shown in the following examples. The Assembler will accept up to 50_{10} characters typed until a carriage return. A form feed is output to the listing when .TITLE is encountered in the source program. The text will appear at the top of each form (page) until the next .TITLE pseudo-op. The .TITLE pseudo-op has no effect on the listing file name.

```
        →| .TITLE␣NAME OF PROGRAM
        →| .TITLE␣NAME OF SUBSECTION IN PROGRAM
```

If subsections in a program are headed by .TITLE statements, these can be used to produce a table of contents at the head of the assembly listing by use of the T option.

3.1.2 Listing Control (.EJECT)

Label Field	Operation Field	Address Field
Not used	.EJECT	Not used

When .EJECT is encountered anywhere in the source program, it causes the listing device that is being used to skip to top-of-form.

3.1.3 Listing Output Control (.NOLST and .LST)

Label Field	Operation Field	Address Field
Not used	.NOLST .LST	Not used

If, while performing an assembly listing operation (L, or N assembly parameters), the Assembler encounters a .NOLST, the listing operation will be terminated until a .LST is found. These pseudo-ops are useful when the user wishes to assemble all of a program, but only needs a listing of certain modules of the program (e.g., those which may not yet work properly). All symbols occurring between .NOLST and .LST will appear in the cross reference and symbol table listings when requested (A, V, X, or S assembly parameters).

3.2 OBJECT PROGRAM OUTPUT PSEUDO-OPERATIONS

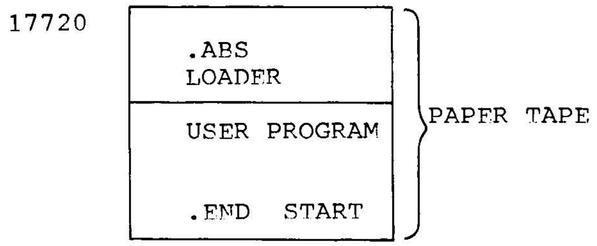
The normal object code produced by the Assembler is relocatable binary which is loaded at run time by the Linking Loader or loaded to build an executable task by CHAIN or TKB. In addition to relocatable output, the user may specify other types of output code to be generated by the Assembler.

3.2.1 Absolute Format (.ABSP and .ABS) (Not available on XVM/RSX)

.ABSP and .ABS, although accepted by the Assembler, will not work properly in XVM/RSX systems because none of the I/O handlers accept dump mode data.

Label Field	Operation Field	Address Field
Not used	.ABSP	NLD or␣or not specified
Not used	.ABS	NLD or␣or not specified

Both pseudo-ops cause absolute, checksummed binary code to be output (no values are relocatable). If no value is specified in the address field and if the output device is the paper tape punch, the Assembler will precede the output with the Absolute Binary Loader (ABL), which will load the punched output at object time. The ABL is loaded, via hardware readin, into location 17720 of any memory bank. (The ABL loads only the paper tape which follows it.) If the address field of the pseudo-op contains NLD, indicating "no loader", or if the binary output device is not the paper tape punch, the ABL will not precede the output.



NOTE

.ABS(P) output can be written on directoried devices. The Assembler assumes .ABS(P) NLD for all .ABS(P) output to file-oriented devices and appends an extension of ABS to the filename. This file can be punched with PIP, using Dump Mode. (There will be no absolute loader at the beginning of the tape.)

a. The .ABS, .ABSP, .FULL, and .FULLP pseudo-ops, specifying the type of output, must appear before any statements generating object code, otherwise the line will be flagged and ignored. Once one of these four pseudo-ops is specified, the user is not allowed to change output modes.

b. The NLD option provided in the address field of .ABS and .ABSP is meaningful only if the output device is paper tape.

A description of the absolute output format follows.

Binary Data Block (variable length, up to 34_8 words)

WORD 1 Starting address to load the block body which follows.

WORD 2 Number of words in the block body (two's complement).

WORD 3 Checksum of block body (two's complement of words 1 and 2, 4 through n).

WORD 4 Binary data to load.

⋮

WORD 3+n Binary data to load.

Starting Block - (two binary words)

WORD 1 Location to start execution of program. It is distinguished from the binary data block by having bit 0 set to 1 (negative).

WORD 2 Dummy word.

If the user requests the absolute loader and the value of the expression of the .END statement is equal to 0, the ABL halts after it has loaded in the object program. To start the program the user must set the starting address in the console address switches and press START. This allows manual intervention by the user, typically to ready I/O devices prior to starting his program. If the value of the .END expression is non-zero, it is treated as the program start address to which the ABL will automatically transfer control after loading the object program.

The .ABSP pseudo-op causes all memory referencing instructions whose addresses are in a different page to be flagged as bank errors. A DBA instruction is executed by the absolute loader before control is given to the user program. Word values which have bit 5 on will signal the processor to use the index register to compute effective addresses.

The .ABS pseudo-op does not flag memory referencing instructions whose addresses are in a different page. An EBA instruction is executed, and control is given to the user in bank addressing mode. Complete bank addressing of 8K is allowed. The processor will interpret bit 5 of all memory referencing instructions as the high order address bit. A listing of the Absolute Binary Loader is given in Appendix E.

3.2.2 Full Binary Format (.FULL and .FULLP) (Not available in XVM/RSX)

.FULL and .FULLP, although accepted by the Assembler, will not work properly in XVM/RSX systems because none of the I/O handlers accept dump mode data.

Label Field	Operation Field	Address Field	(Only useful if output is paper tape)
Not used	.FULL	Not used	
Not used	.FULLP	Not used	

The .FULL and .FULLP pseudo-ops cause full binary mode output to be produced. The program is assembled as unchecksummed absolute code and each physical record of output contains nothing other than 18-bit binary storage words generated by the Assembler. This mode is used to produce paper tapes which can be loaded via hardware readin mode. If no address is specified in the .END statement or if the address value is zero, at the end of tape the Assembler will punch a HLT instruction with channel 7 punched in the third frame. If the .END address value is non-zero, the Assembler will punch a JMP to that address, also with channel 7 of the third frame punched.

In addition, with .FULLP assembly direct memory references in page 1 to addresses in page 1 will have bit 5 set to 0 unless indexing is specified.

The only difference between the .FULL and .FULLP pseudo-ops is that memory references across page boundaries are flagged in .FULLP mode; in .FULL mode they are not.

The following specific restrictions apply to programs assembled in .FULL or .FULLP mode output.

.LOC Should be used only at the beginning of the program

.BLOCK May be used once and only if no literals, variables or undefined symbols appear in the program, and must immediately precede .END.

Variables and undefined symbols may be used if no literals or .BLOCKS appear in the program.

Literals may be used only if the program has no variables, .BLOCKs, or undefined symbols.

The reason for these restrictions, not alleviated by the use of .LTORG, is the fact that .FULL(P) mode output contains no addressing information for storing binary words other than in sequence. The .LOC and .BLOCK pseudo-ops do not generate binary output, hence there is no way to indicate skipped locations in the output. This is also true of variables and undefined symbols.

3.2.3 Relocation Mode (.EBREL and .DBREL)

Label Field	Operation Field	Address Field
Not used	.EBREL	Not used
Not used	.DBREL	Not used

The following two pseudo-ops (.EBREL and .DBREL) enable relocation mode switching. They can be used anywhere and as often as the programmer wishes in a relocatable program. In the absence of one of these mode declaration pseudo-ops, the page mode assembler assumes it is assembling 12-bit (page mode) relocatable addresses for memory reference instructions and the bank mode Assembler assumes 13-bit addresses (bank mode).

Pseudo Operations

A typical user program may omit the use of these pseudo-ops and simply prepare his object code by using the desired (bank or page mode) version of the Assembler.

For XVM page mode programs which contain display code to be interpreted by the VT15 graphics processor, it is necessary to bracket the display code with .EBREL, .DBREL. Unlike the Central Processor, the VT15 processor runs only in bank mode; hence its instruction addresses must be relocated as 13-bit values.

Mnemonic — Description

.EBREL — Enable Bank mode RELocation

Regardless of the type of Assembler being used (bank or page mode version), .EBREL causes all subsequent memory reference instruction addresses to be treated as 13-bit values, i.e., bank mode. Although in this mode, the page mode assembler will still output the "PROG>4K" warning message if the program size exceeds 4096. The 12- or 13-bit relocation is performed by the loaders. .EBREL signals the loaders to switch to 13-bit relocation by causing a dummy data word (which is not loaded) to be inserted in the binary output and having a loader code of 31_8.

.DBREL — Disable Bank mode RELocation

.DBREL is the counterpart to .EBREL. It signals the loaders, with a dummy data word (which is not loaded) and loader code of 32_8 to switch to 12-bit (page mode) relocation.

NOTE

The previous mode is not saved when an .EBREL or .DBREL is encountered; for this reason, a .DBREL pseudo-op goes directly to page mode relocation rather than entering the previous mode.

3.3 TEXT HANDLING PSEUDO OPERATIONS

The two text handling pseudo-ops enable the user to represent the 7-bit ASCII or 6-bit trimmed ASCII character sets. The Assembler converts the desired character set to its appropriate numerical equivalent (see Appendix A).

Label Field	Operation Field	Address Field
SYMBOL	$\left\{\begin{matrix}\text{.ASCII}\\ \text{.SIXBT}\end{matrix}\right\}$	Delimiter - character string - delimiter - <expression>.....

Pseudo Operations

Only the 64 printing characters (including space) may be used in the text pseudo-instructions. See nonprinting characters, Section 3.8.5. The numerical values generated by the text pseudo-ops are left-justified in the storage word(s) they occupy with the unused portion (bits) of a word filled with zeros.

3.3.1 IOPS ASCII Packed Format (.ASCII)

.ASCII denotes 7-bit ASCII characters. (It is the character set used by the operating system monitor or executive.) The characters are packed five per two words of memory with the rightmost bit of every second word set to zero. An even number of words will always be output:

First Word			Second Word			
0 6	7 13	14 17	0 2	3 9	10 16	17
1st Char.	2nd Char.	3rd Char.		4th Char.	5th Char.	0

3.3.2 Trimmed Six-Bit Format (.SIXBT)

.SIXBT denotes 6-bit trimmed ASCII characters, which are formed by truncating the leftmost bit of the corresponding 7-bit character. Characters are packed three per storage word.

0 5	6 11	12 17
1st Char.	2nd Char.	3rd Char.

3.3.3 .ASCII and .SIXBT Statement Syntax

The statement format is the same for both of the text pseudo-ops. The format is as follows.

MYTAG→	{.ASCII .SIXBT} →	delimiter	character string	delimiter	<expression>.....

3.3.4 Text Delimiter

Spaces or tabs prior to the first text delimiter or angle bracket (<) will be ignored; afterwards, if they are not enclosed by delimiters or angle brackets, they will terminate the pseudo-instruction.

Any printing character may be used as the text delimiter, except those listed below.

a. < as it is used to indicate the start of an expression.

b. ↵ as it terminates the pseudo-instruction.

(The apostrophe (') is the recommended text delimiting character.) The text delimiter must be present on both the left-hand and the right-hand sides of the text string; otherwise, the user may get more characters than desired.

3.3.5 Non-Printing Characters

The octal codes for non-printing characters may be entered in .ASCII statements by enclosing them in angle bracket delimiters. In the following statement, five characters are stored in two storage words.

```
⇥.ASCII␣'AB'<015>'CD'↵
```

Octal numbers enclosed in angle brackets will be truncated to 7 bits (.ASCII) or 6 bits (.SIXBT).

Example:

Source Line	Recognized Text	Comments
TAG⇥.ASCII␣'ABC' ⇥.SIXBT␣'ABC' ⇥.SIXBT␣'ABC'#'/#	ABC ABC ABC'/	The # is used as a delimiter in order that (') may be interpreted as text.
⇥.ASCII␣'ABCD'EFGE ⇥.ASCII␣'AB'<11> ⇥.ASCII␣'AB<11>'	ABCDFG AB ⇥ AB<11>	<11> used to represent tab. There is no delimiter after B, therefore, (<11>) is treated as text.
⇥.ASCII␣<15 × 012>'ABC' ⇥.ASCII␣<15 × 12>ABC␣(s)A	↵↓ABC ↵↓BC␣(s)	A is interpreted as the text delimiter.

The following example shows the binary word format which the Assembler generates for a given line of text.

Example:

```
→|.ASCII→|'ABC'<015 × 12>'DEF'
```

Generated Coding

Word Number	Octal	Binary
Word 1	406050	1000001 1000010 1000
Word 2	306424	011 0001101 0001010 0
Word 3	422130	1000100 1000101 1000
Word 4	600000	110 0000000 0000000 0

3.4 MACRO DEFINITION PSEUDO-OPERATIONS (.DEFIN, .ETC, and .ENDM)

The .DEFIN pseudo-op is used to define macros (described in Chapter 4). The address field in the .DEFIN statement contains the macro name, followed by a list of dummy arguments. If the list of dummy arguments will not fit on the same line as the .DEFIN pseudo-op, it may be continued by means of the .ETC pseudo-op in the operation field and additional arguments in the address field of the next line. The coding that is to constitute the body of the macro follows the .DEFIN statement. The body of the macro definition is terminated by an .ENDM pseudo-op in the operation field. (See Chapter 4 for more details on the use of macros.)

3.5 COMMON BLOCK PSEUDO-OPERATIONS

This class of pseudo-operations allows the programmer to define, reference and initialize FORTRAN-style COMMON blocks. Special Loader Codes are placed in the object output of the Assembler to allow the Linking Loader, CHAIN, or TKB to allocate memory for the specified COMMON blocks and link their addresses to transfer vectors in all programs which reference them. Additionally, the programmer may specify the initial contents of the COMMON blocks (a facility similar to the FORTRAN BLOCK DATA function).

3.5.1 Common Block Definition (.CBD)

The pseudo-op .CBD enables the programmer to declare a COMMON area of an indicated name and size and to specify the word to be set to its base address. The general format of this pseudo-op is:

Label Field	Operation Field	Address Field
User Symbol	.CBD	Name,Size

The .CBD pseudo-op takes a COMMON name and size as arguments, reserves one word of core for the base address, and outputs loader codes and parameters to direct the Linking Loader, CHAIN or TKB programs to set a transfer vector to the base address (first element) of the named COMMON array. For example, the statement:

```
BASE⇥.CBD⇥ABCD,6
```

provides location BASE with the address of the first word of the COMMON area named ABCD whose size is 6. FORTRAN blank COMMON is given a special name by the system software, .XX. To reference blank COMMON in a .CBD statement, .XX should be given as the COMMON name.

3.5.2 Common Block Definition -- Relative (.CBDR)

The pseudo-operation .CBDR (common block definition relative) takes an offset as its only argument. The general format of this pseudo-op is:

Label Field	Operation Field	Address Field
User Symbol	.CBDR	Displacement

This pseudo-op directs the Linking Loader, CHAIN or TKB to enter the starting address of the last COMMON block specified in a .CBD plus the offset given in the .CBDR into the word corresponding to the location of the .CBDR.

For example, the statements

```
BASE ⇥ .CBD ⇥  ABCD,5
BASE3⇥ .CBDR⇥  3
```

will cause the task builder to enter the starting address of the COMMON block ABCD into the location corresponding to the tag BASE; in addition, the location corresponding to BASE3 will contain the starting address of ABCD plus 3.

Note that .CBDR is relative to the last COMMON definition only. Any other assembler instructions or pseudo-operations may intervene between the .CBD and .CBDR.

3.5.3 Common Block Initialization Start (.CBS)

The pseudo-operation .CBS is used to prepare the Assembler to accept COMMON block initialization statements. The general format of this pseudo-op is:

Label Field	Operation Field	Address Field
Not used	.CBS	name [,size]

The <u>name</u> parameter specifies the name of the COMMON block which is to be initialized, see description in .CBD (Section 3.5.1) for details regarding blank COMMON. The <u>size</u> parameter is optional, and if specified represents the minimum size of the COMMON block.

This pseudo-op, unlike .CBD or .CBDR does not generate a transfer vector, hence, a label on this operation is meaningless. After a .CBS instruction and up to the next .CBE instruction (i.e., between .CPS and .CBE operations), the following rules apply to the type of statements which may be specified.

1. .CBC statements are allowed.
2. .DEC, .EJECT, .IFxxx, .ENDC, .LST, .NOLST, .OCT, .REPT, .TITLE, .DEFIN, .ENDM, .ETC are allowed.
3. Macro instructions which generate only statements belonging to 1. or 2. above are allowed.
4. Direct assignment statements are allowed.
5. Machine instructions and transfer vectors are <u>not</u> allowed.
6. Pseudo-operations other than those listed in 1 and 2 above are <u>not</u> allowed.
7. Macro instructions which generate statements belonging to 5 or 6 above are <u>not</u> allowed.

Example:

```
→|.CBS →|ABCD,6
```

indicates that a COMMON block named ABCD with a minimum length of 6 words is to be initialized by statements which follow.

3.5.4 Common Block Initialization Constant (.CBC)

The .CBC statement is used to initialize a single word of the COMMON block declared in the preceding .CBS statement. The format of the .CBC statement is:

Label Field	Operation Field	Address Field
Not used	.CBC	Displacement,Constant

The displacement parameter specifies the offset from the start of the COMMON block of the word to be initialized. The constant parameter is an absolute expression, the value of which will be used as the initial contents of the specified word in the COMMON block. If the .CBC statement is used outside the .CBS - .CBF instructions, it is flagged and ignored by the Assembler. If a .CBC statement is preceded by a .REPT statement which has a non-zero increment, the data will be incremented, and the displacement will be incremented by one. Therefore, the data generated will be placed in succeeding locations in common.

Example:

```
    .CBC   2,4
```

will set the third word (base address+2) of the COMMON block specified by the preceding .CBS to the initial value of 4.

3.5.5 Common Block Initialization End (.CBE)

The .CBE pseudo-op is used to terminate the COMMON block initialization section initiated by the .CBS operation. The general format is:

Label Field	Operation Field	Address Field
Not used	.CBE	Not used

A COMMON block initialization section (consisting of one .CBS followed by one or more .CBC's followed by a .CBE) may appear anywhere in a program without affecting the flow of the object program. Also, the same COMMON block may be initialized any number of times by any number of programs.

3.6 CONDITIONAL ASSEMBLY (.IFxxx and .ENDC)

It is often useful to assemble some parts of the source program on an optional basis. This is done in MACRO by means of conditional assembly statements, of the form:

```
    →|.IFxxx→|expression
```

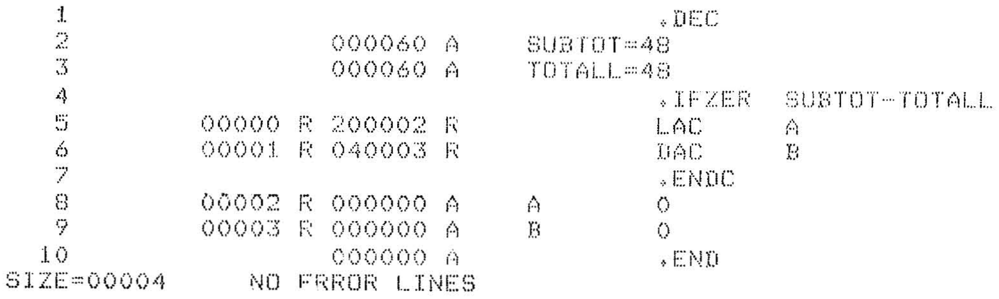
The pseudo-op may be any of the eight conditional pseudo-ops shown below, and the address field may contain any number, symbol, or expression. If there is a symbol, or an expression containing symbolic elements, such a symbol must have been previously defined in the source program or the parameter file (except for .IFDEF and .IFUND). If not, the value of the symbol or expression is assumed to be Ø, thereby satisfying three of the numeric conditionals.

If the condition is satisfied, that part of the source program starting with the statement immediately following the conditional statement and up to but not including an .ENDC (end conditional) pseudo-op is assembled. If the condition is not satisfied, this coding is not assembled.

The eight conditional pseudo-ops (sometimes called IF statements) and their meanings are shown below.

Pseudo-op	Assemble IF x is:
→|.IFPNZ␣x	Positive and non-zero
→|.IFNEG␣x	Negative
→|.IFZER␣x	Zero
→|.IFPOZ␣x	Positive or zero
→|.IFNOZ␣x	Negative or zero
→|.IFNZR␣x	Not zero
→|.IFDEF␣x	A defined symbol
→|.IFUND␣x	An undefined symbol

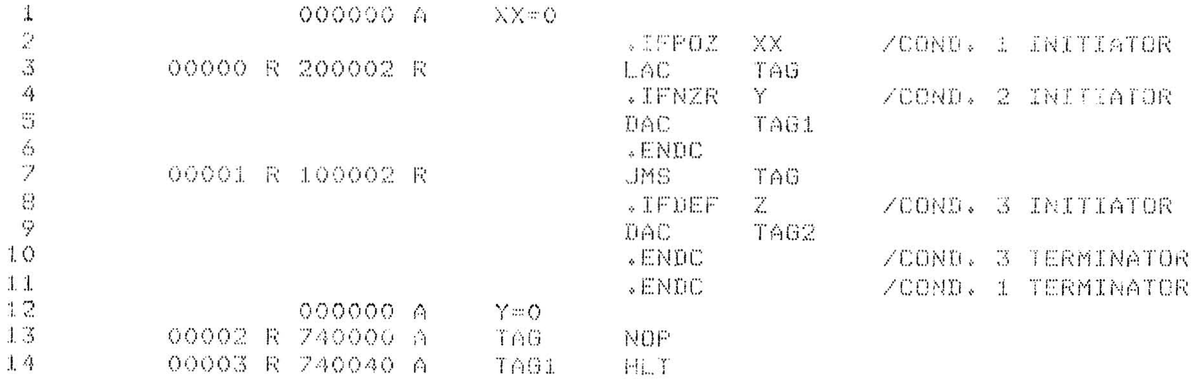
In the following sequence, the pseudo-op .IFZER is satisfied, and the source program coding between .IFZER and .ENDC is assembled.



```
      1                                                  .DEC
      2                   000060 A        SUBTOT=48
      3                   000060 A        TOTALL=48
      4                                                  .IFZER   SUBTOT-TOTALL
      5           00000 R 200002 R                       LAC      A
      6           00001 R 040003 R                       DAC      B
      7                                                  .ENDC
      8           00002 R 000000 A        A              0
      9           00003 R 000000 A        B              0
     10                   000000 A                       .END
 SIZE=00004           NO ERROR LINES
```

Conditional statements may be nested. For each IF statement there must be a terminating .ENDC statement. If the outermost IF statement is not satisfied, the entire group is not assembled. If the first IF is satisfied, the following coding is assembled. If another IF is encountered, however, its condition is tested, and the following coding is assembled only if the second IF statement is satisfied. Logically, nested IF statements are like AND circuits. If the first, second, and third conditions are satisfied, then the coding that follows the third nested IF statement is assembled.

Example:



```
 1                   000000 A     XX=0
 2                                             .IFPOZ   XX          /COND. 1 INITIATOR
 3           00000 R 200002 R                  LAC      TAG
 4                                             .IFNZR   Y           /COND. 2 INITIATOR
 5                                             DAC      TAG1
 6                                             .ENDC
 7           00001 R 100002 R                  JMS      TAG
 8                                             .IFDEF   Z           /COND. 3 INITIATOR
 9                                             DAC      TAG2
10                                             .ENDC                /COND. 3 TERMINATOR
11                                             .ENDC                /COND. 1 TERMINATOR
12                   000000 A     Y=0
13           00002 R 740000 A     TAG          NOP
14           00003 R 740040 A     TAG1         HLT
```

Conditional statements can be for a variety of purposes. One of the most useful is in terminating recursive MACRO calls (described in Chapter 4). In general, a counter is changed each time through the loop, or recursive call, until the condition is not satisfied. This process concludes assembly of the loop or recursive call.

3.7 LOCAL SYMBOLS (.LOCAL AND .NDLOC)

Label Field	Operation Field	Address Field
Not used	.LOCAL	Not used
Not used	.NDLOC	Not used

The size of a program that can be assembled with the Assembler is determined by the number of user symbols in that program and therefore by the amount of core available at assembly time in which to store those symbols. Each user symbol requires three words of core in the assembler's symbol table. This additional core is not required at run-time (unless using a debugging program like DDT) because user symbols are not loaded into core along with the object code.

The .LOCAL and .NDLOC pseudo-ops enable deletion of certain symbols from the user symbol table. In so doing, larger programs can be assembled without increasing core size. The area between these two pseudo-ops is defined as having a number of symbols, most of which are used only in this area and which can be deleted, once this area has been passed by the Assembler.

The Assembler creates a separate symbol table (local users symbol table) when the .LOCAL pseudo-op is encountered. Only labels and direct assignments may be stored in this table. Labels which have the # sign as part of the symbol are stored in the resident users symbol table (RUST). This feature is useful where a subroutine name is part of a local area but must go into the RUST because of subroutine calls from without the local area (see Section D of the following example). Symbols which are forward references (used before defined) are stored as part of the resident users symbol table. When the .NDLOC pseudo-op is encountered the local table disappears and the resident UST is left unchanged.

An example of a program which uses the .LOCAL and .NDLOC pseudo-ops follows. The symbols that are stored in the tables are represented in the comment field in the order that they are stored during PASS 1.

```
   1                                    .ABS
   2         00100                      .LOC     100
   3         00100   703302             CAF
   4         00101   100111             JMS      TTYIN
   5         00102   600137   A         JMP      C
   6         00103   777770             LAW      -10
   7         00104   040140             DAC      D
   8         00105   440140   KK        ISZ      D
   9         00106   600102             JMP      A
  10         00107   200105             LAC      KK
  11         00110   600127             JMP      AA
  12                          /-----------------------------------------------
  13                                    .LOCAL
  14         00111   000000   TTYIN     0                 /ALREADY STORED IN
  15                                                      /RUST FROM LINE 4.
  16         00112   600117             JMP      .+5
  17         00113   000000   XXX       0                 /TEMP. STORAGE OF
  18         00114   000000   YY        0                 /SUBR. TTYIN
  19         00115   000000   Z         0
  20    P    00116   000000   X1        0
  21    U    00117   000142             KSP
  22         00120   600117             JMP      .-1
  23    U    00121   000141             KRB
  24         00122   040113             DAC      XXX
  25         00123   040114             DAC      YY
  26         00124   040115             DAC      Z
  27    U    00125   040143             DAC      X1
  28         00126   620111             JMP*     TTYIN
  29                                    .NDLOC
  30                          /-----------------------------------------------
  31    U    00127   200143   AA        LAC      X1       /FORWARD REFERENCE
  32         00130   600105             JMP      KK
  33                          /-----------------------------------------------
  34                                    .LOCAL
  35         00131   000000   SYM1      0                 /TEMP. STORAGE OF
  36         00132   000000   SYM2      0                 /SUBR. TSUBR
  37         00133   000000   TSUBR#    0                 /NOT LOCAL, DUE TO #
  38         00134   200131             LAC      SYM1
  39         00135   040132             DAC      SYM2
  40         00136   620133             JMP*     TSUBR
  41                                    .NDLOC
  42                          /-----------------------------------------------
  43         00137   000001   C         1
  44         00140   000002   D         2
  45                 000000             .END
SIZE=00144         5 ERROR LINES
```

Pseudo Operations

For purposes of illustration, lines 1-11, 12-26, 27-29, and 30-36 are broken into sections A, B, C, and D respectively. The following tables show the resident and local users symbol tables (UST) at the end of each section (PASS 1 only).

	RESIDENT UST	LOCAL UST
	-	-
SECTION A		
	-	
	A	(NO SYMBOLS)
	AA	
	C	
	D	
	KK	
	TTYIN	
SECTION B		
	-	
	A	X
	AA	X1
	C	Y
	D	Z
	KK	
	TTYIN	
SECTION C		
	-	
	A	(NO SYMBOLS)
	AA	
	C	
	D	
	KK	
	TTYIN	
	X1	
SECTION D		
	-	
	A	SYM1
	AA	SYM2
	C	
	D	
	KK	
	TTYIN	
	X1	
	TSUBR	

In Section A, the symbol TTYIN is used. TTYIN is in a local area yet it is put into the resident user symbol table because it is a forward reference. The same is true of symbol X1 from Section C. Once the .NDLOC pseudo-op is encountered, the local UST no longer exists. For that reason, the X1 reference from line 28 is a forward reference. At the end of PASS 1, X1 would be represented as an undefined symbol. When Section B is processed during PASS 2, the symbol X1 would not be stored in the local UST because it already has been put into the resident table.

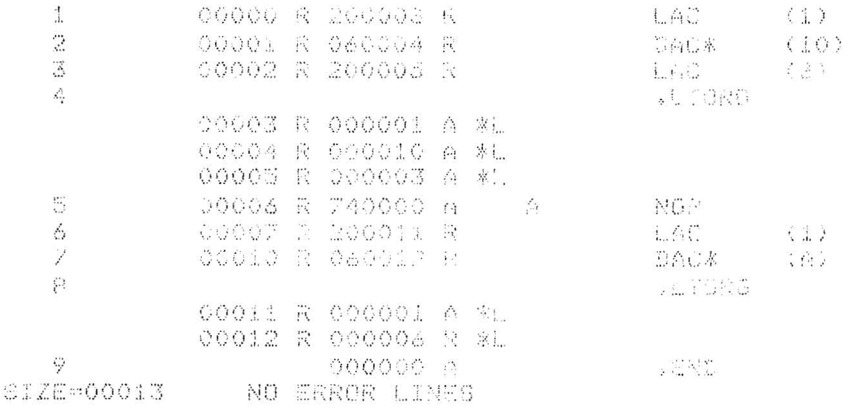
LIMITATIONS

The .LOCAL pseudo-op causes the local UST to be built just above the macro definitions. Consequently, the .DEFIN pseudo-op is illegal in a local area.

3.8 LITERAL ORIGIN (.LTORG)

Label Field	Operation Field	Address Field
Not used	.LTORG	Not used

As previously stated, a literal is an item of data with its value as stated or listed. The pseudo-op .LTORG allows the user to specifically state where he wants his literal table(s) to be stored; thus enabling the user to store literal tables in different pages or banks. As many as eight literal tables are allowed. Notice in the following example that literals are not saved from one .LTORG to the next.



```
      1         00000 R 200003 R                  LAC       (1)
      2         00001 R 060004 R                  DAC*      (10)
      3         00002 R 200005 R                  LAC       (3)
      4                                           .LTORG
                00003 R 000001 A *L
                00004 R 000010 A *L
                00005 R 000003 A *L
      5         00006 R 740000 A      A           NOP
      6         00007 R 200011 R                  LAC       (1)
      7         00010 R 060012 R                  DAC*      (6)
      8                                           .LTORG
                00011 R 000001 A *L
                00012 R 000006 A *L
      9                 000000 A                  .END
SIZE=00013        NO ERROR LINES
```

The literals 1 and 2 are stored twice even though they appear in the same bank.

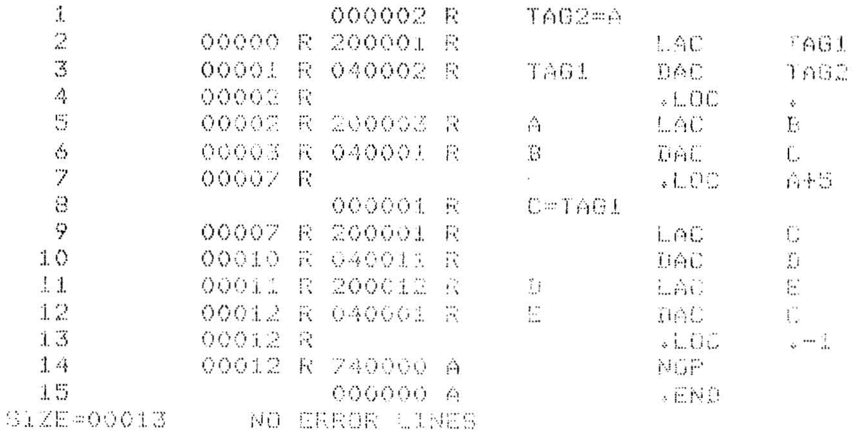
If more than eight .LTORG statements appear in a program, the excess ones will be ignored and flagged with an 1 error. Subsequent literals will be assigned core locations following the end of the program in the normal manner.

3.9 SETTING THE LOCATION COUNTER (.LOC)

Label Field	Operation Field	Address Field
Not used	.LOC	defined expression

The .LOC pseudo-op sets or resets the location counter to the value of the expression contained in the address field. The symbolic elements of the expression must have been defined previously; otherwise, phase errors will occur in PASS 2. The .LOC pseudo-op may be used anywhere and as many times as required.

Examples:



```
 1              000002 R    TAG2=A
 2      00000 R 200001 R              LAC     TAG1
 3      00001 R 040002 R    TAG1      DAC     TAG2
 4      00002 R                       .LOC    .
 5      00002 R 200003 R    A         LAC     B
 6      00003 R 040001 R    B         DAC     C
 7      00007 R                       .LOC    A+5
 8              000001 R    C=TAG1
 9      00007 R 200001 R              LAC     C
10      00010 R 040011 R              DAC     D
11      00011 R 200012 R    D         LAC     E
12      00012 R 040001 R    E         DAC     C
13      00012 R                       .LOC    .-1
14      00012 R 740000 A              NOP
15              000000 A              .END
SIZE=00013      NO ERROR LINES
```

A program headed by an absolute statement, e.g., .LOC 100 is an absolute binary program and the binary is output in link-loadable format.

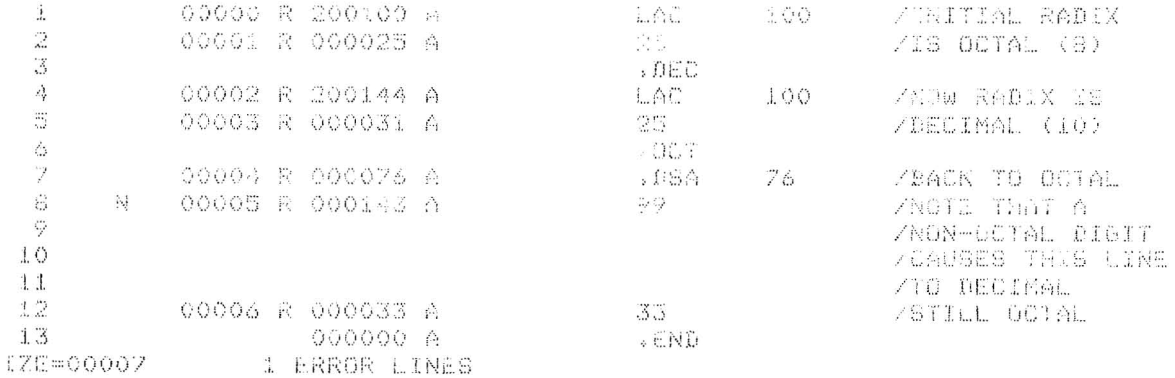
3.10 RADIX CONTROL (.OCT and .DEC)

The initial radix (base) used in all number interpretation by the Assembler is octal (base 8). In order to allow the user to express decimal values, and then restore to octal values, two radix setting pseudo-ops are provided.

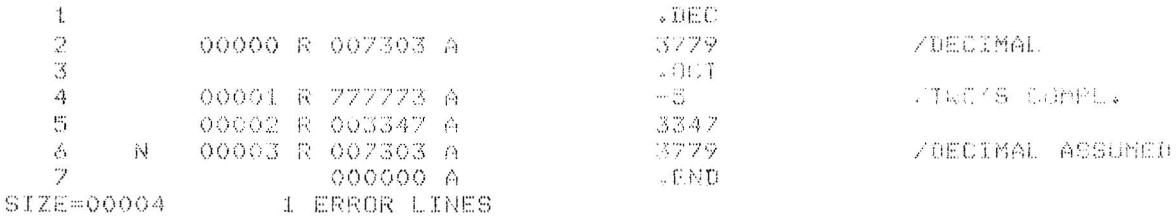
Pseudo Operations

Pseudo-op Code	Meaning
.OCT	Interpret all succeeding numerical values in base 8 (octal)
.DEC	Interpret all succeeding numerical values in base 10 (decimal)

These pseudo-instructions must be coded in the operation field of a statement. All numbers are decoded in the current radix until a new radix control pseudo-instruction is encountered unless the pseudo-op occurs within a macro expansion (see Section 4.2). The programmer may change the radix at any point in a program.



```
  1         00000 R 200100 A          LAC     100      /INITIAL RADIX
  2         00001 R 000025 A          25               /IS OCTAL (8)
  3                                   .DEC
  4         00002 R 200144 A          LAC     100      /NOW RADIX IS
  5         00003 R 000031 A          25               /DECIMAL (10)
  6                                   .OCT
  7         00004 R 000076 A          .DSA    76       /BACK TO OCTAL
  8     N   00005 R 000143 A          99               /NOTE THAT A
  9                                                    /NON-OCTAL DIGIT
 10                                                    /CAUSES THIS LINE
 11                                                    /TO DECIMAL
 12         00006 R 000033 A          33               /STILL OCTAL
 13                 000000 A          .END
IZE=00007         1 ERROR LINES
```



```
  1                                   .DEC
  2         00000 R 007303 A          3779             /DECIMAL
  3                                   .OCT
  4         00001 R 777773 A          -5               /TWO'S COMPL.
  5         00002 R 003347 A          3347
  6     N   00003 R 007303 A          3779             /DECIMAL ASSUMED
  7                 000000 A          .END
SIZE=00004        1 ERROR LINES
```

If a number is encountered which contains a decimal digit while in octal mode, the number is evaluated as if the Assembler were in decimal mode, and the line is flagged with an N.

3.11 RESERVING BLOCKS OF STORAGE (.BLOCK)

.BLOCK reserves a block of memory equal to the value of the expression contained in the address field. If the address field contains a numerical value, it will be evaluated according to the radix in effect. The symbolic elements of the expression must have been defined previously, i.e., no forward referencing is allowed; otherwise, phase errors might occur in PASS 2. The expression is evaluated modulo $2^{15}(77777_8)$. The user may reference the first location in the block of reserved memory by defining a symbol in the label field. The initial contents of the reserved locations are unspecified.

Label Field	Operation Field	Address Field
User Symbol	.BLOCK	defined expression

Examples:

```
BUFF →| .BLOCK␣12↵
     →| .BLOCK␣A+B+65↵
```

3.12 END OF PROGRAM (.END)

One pseudo-op must be included in every source program. This is the .END statement, which must be the last statement in the main program. This statement marks the physical end of the source program, and also may contain the location of the first instruction in the object program to be executed at run-time.

The .END statement is written in the general form

```
→|.END␣START↵
```

START may be a symbol, number, or expression whose value is the address of the first program instruction to be executed. In relocatable programs to be loaded by the Linking Loader, CHAIN or TKB, only the main program requires a starting address; all other subprogram starting addresses, if specified, will be ignored.

A starting address may appear in absolute or self-loading programs; if not, the program will halt after being loaded and the user must manually start his program.

These are legal .END statements

```
→|.END␣BEGIN+5↵
→|.END␣200↵
```

If no .END statement is included, the Assembler will treat it as if a .EOT was included.

3.13 END OF PROGRAM SEGMENT (.EOT)

If a program is physically segmented (on paper tape, disk, DECtape or magtape), each segment except the last may terminate with an .EOT (end-of-tape) statement or with nothing at all (neither .EOT nor .END). Termination with nothing is equivalent to termination with .EOT. The last segment must terminate with an .END statement. The .EOT statement is written without label and address fields, as follows,

→|.EOT↵

The following are typical reasons for segmenting programs:

1. A source program is prepared on three different paper tapes because one tape alone would be too large to fit in the reader.
2. A source program is split in two and stored on two DECtapes because it is larger than the capacity of a single tape.
3. To simplify program preparation, a disk file containing commonly used macro definitions is kept physically separate from user main programs. Thus, one does not have to include the macro definitions in each main program.
4. Programs can be conditionally assembled for different machine configurations or different software options. This is done by defining conditional assembly parameters at assembly time. The process can be simplified if one prepares paper tapes or mass storage files defining all parameters for a given set of options. The main program and parameter file are physically segmented one from the other but can be assembled together.

3.14 GLOBAL SYMBOL DECLARATION (.GLOBL)

Label Field	Operation Field	Address Field
Not used	.GLOBL	symbol[, symbol...]

The standard output of the Assembler is a relocatable object program. The Linking Loader, CHAIN or TKB joins relocatable programs by supplying definitions for global symbols which are referenced in one program and defined in another. The pseudo-op .GLOBL, followed by a list of symbols, is used to define to the Assembler those global symbols which are either

a. internal globals - defined in the current program and referenced by other programs

b. external symbols - referenced in the current program and defined in another program

The loader (Linking Loader, CHAIN or TKB) uses this information to include in the load and then link the relocatable programs to each other.

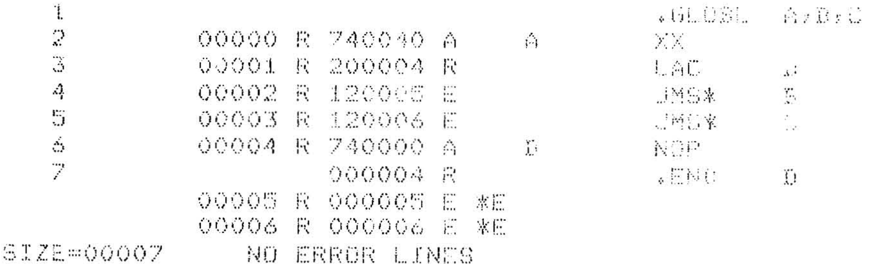
All references to external symbols must be indirect references since XVM software systems use transfer vectors for referencing external symbols. Each external symbol causes an additional word (the transfer vector word) to be reserved in the user program. The loading program will store the actual address of the external symbol in the transfer vector word. Thus, an indirect reference (through the transfer vector) will cause the external symbol location to be addressed.

Example:

```
  ⇥.GLOBL ⇥ A,B,C
A⇥LAC    ⇥ D             /A is an internal global
D⇥JMS*   ⇥ B             /These two instructions reference
 ⇥JMS*   ⇥ C             /External symbols indirectly
   .END  ⇥ D
```

The .GLOBL statement may appear anywhere within the program.

The example above is assembled as follows:



```
   1                                               .GLOBL   A,B,C
   2         00000 R 740040 A     A                XX
   3         00001 R 200004 R                      LAC      D
   4         00002 R 120005 E                      JMS*     B
   5         00003 R 120006 E                      JMS*     C
   6         00004 R 740000 A     D                NOP
   7                 000004 R                      .END     D
             00005 R 000005 E *E
             00006 R 000006 E *E
SIZE=00007     NO ERROR LINES
```

The real values for locations 3 and 4 will be supplied by the loading program: these two words will contain the addresses in memory of external symbols B and C.

3.15 REQUESTING AN I/O DEVICE HANDLER .IODEV (Not supported in XVM/RSX)

The .IODEV pseudo-op appears anywhere in the program and is used to cause the Assembler to output code for the Linking Loader or CHAIN which specifies the slots in the Monitor's device assignment table (DAT) whose associated device handlers are required by the program. This is used in XVM/DOS where device handlers are brought into core at the time a program is loaded to run.

Label Field	Operation Field	Address Field
Not used	.IODEV	datslot [,datslot...]

The arguments may be numeric or symbolic. If the argument is symbolic, the symbol must be defined by a direct assignment statement.

3.16 DESIGNATING A SYMBOLIC ADDRESS (.DSA)

.DSA (designate symbol address) is used in the operation field when it is desired to create a word composed of just a transfer vector (17-bit address). It is useful when a user tag symbol is also a permanent instruction or pseudo-op symbol.

Label Field	Operation Field	Address Field
User Symbol	.DSA	expression

Examples:

```
JMP →| LAC  →|TAG
    →| .DAS →|JMP )   Equivalent methods of designating the user
    →|      →|JMP )   symbol JMP (rather than the instruction JMP)
                      to be in the address field.
```

3.17 REPEAT OBJECT CODE (.REPT)

Label Field	Operation Field	Address Field
Not used	.REPT	count [,increment]

The .REPT pseudo-op causes the object code of the next sequential object code generating instruction to be repeated "count" times. Optionally, the object code may be incremented for each time it is repeated by specifying an increment. The count and increment may be represented by a numeric or symbolic value. If a symbol is used, it must be defined by an absolute direct assignment statement which must occur before the symbol is used. The repeated instruction may contain a label, which will be associated with the first statement generated. Note that arithmetic expressions in the increment field are illegal.

Examples:

```
     1                                                    .REPT    5
     2          00000 R 000000 A                          0
                00001 R 000000 A *R
                00002 R 000000 A *R
                00003 R 000000 A *R
                00004 R 000000 A *R
     3                                                    .REPT    4,1
     4          00005 R 000001 A                          1
                00006 R 000002 A *R
                00007 R 000003 A *R
                00010 R 000004 A *R
     5                000011 R     TAG=.
     6                                                    .REPT    4,1
     7          00011 R 600011 R                          JMP      TAG
                00012 R 600012 R *R
                00013 R 600013 R *R
                00014 R 600014 R *R
     8                000000 A                            .END
  SIZE=00015       NO ERROR LINES
```

NOTE

If the statement to be repeated generates more than one location of code, the .RFPT will repeat only the last location. For example,

```
⇥.REPT␣3
⇥.ASCII␣'A'
```

will generate the following:

```
404000      5/7 A
000000
000000      last word is
000000      repeated
```

3.18 REQUEST PROGRAM SIZE (.SIZE)

Label Field	Operation Field	Address Field
user symbol	.SIZE	not used

When the assembler encounters .SIZE, it outputs one word which contains the address of the last location plus one occupied by the object program. This is normally the length of the object program (in octal). However, if a given program is 121_8 words long and has a .LOC 4ØØ statement at the head of the program, the value of the .SIZE word will be 521_8.

CHAPTER 4
MACROS

When a program is being written, it often happens that certain coding sequences are repeated several times with only the arguments changed. It would be convenient if the entire repeated sequence could be generated by a single statement. To accomplish this, it is first necessary to define the coding sequence with dummy arguments as a macro instruction, and then use a single statement referring to the macro name along with a list of real arguments which will replace the dummy arguments and generate the desired sequence.

Consider the following coding sequence.

```
    →|LAC→|A
    →|TAD→|B
    →|DAC→|C
          .
          .
          .
    →|LAC→|D
    →|TAD→|E
    →|DAC→|F
```

The sequence

```
    →|LAC→|x
    →|TAD→|y
    →|DAC→|z
```

is the model upon which the repeated sequence is based. The characters x, y, and z are called dummy arguments and are identified as such by being listed immediately after the macro name when the macro instruction is defined.

4.1 DEFINING A MACRO

Macros must be defined before they are used. The process of defining a macro is as follows.

```
                                    (Macro Name)     (Dummy Arguments)
     (Definition Line)    ⇥.DEFIN⇥MACNME,ARG1,ARG2,ARG3⇥ /comment
                          ⇥LAC    ⇥ARG1
     (Body)               ⇥TAD    ⇥ARG2
                          ⇥DAC    ⇥ARG3
     (Terminating Line)   ⇥.ENDM
```

The pseudo-op .DEFIN in the operation field defines the symbol following it as the name of the macro. Next, follow the dummy arguments, as required, separated by commas and terminated by any of the following symbols.

a. space (␣)
b. tab (⇥)
c. carriage return (↵)

The macro name and the dummy arguments must be legal assembler symbols. Any previous definition of a dummy argument is ignored while in a macro definition. Comments after the dummy argument list in a definition are legal.

If the list of dummy arguments cannot fit on a single line (that is, if the .DEFIN statement requires more than 72_{10} characters) it may be continued on the succeeding line or lines by the usage of the .ETC pseudo-op, as shown below.

```
     ⇥DEFIN⇥MACNME,ARG1,ARG2,ARG3   /comment
     ⇥.ETC  ⇥ARG4,ARG5   /argument continuation
                 .
                 .
                 .
     ⇥.DEFIN⇥MACNME
     ⇥.ETC   ⇥ARG1
     ⇥.ETC   ⇥ARG2
     ⇥.ETC   ⇥ARG3
     ⇥.ETC   ⇥ARG4
     ⇥.ETC   ⇥ARG5
```

4.2 MACRO BODY

The body of the macro definition follows the .DEFIN statement. Appearances of dummy arguments are marked and the character string of the body is stored, five characters per two words in the macro definition table, until the macro terminating pseudo-op .ENDM is encountered. Comments within the macro definition are not stored.

Dummy arguments may appear in the definition lines only as symbols or elements of an expression. They may appear in the label field, operation field, or address field. Dummy arguments may appear within a literal or they may be defined as variables. They will not be recognized if they appear within a comment.

The following restrictions apply to the usage of the .DEFIN, .ETC and .ENDM pseudo-ops:

a. If they appear in other than the operation field within the body of a macro definition, they will cause erroneous results.
b. If .ENDM or .ETC appears outside the range of a macro definition, it will be flagged as undefined.

If index register usage is desirable, it should be specified in the body of the definition, not in the argument string.

```
.DEFIN       XUSE,A,B,C
LAC A
DAC B,X
LAC C
.ENDM
```

If .ASCII or .SIXBT is used in the body of a macro, a slash (/) or number sign (#) must not appear as part of the text string or as a delimiter (use <57> to represent a slash and <43> to represent a number sign). Be careful when using a dummy argument name as part of the text string. For example,

```
.DEFIN       TEXT A
.SIXBT       ,A,
.SIXBT       .A.
.ENDM
```

followed by the macro call,

```
     TEXT XYZ
```

will generate the following code

```
     .SIXBT         ,XYZ,
     .SIXBT         .A.
```

In the first .SIXBT statement, A is recognized as a dummy argument resulting in the substitution of XYZ. In the second statement, A is not recognized as a dummy argument because the string delimiter, period, is itself a legal symbol constituent.

```
                Definition                          Comments

     ⇥.DEFIN  ⇥ MAC,A,B,C,D,E,F
     ⇥LAC     ⇥ A#
     ⇥SPA
     ⇥JMP     ⇥ B
     ⇥ISZ     ⇥ TMP  /E           E is not recognized as an argument
     ⇥LAC     ⇥ (C
     ⇥DAC     ⇥ D + 1
     ⇥F
     ⇥.ASCII  ⇥ E
     B=.
     ⇥.ENDM
```

4.3 MACRO CALLS

A macro call consists of the macro name, which must be in the operation field, followed by a list of real arguments separated by commas and terminated by one of the characters listed below.

a. space (␣)
b. tab (⇥)
c. carriage return (↵)

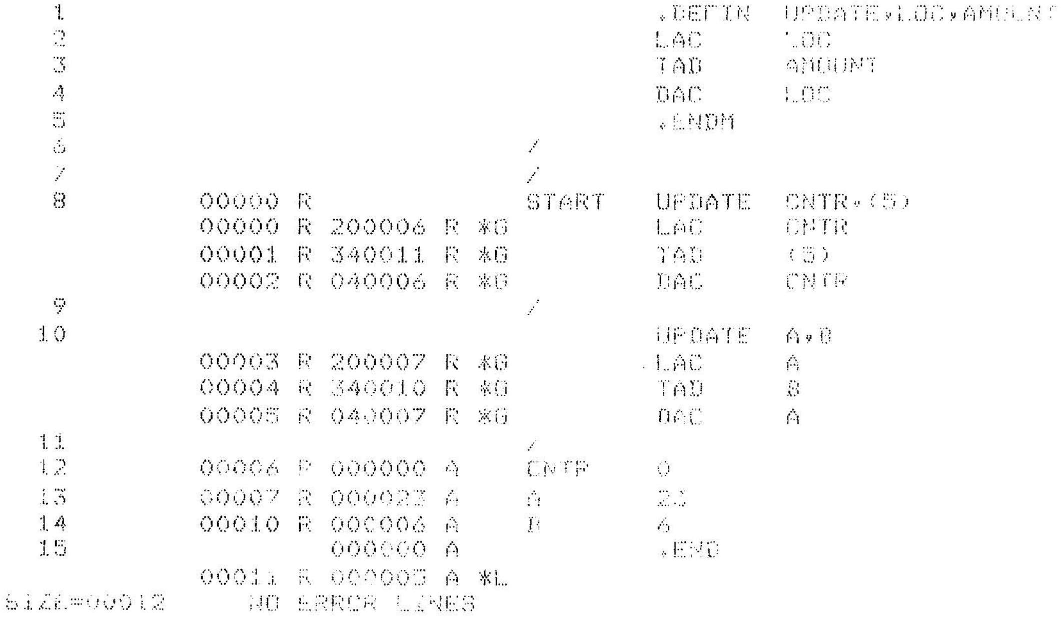
If the real arguments cannot fit on one line of coding, they may be continued on succeeding lines by terminating the current line with a dollar sign ($). When they are continued on succeeding lines they must start in the label field.

Example:

```
→|MAC→|REAL1,REAL2,REAL3,$
  REAL4,REAL5
```

If there are n dummy arguments in the macro definition, all real arguments in the macro call beyond the nth dummy argument will be ignored. A macro call may have a label associated with it; this label will be assigned to the current value of the location counter.

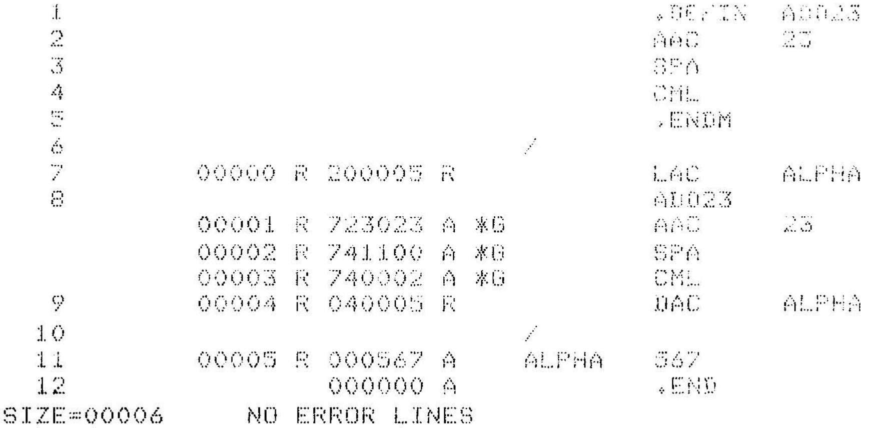
Example:



```
  1                                     .DEFIN  UPDATE,LOC,AMOUNT
  2                                     LAC     LOC
  3                                     TAD     AMOUNT
  4                                     DAC     LOC
  5                                     .ENDM
  6                            /
  7                            /
  8        00000 R             START    UPDATE  CNTR,(5)
           00000 R 200006 R *G          LAC     CNTR
           00001 R 340011 R *G          TAD     (5)
           00002 R 040006 R *G          DAC     CNTR
  9                            /
 10                                     UPDATE  A,B
           00003 R 200007 R *G          LAC     A
           00004 R 340010 R *G          TAD     B
           00005 R 040007 R *G          DAC     A
 11                            /
 12        00006 R 000000 A    CNTR     0
 13        00007 R 000023 A    A        23
 14        00010 R 000006 A    B        6
 15                000000 A             .END
           00011 R 000003 A *L
SIZE=00012     NO ERROR LINES
```

The prevailing radix will be saved prior to expansion and restored after expansion takes place. Default assumption will be octal for the macro call. It is not necessary for the macro definition to have any dummy arguments associated with it.

Example:



```
  1                                     .DEFIN  AD023
  2                                     AAC     23
  3                                     SPA
  4                                     CML
  5                                     .ENDM
  6                            /
  7        00000 R 200005 R             LAC     ALPHA
  8                                     AD023
           00001 R 723023 A *G          AAC     23
           00002 R 741100 A *G          SPA
           00003 R 740002 A *G          CML
  9        00004 R 040005 R             DAC     ALPHA
 10                            /
 11        00005 R 000567 A    ALPHA    567
 12                000000 A             .END
SIZE=00006     NO ERROR LINES
```

4.3.1 Argument Delimiters

It was stated that the list of arguments is terminated by any of the following symbols.

a. space (␣)

b. tab (→|)

c. carriage return (↵)

These characters may be used within real arguments only by enclosing them in angle brackets (<>). Angle brackets are not recognized if they appear within a comment.

Example:

```
    1                                          .DEFIN  MAC,A,B,C
    2                                          LAC     A
    3                                          TAD     B
    4                                          DAC     C
    5                                          .ENDM
    6
    7                                          MAC     XX,YY,ZZ
            00000 R 200007 R *G                LAC     XX
            00001 R 340010 R *G                TAD     YY
            00002 R 040011 R *G                DAC     ZZ
    8
    9                                          MAC     XX,<YY
   10                                          IAC>,ZZ
            00003 R 200007 R *G                LAC     XX
            00004 R 340010 R *G                TAD     YY
            00005 R 740030 A *G                IAC
            00006 R 040011 R *G                DAC     ZZ
   11
   12
   13
   14       00007 R 000123 A       XX          .DSA    123
   15       00010 R 000065 A       YY          .DSA    65
   16       00011 R 000000 A       ZZ          .DSA    0
   17               000000 A                   .END
 SIZE=00012       NO ERROR LINES
```

All characters within a matching pair of angle brackets are considered to be one argument, and the entire argument, with the delimiters (<>) removed, will be substituted for the dummy argument in the original definition.

The Assembler recognizes the end of an argument only on seeing a terminating character not enclosed within angle brackets.

If brackets appear within brackets, only the outermost pair is deleted. If angle brackets are required within a real argument, they must be enclosed by argument delimiter angle brackets.

Example:

```
  1                                          .DEFIN  ERRMSG,TEXT
  2                                          JMP     PRINT
  3                                          .ASCII  TEXT
  4                                          .ENDM
  5
  6                                          ERRMSG  <'ERROR IN LINE'<15>>
```

```
     U    00000 R 600007 R *G           JMP      PRINT
          00001 R 426452 A *G           .ASCII   'ERROR IN LINE'<15>
          00002 R 247644 A *G
          00003 R 202231 A *G
          00004 R 620230 A *G
          00005 R 446350 A *G
          00006 R 506400 A *G
   7                000000 A            .END
SIZE=00010        1 ERROR LINES
```

4.3.2 Created Symbols

Often, it is desirable to attach a label to a line of code within a macro definition. As this label is defined each time the macro is called, a different symbol must be supplied at each call to avoid multiply defined symbols.

This symbol can be explicitly supplied by the user or the user can implicitly request the Assembler to replace the dummy argument with a created symbol which will be unique for each call of the macro. For example,

```
→|.DEFIN→|MAC,A,?B
```

The question mark (?) prefixed to the dummy argument B indicates that it will be supplied from a created symbol if not explicitly supplied by the user when the macro is called for.

The created symbols are of the form ..0000→..9999. Like other symbols, they are entered into the symbol table as they are defined.

Unsupplied real arguments corresponding to dummy arguments not preceded by a question mark are substituted in as empty strings; and supplied real arguments corresponding to dummy arguments preceded by a question mark suppress the generation of a corresponding created symbol.

Example:

```
   1                                        .DEFIN   MAC,A,B,?C,?D,?E
   2                                        LAC      A
   3                                        SZA
   4                                        JMP      D
   5                                        LAC      B
   6                                        DAC      C#
   7                                        DAC      E
   8                                   D=.
   9                                        .ENDM
  10                                        MAC      Y#,,,,MYTAG
```

```
                  00000 R 200007 R *G             LAC       Y#
                  00001 R 740200 A *G             SZA
                  00002 R 600006 R *G             JMP       ..0003
                  00003 R 200000 A *G             LAC
                  00004 R 040010 R *G             DAC       ..0002#
                  00005 R 040006 R *G             DAC       MYTAG
                          000006 R *G ..0003=.
     11                               /
     12           00006 R 000000 A    MYTAG   0
     13                   000000 A            .END
   SIZE=00011      NO ERROR LINES
```

If one of the elements in a real argument string is not supplied, that element must be replaced by a comma, as in the call above. A real argument string may be terminated in several ways as shown below:

Example:

```
MAC      A,B, ␣
MAC      A,B,, ↵
MAC      A,B  ␣
MAC      A,B  ↵
MAC      A,B, ↵
```

4.3.3 Concatenation

If a dummy argument in a definition line of the macro body is delimited by the concatenation operation '@' and immediately preceded or followed by other characters or another dummy argument, the characters that correspond to the value of the dummy argument (real argument) are combined (juxtaposed) in the generated statement with the other characters or the real argument that corresponds to the other dummy argument. This process is called concatenation.

The following example illustrates this operation.

```
     1                                        .DEFIN  CALL,TYPE,ADDR
     2                                        JM@TYPE ADDR
     3                                        .ENDM
     4                               /
     5                                        CALL    P,ROUT1
           00000 R 600003 R *G                JMP     ROUT1
     6                               /
     7                                        CALL    S,<SUBRT1
     8                                        .DSA    ABC>
           00001 R 100004 R *G                JMS     SUBRT1
           00002 R 000005 R *G                .DSA    ABC
     9                               /
    10     00003 R 740040 A          ROUT1    XX
    11     00004 R 000000 A          SUBRT1   0
    12     00005 R 000000 A          ABC      0
    13             000000 A                   .END
  SIZE=00006      NO ERROR LINES
```

The dummy argument TYPE is used to vary the mnemonic operation code of the generated statement. The character P, which is the corresponding value of TYPE in the first call to the macro, will be concatenated with the characters JM to form the mnemonic JMP. This action occurs because a dummy argument (i.e., TYPE) is delimited by the concatenation operator (i.e., is preceded by @ and is immediately preceded or followed by other characters or another dummy argument).

Of course, in the case where other characters are to be concatenated with the value of a dummy argument, and the first of the other characters is a delimiter, it is not necessary to further delimit the dummy with the concatenation operator. The following example illustrates this rule.

```
1                                     .DEFIN  MOVE,FROM,TO,LVL
2                                     .IFZER SVC.@LVL
3                                     SKP
4                            SV.@LVL  .BLOCK 1
5                            SVC.@LVL=1
6                                     .ENDC
7                                     DAC     SV.@LVL
8                                     LAC     FROM@LVL,X
9                                     DAC     TO@LVL,X
10                                    LAC     SV.@LVL
11                                    .ENDM
12                000000 A   SVC.0=0          /MUST DEFINE 1 FOR EACH LEVEL
13                000000 A   SVC.1=0          / CALLED IN 'MOVE'.
14                           /
15                           /
16                                    MOVE    UST,RUST,0
                           *G         .IFZER SVC.0
          00000 R 741000 A *G         SKP
          00001 R        A *G SV.0    .BLOCK 1
                  000001 A *G SVC.0=1
                           *G         .ENDC
          00002 R 040001 R *G         DAC     SV.0
          00003 R 210020 R *G         LAC     USTO,X
          00004 R 050021 R *G         DAC     RUSTO,X
          00005 R 200001 R *G         LAC     SV.0
17                           /
18                                    MOVE    OPSTK,ARGST,0
                           *G         .IFZER SVC.0
                           *G         SKP
                           *G SV.0    .BLOCK 1
                           *G SVC.0=1
                           *G         .ENDC
          00006 R 040001 R *G         DAC     SV.0
          00007 R 210022 R *G         LAC     OPSTK0,X
          00010 R 050023 R *G         DAC     ARGST0,X
          00011 R 200001 R *G         LAC     SV.0
19                           /
20                                    MOVE    A,B,1
                           *G         .IFZER SVC.1
          00012 R 741000 A *G         SKP
          00013 R        A *G SV.1    .BLOCK 1
                  000001 A *G SVC.1=1
                           *G         .ENDC
          00014 R 040013 R *G         DAC     SV.1
          00015 R 210024 R *G         LAC     A1,X
          00016 R 050025 R *G         DAC     B1,X
          00017 R 200013 R *G         LAC     SV.1
```

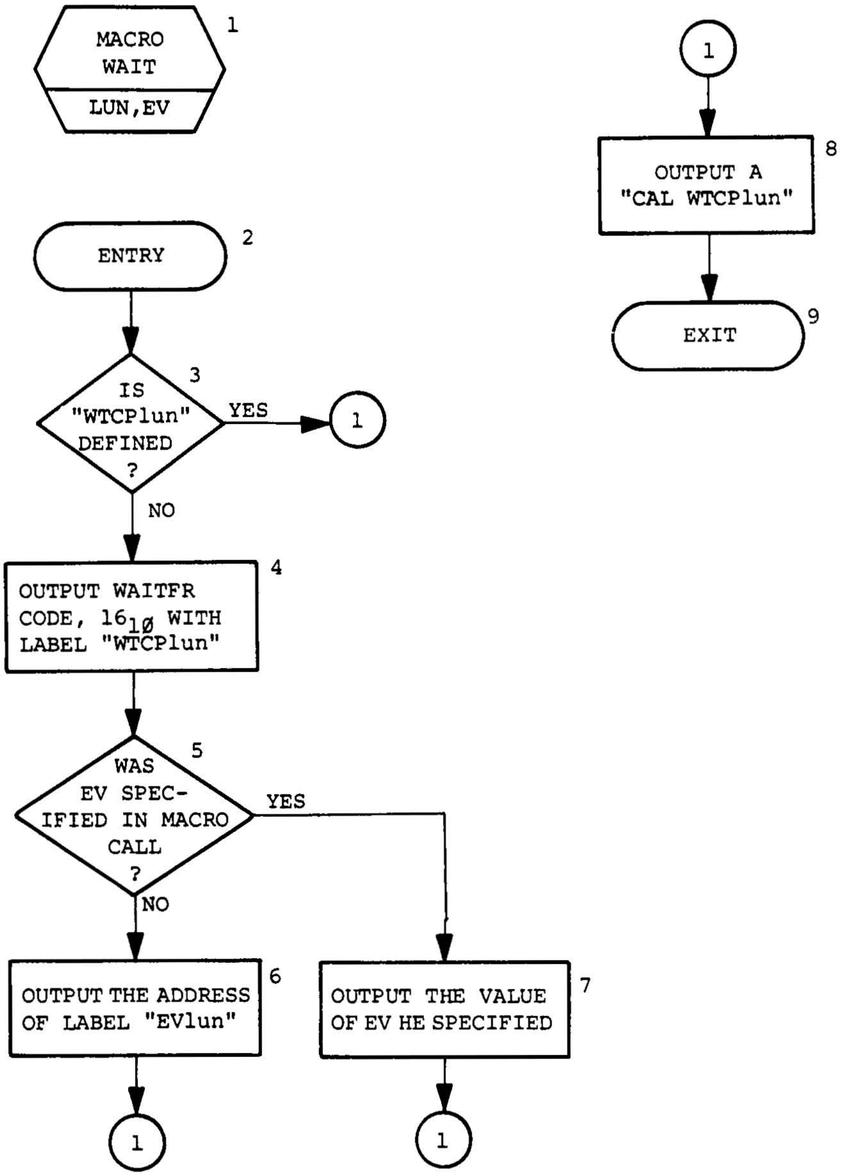
In this example concatenation is used to test the existence of a named temporary location, and, if necessary, output code to define it. Then the concatenation operator - Assembler delimiter rule is presented by concatenating two dummy arguments and other characters beginning with a delimiter. In detail, one such concatenation string is a delimiter (i.e.,→|), a dummy argument (i.e., FROM), the concatenation operator (i.e., @), a second dummy argument (i.e., LVL), finally followed by other characters beginning with a delimiter (i.e., ,X).

The general case of real argument for dummy argument substitution performed by MACRO is the application of the "other characters beginning with a delimiter" rule presented above. In other words, argument substitution may be thought of as concatenation when the dummy argument is bounded by delimiters, rather than a concatenation operator.

Note that one ambiguous case can arise in use of the concatenation operator when the other character string to be concatenated with an argument value is the same as a dummy argument name. The following example illustrates this problem.

```
               .DEFIN   WAIT,LUN,EV,TMP
               .DEC
               .IFUND   WTCP@LUN
               JMP      .+2
WTCP@LUN                16
TMP=EV+0
               .IFZER   TMP
               EV@LUN
               .ENDC
               .IFPNZ   TMP
               EV
               .ENDC
               .ENDC
               CAL      WTCP@LUN
               .ENDM
```

This macro was written with the intention of satisfying the following flow diagram.


MACRO
WAIT
LUN,EV
1
ENTRY
2
IS "WTCPlun" DEFINED ?
3
YES
1
NO
OUTPUT WAITFR CODE, $16_{1Ø}$ WITH LABEL "WTCPlun"
4
WAS EV SPEC-IFIED IN MACRO CALL ?
5
YES
NO
OUTPUT THE ADDRESS OF LABEL "EVlun"
6
OUTPUT THE VALUE OF EV HE SPECIFIED
7
1
1
1
OUTPUT A "CAL WTCPlun"
8
EXIT
9

For instance, if the following call to the WAIT macro were coded (with WTCP1Ø undefined):

```
                                      WAIT     10
                          *G          .DEC
                          *G          .IFUND   WTCP10
                          *G          JMP      .+2
                          *G WTCP10   16
                          *G ..0002=0
                          *G          .IFZER   ..0002
                          *G          10
                          *G          .ENDC
                          *G          .IFNZ    ..0002
                          *G
                          *G          .ENDC
                          *G          .ENDC
         00000 R 000001 R *G          CAL      WTCP10
```

Note that according to box 6 of the preceding flow chart, under these conditions it was desired to output:

    →|EV1Ø

for line 7 of the above expansion rather than what was actually generated. This discrepancy occurs because the characters EV on the appropriate line of the body of the definition are not recognized as "other characters". EV is also a dummy argument which is bounded by an Assembler delimiter (i.e.,→|on the left) and the concatenation operator (i.e., @ on the right). This will cause the concatenation of the value of dummy argument EV (i.e., null) and the value of the dummy argument LUN (i.e., 1Ø), thus producing the output shown on line 7 of the expansion. The only solution to this problem is to choose the names of dummy arguments to be different from any character strings to be used for concatenating.

Following is a comprehensive example of the use of the concatenation operation in defining user macros: the definition of two macros, ERRMSG and MESSAGE. The purpose of ERRMSG is to cause a subroutine to be called (named ER.PRO) which will print an error message.

It has as arguments the error number (from Ø to 77_8) and an optional return address. The label of the error message to be output is created by concatenating 'ERM.' with the error number. (ERM.Ø, ERM.1, etc.) If no return address is specified, control is transferred to a label named ER.NOR by default. The second macro, MESSAGE, is used to create an IOPS ASCII line buffer with the error message to be printed, presumably via the ERRMSG macro. It also has two arguments: the error number, and the message text. The output of the macro is a properly set up header word pair labeled 'ERM.xx' where 'xx' is the specified error number, and a .ASCII statement which contains the text specified, preceded by 'ERR#xx--', where 'xx' once again is the error number. The reader should examine the example noting the use of the conditional assembly parameters to accomplish macro-time error detection.

```
        .TITLE  CONCATENATION EXAMPLE FOR MACRO MANUAL
/
/MACRO 'ERRMSG' DEFINITION .  ERROR MESSAGE OUPUT MACRO.
/
/       CALLING SEQUENCE:
/
/       ERRMSG  ERRNO[,RETURN]
/
/       WHERE:
/       ERRNO = AN OCTAL NUMBER FROM Ø TO 77 REPRESENTING
/               THE ERROR CODE.
/       RETURN = (OPTIONAL) THE LOCATION TO WHICH CONTROL
/               SHOULD BE RETURNED FOLLOWING OUTPUT OF
/               THE ERROR MESSAGE.  IF NOT SPECIFIED,
/               CONTROL WILL BE GIVEN TO LOCATION 'ER.NOR'.
/
/       OUTPUT:
/
/       OUTPUT OF ERRMSG CONSISTS OF A JMS TO THE ERROR PROCESSOR
/       'ER.PRO', FOLLOWED BY A .DSA   ERM.XX WHERE XX = ERRNO.
/       ERM.XX IS ASSUMED TO BE A STANDARD IOPS ASCII LINE BUFFER
/       WHICH CONTAINS THE DESIRED MESSAGE.  IT MAY BE DEFINED USING
/       THE 'MESSAGE' MACRO (SEE BELOW).
/
/       ERROR DETECTION:
/
/       THE ERROR NUMBER ('ERRNO') IS CHECKED TO BE BETWEEN
/       Ø AND 77.  OTHERWISE AN ASSEMBLER ERROR LINE IS
/       OUTPUT RATHER THAN THE CALL TO 'ER.PRO'.  THE ILLEGAL
/       ASSEMBER LINE WILL CAUSE AN 'N' ERROR (AMONG OTHERS) TO BE
/       GENERATED BY THE ASSEMBER, THUS INDICATING A 'NUMBER'
/       ERROR.
/
        .DEFIN  ERRMSG,ERRNO,RTN
        .IFNEG  ERRNO-1ØØ               /VALIDATE ERROR CODE NUMBER
        .IFPOZ  ERRNO                   /TO BE Ø <= ERRNO <= 77
ZZRTNC=RTN+Ø                            /SETUP RETURN ADDR. IF SPECIFIED
        .IFZER  ZZRTNC
ZZRTNC=ER.NOR           /IF NO RETURN, SET TO STD. ADDR.
        .ENDC
        JMS     ER.PRO                  /CALL THE ERROR PROCESSOR
        .DSA    ERM.@ERRNO              /POINT TO RIGHT MESSAGE
        JMP     ZZRTNC                  /EITHER RETURN TO STD. EXIT, OR WHERE I SAID
        .ENDC
        .ENDC
        .IFNEG  ERRNO                   /PUT OUT ERROR IF NECESSARY
        9       **ERROR CODE IS < Ø OR > 77**
        .ENDC
        .IFPOZ  ERRNO-1ØØ
        9       **ERROR CODE IS < Ø OR > 77**
        .ENDC
        .ENDM

/MACRO 'MESSAGE' DEFINITION.  BUILD AN ERROR MESSAGE LINE BUFFER.
/
/       CALLING SEQUENCE:
/
/       MESSAGE ERRNO,<TEXT>
/
/       WHERE:
/       ERRNO = THE ERROR NUMBER, FROM Ø TO 77 (OCTAL)
/       <TEXT> = THE MESSAGE TEXT (ENCLOSED IN ANGLE
/               BRACKETS, AS SHOWN) TO BE ASSOCIATED WITH THIS
/               'ERRNO'.
/        OUTPUT:
/
/        A STANDARD IOPS ASCII LINE BUFFER IS CREATED WITH THE NAME
/        'ERM.XX' WHERE XX = 'ERRNO' (SEE ABOVE).   THE ACTUAL MESSAGE
/        WILL HAVE THE FORMAT 'ERR#XX-- TEXT '.  WHERE XX AND TEXT ARE AS
/        ABOVE.  OF COURSE, THE LINE BUFFER HEADER PAIR WILL BE PROVIDED.
/
/        ERROR DETECTION:
/
/        'ERRNO' WILL BE CHECKED TO BE BETWEEN  0 AND 77.
/        IF THE CHECK SHOWS AN ERROR, AN ASSEMBLER ERROR
/        LINE WILL BE GENERATED RATHER THE THE MESSAGE CODE.  THE ERROR
/        LINE WILL CAUSE AT LEAST AN 'N' FLAG, INDICATING A 'NUMBER'
/        ERROR.
/
         .DEFIN  MESSAGE,ERRNO,TEXT,?A
         .IFNEG  ERRNO-100
         .IFPOZ  ERRNO
ERM.@ERRNO       A-ERM.@ERRNO/2*1000+2
         0
         .ASCII  'ERR#@ERRNO--TEXT'<15>
A=.
         .ENDC
         .ENDC
         .IFNEG  ERRNO
         9       **ERROR CODE IS < 0 OR > 77**
         .ENDC
         .IFPOZ  ERRNO-100
         9       **ERROR CODE IS < 0 OR > 77**
         .ENDC
         .ENDM
         .EJECT
            ERRMSG   4        /OUTPUT ERROR MESSAGE #4, TAKE STANDARD EXIT
*G          .IFNEG   4-100
*G          .IFPOZ   4
*G ZZRTNC=+0
*G          .IFZER   ZZRTNC
*G ZZRTNC=ER.NOR
*G          .ENDC
*G          JMS      ER.PRO
*G          .DSA     ERM.4
*G          JMP      ZZRTNC
*G          .ENDC
*G          .ENDC
*G          .IFNEG   4
*G          9        **ERROR CODE IS < 0 OR > 77**
*G          .ENDC
*G          .IFPOZ   4-100
*G          9        **ERROR CODE IS < 0 OR > 77**
*G          .ENDC
            ERRMSG   45,RECOV         /GIVE ERROR #45, AND RETURN TO LOC 'RECOV' WHEN DONE
*G          .IFNEG   45-100
*G          .IFPOZ   45
*G ZZRTNC=RECOV+0
*G          .IFZER   ZZRTNC
*G ZZRTNC=ER.NOR
*G          .ENDC
*G          JMS      ER.PRO
*G          .DSA     ERM.45
*G          JMP      ZZRTNC
*G          .ENDC
*G          .ENDC
```

```
*G
*G
*G
*G ..ØØØ6=.
*G         .ENDC
*G         .ENDC
*G         .IFNEG  4
*G         9       **ERROR CODE IS < Ø OR > 77**
*G         .ENDC
*G         .IFPOZ  4-1ØØ
*G         9       **ERROR CODE IS < Ø OR > 77**
*G         .ENDC
           MESSAGE 45,<AMBIGUOUS USE OF A COMPILER KEYWORD>
*G         .IFNEG  45-1ØØ
*G         .IFPOZ  45
*G ERM.45  ..ØØØ9-ERM.45/2*1ØØØ+2
*G         Ø
*G         .ASCII  'ERR#45--AMBIGUOUS USE OF A COMPILER KEYWORD'<15>
*G
*G
*G
*G
*G
*G
*G
*G
*G
*G
*G
*G
*G
*G         .IFNEG  45
*G         9       **ERROR CODE IS < Ø OR > 77**
*G         .ENDC
*G         .IFPOZ  45-1ØØ
*G         9       **ERROR CODE IS < Ø OR > 77**
*G         .ENDC
           ERRMSG  -34,RECOV       /SHOW THAT A NEGATIVE ERROR NO. IT ILLEGAL
*G         .IFNEG  -34-1ØØ
*G         .IFPOZ  -34
*G ZZRTNC=RECOV+Ø
*G         .IFZER  ZZRTNC
*G ZZRTNC=ER.NOR
*G         .ENDC
*G         JMS     ER.PRO
*G         .DSA    ERM.-34
*G         JMP     ZZRTNC
*G         .ENDC
*G         .ENDC
*G         .IFNEG  -34
*G         9       **ERROR CODE IS < Ø OR > 77**
*G         .ENDC
*G         .IFPOZ  -34-1ØØ
*G         9       **ERROR CODE IS < Ø OR > 77**

*G         .ENDC
           ERRMSG  456     /SHOW THAT AN ERROR NO. > 77(8) IS ILLEGAL
*G         .IFNEG  456-1ØØ
*G         .IFPOZ  456
*G ZZRTNC=+Ø
*G         .IFZER  ZZRTNC
*G ZZRTNC=ER.NOR
```

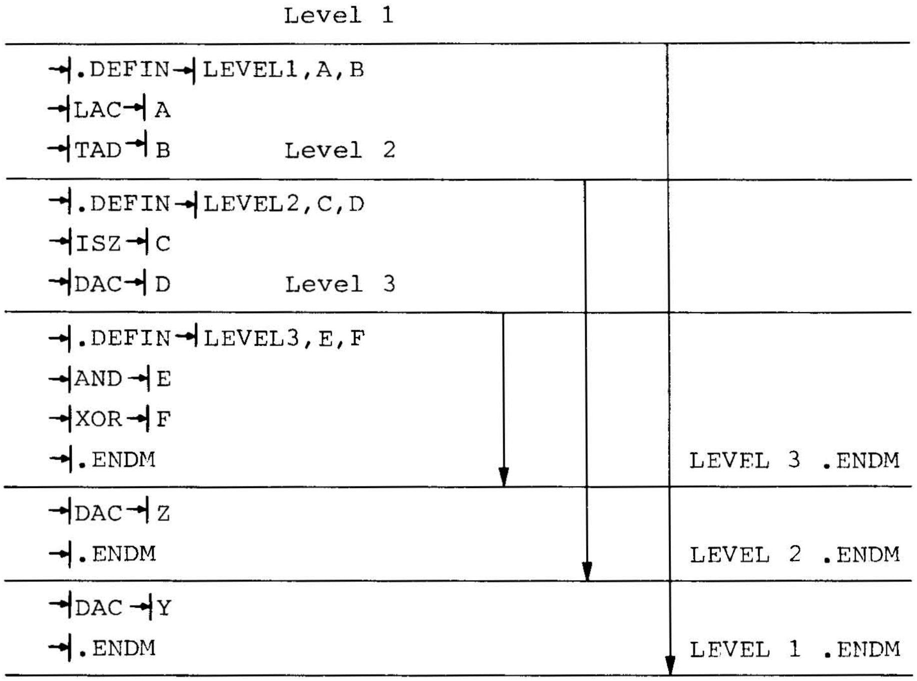
```
*G          .ENDC
*G          JMS       ER.PRO
*G          .DSA      ERM.456
*G          JMP       ZZRTNC
*G          .ENDC
*G          .ENDC
*G          .IFNEG    456
*G          9         **ERROR CODE IS < Ø OR > 77**
*G          .ENDC
*G          .IFPOZ    456-1ØØ
*G          9         **ERROR CODE IS < Ø OR > 77**
*G          .ENDC
            .EJECT

            MESSAGE 4,<ILLEGAL OR UNRECOGNIZABLE SYNTAX IN STMNT>
*G          .IFNEG    4-1ØØ
*G          .IFPOZ    4
*G ERM.4    ..ØØØ6-ERM.4/2*1ØØØ+2
*G          Ø
*G          .ASCII    'ERR#4--ILLEGAL OR UNRECOGNIZABLE SYNTAX IN STMNT'<15>
*G
*G
*G
*G
*G
*G
*G
*G
*G
*G
*G
*G
*G
*G
*G
*G
*G
*G
*G
*G
*G ..ØØØ9=.
*G          .ENDC
*G          .ENDC
*G          .IFNEG    45
*G          9         **ERROR CODE IS < Ø OR > 77**
*G          .ENDC
*G          .IFPOZ    45-1ØØ
*G          9         **ERROR CODE IS < Ø OR > 77**
*G          .ENDC
            MESSAGE -1,<THIS SHOULD GIVE A MACRO-DETECTED ERROR>
*G          .IFNEG    -1-1ØØ
*G          .IFPOZ    -1
*G ERM.-1   ..ØØ12-ERM.-1/2*1ØØØ+2
*G          Ø
*G          .ASCII    'ERR#-1--THIS SHOULD GIVE A MACRO-DETECTED ERROR'<15>
*G ..ØØ12=.
*G          .ENDC
*G          .ENDC
*G          .IFNEG    -1
*G          9         **ERROR CODE IS < Ø OR > 77**
*G          .ENDC
*G          .IFPOZ    -1-1ØØ
*G          9         **ERROR CODE IS < Ø OR > 77**
*G          .ENDC
            .EJECT
```

4.4 NESTING OF MACROS

Macros may be nested; that is, macros may be defined within other macros. For ease of discussion, levels may be assigned to these nested macros. The outermost macros (those defined directly) will be called first-level macros. Macros defined within first-level macros will be called second-level macros; macros defined within second-level macros will be called third-level macros, etc. Each nested macro requires an .ENDM pseudo-op to denote its termination.

Example:



```
→|.DEFIN→|LEVEL1,A,B
→|LAC→|A
→|TAD→|B
→|.DEFIN→|LEVEL2,C,D
→|ISZ→|C
→|DAC→|D
→|.DEFIN→|LEVEL3,E,F
→|AND→|E
→|XOR→|F
→|.ENDM
→|DAC→|Z
→|.ENDM
→|DAC→|Y
→|.ENDM
```

At the beginning of processing, first-level macros are defined and may be called in the normal manner. Second and higher level macros are not yet defined. When a first-level macro is called, all its second-level macros are defined. Thereafter, the level of definition is irrelevant and macros may be called in the normal manner. If the second-level macros contain third-level macros, the third-level macros are not defined until the second-level macros containing them have been called.

Using the example above, the following would occur:

```
                      LEVEL1   TAG1,TAG2        /CAUSES LEVEL2 TO BE DEFINED.
00000 R 200010 R *G   LAC      TAG1
00001 R 340011 R *G   TAD      TAG2
                 *G   .DEFIN   LEVEL2,C,D
                 *G   ISZ      C
                 *G   DAC      D
                 *G   .DEFIN   LEVEL3,E,F
```

```
                  *G          AND     E
                  *G          XOR     F
                  *G          .ENDM
                  *G          DAC     Z
                  *G          .ENDM
00002 R 040016 R *G           DAC     Y
                              LEVEL2  TAG3,TAG4          /CAUSES LEVEL3 TO BE DEFINED.
00003 R 440012 R *G           ISZ     TAG3
00004 R 040013 R *G           DAC     TAG4
                  *G          .DEFIN  LEVEL3,E,F
                  *G          AND     E
                  *G          XOR     F
                  *G          .ENDM
00005 R 040017 R *G           DAC     Z
                              LEVEL3  TAG5,TAG6
00006 R 500014 R *G           AND     TAG5
00007 R 240015 R *G           XOR     TAG6
```

If LEVEL3 is called before LEVEL2 it would be an error and the line would be flagged as undefined.

When a macro of level n contains another macro of the level n + 1, calling the level n macro results in the generation of the body of the macro into the user's program in the normal manner until the .DEFIN statement of the level n + 1 macro is encountered; the level n + 1 macro is then defined and does not appear in the user's program. When the definition of the level n + 1 is completed (.ENDM encountered), the Assembler continues to generate the level n body into the user's program until, or unless, the entire level n macro has been generated.

4.5 REDEFINITION OF MACROS

If a macro name, which has been previously defined, appears within another definition, the macro is redefined and the original definition is eliminated. For example,

```
          .DEFIN   INDXSV
          JMS      SAVE
          JMP      SAVXIT
SAVE      0
          LAC      10
          DAC      TMP1
          LAC      11
          DAC      TMP10
          JMP*     SAVE
SAVXIT=.
          .DEFIN   INDXSV
          JMS      SAVE
          .ENDM
          .ENDM
```

When the macro INDXSV is called for the first time, the subroutine calling sequence is generated and followed immediately by the subroutine itself. After the subroutine is generated, a .DEFIN that contains the name INDEXSV is encountered. This new macro is defined and takes the place of the original macro INDEXSV. All subsequent calls to INDXSV cause only the calling sequence to be generated. The original definition of INDXSV will not be removed until after the expansion is complete.

```
                                         INDXSV
00000 R 100002 R *G                      JMS      SAVE
00001 R 600010 R *G                      JMP      SAVXT
00002 R 000000 A *G SAVE                 0
00003 R 200010 R *G                      LAC      10
00004 R 040010 R *G                      DAC      TMP#
00005 R 200011 A *G                      LAC      11
00006 R 040011 R *G                      DAC      TMP1#
00007 R 620002 R *G                      JMP*     SAVE
        000010 R *G SAVXT=.
                 *G                      .DEFIN   INDXSV
                 *G                      JMS      SAVE
                 *G                      .ENDM
```

4.6 MACRO CALLS WITHIN MACRO DEFINITIONS

The body of a macro definition may contain calls for other macros which have not yet been defined. However, the embedded calls must be defined before a call is issued to the macro which contains the embedded call. Embedded calls are allowed only to three levels.

Example:

```
          .DEFIN   MAC1,A,B,C,D,E
          LAC      A
          TAD      B
          MAC2     C,D                 /EMBEDDED CALL.
          DAC      E
          .ENDM
          .DEFIN   MAC2,A,B            /DEFINITION OF EMBEDDED CALL.
          XOR      A
          AND      B
          .ENDM
          MAC1     TAG1,TAG2,(400),(777),TAG3
*G        LAC      TAG1
*G        TAD      TAG2
*G        MAC2     (400),(777)
*G        XOR      (400)
*G        AND      (777)
*G        DAC      TAG3
```

The call

causes generation of

4.7 RECURSIVE CALLS

Although it is legal, avoid making a macro definition containing an embedded call to itself because the expansion will cause more than three levels to occur.

Example:

```
.DEFIN  MAC,A,B,C
LAC     A
TAD     B
DAC     C
MAC     A,B,C               /RECURSIVE CALL.
.ENDM
```

When a call for MAC is encountered, the Assembler searches memory for the definition and expands it. Since there is another call for MAC contained within the definition, the Assembler goes back once again to obtain the definition; this process would never cease if more than three levels were allowed. A conditional assembly statement could be used, however, to limit the number of levels as in the following example.

Example:

```
A=0
B=3
        .DEFIN  MAC,C,D
        LAC     C
        DAC     D
A=A+1
        .IFNZR  B-A
        MAC     SAVE,TEMP           /RECURSIVE CALL.
        .ENDC
        .ENDM
```

Names and arguments of nested macros and arguments of embedded calls may be substituted and used with perfect generality.

```
           .DEFIN  MAC1,A,B,C,D
           LAC     A
           ADD     B
           DAC     C
           .DEFIN  D,E
           AND     A
           DAC     E
           .ENDM
           .ENDM



           .DEFIN  MAC2,M,N,O,P,Q,?R
           ISZ     M
           JMP     R
           MAC1    N,O,P,Q
   R=.
           .ENDM



           MAC2    COUNT,TAG1,TAG2,TAG3,MAC3
*G         ISZ     COUNT
*G         JMP     ..0005
*G         MAC1    TAG1,TAG2,TAG3,MAC3
*G         LAC     TAG1
*G         ADD     TAG2
*G         DAC     TAG3
*G         .DEFIN  MAC3,E
*G         AND     TAG1
*G         DAC     E
*G         .ENDM
*G ..0005=.
```

CHAPTER 5
OPERATING PROCEDURES

5.1 INTRODUCTION

Detailed descriptions of the assembler calling procedure, command string format, general operating procedures, and printouts are given in this chapter.

5.2 CALLING PROCEDURE

5.2.1 XVM/DOS

In the XVM/DOS systems, the MACRO Assembler is called by typing MACRO↵ after the Monitor's $ request. When the Assembler has been loaded, it identifies itself by typing:

```
MACRO XVM Vnxnnn or BMACRO XVM Vnxnnn
>                   >
```

on the teleprinter. The > character indicates that the Assembler is waiting for the user to type in a command string.

There are two differences between MACRO XVM (the Page Mode Assembler) and BMACRO XVM (the Bank Mode Assembler). MACRO XVM starts each assembly assuming page mode relocation (.DBREL implied) and BMACRO XVM assumes bank mode relocation (.EBREL implied). When program sizes exceed 4096, MACRO outputs the warning message "PROG 4K" in the assembly listing but BMACRO does not. This message will appear even if the program is assembled under influence of .EBREL. This warning message has no other effect; the program will be assembled and output will be produced anyway.

5.2.2 RSX/XVM

In the RSX systems, MACRO is invoked by typing in the Assembler's name and also the command string on the same line following the prompting message "TDV ". For example:

```
        TDV>MAC BLXR ← FILE↲
        MACRO XVM Vnxnnn
```

The Assembler identifies itself, as just shown, only if the R option is designated in the command. The RSX version of the Assembler is equivalent to BMACRO in that it assumes .EBREL to begin with and does not print "PROG>4K".

5.3 GENERAL COMMAND CHARACTERS

The following characters are frequently used in the entry and control of MACRO programs.

Character Printout

RUBOUT (Echoes \\)	delete single character
CTRL U (Echoes @)	delete current line
CTRL P (Echoes↑P)	a. If the input source is physically segmented so that all but the last segment end with .EOT or nothing, the Assembler will print out the message

EOT

when the end of a segment is reached. In XVM/RSX, the Assembler does not type any such message.

b. If the source is segmented in such a way that operator intervention is required to load another segment, MACRO will print

↑P

(MAC-↑P in XVM/RSX) and wait for the user to key in CTRL P (CTRL P↲ in XVM/RSX). Except in XVM/RSX, the user response will be printed also and the line will appear as

↑P↑P

In XVM/RSX if no other tape is to be loaded, terminate assembly by typing CTRL Q↲.

c. At the start of PASS 2 or PASS 3 if input is on paper tape or if the source is segmented on DECtape or Magtape with segments being read via the same .DAT slot, the Assembler will request a CTRL P response as above.

d. If the Assembler is not waiting for more input, or is not waiting to start the next pass, typing CTRL P causes the Assembler to restart at PASS 1. This is true for all systems except XVM/RSX.

CTRL D (Echoes ↑ D) If the user specifies the Teleprinter as the input parameter device, he can delimit the parameter code by typing CTRL D (↑D) (followed by ↵ with the XVM/RSX Monitor). MACRO responds with EOT. MACRO immediately begins assembling the program from the device assigned to .DAT-11 (LUN 15 with XVM/RSX).

5.4 COMMAND STRING

The command string format consists of a string of options, followed by a left arrow, followed by the program name(s), followed by a terminator.

```
options←filnml,filnm2,...
```

The following sections describe the rules for forming proper command strings and show typical assembly examples. The character terminating the command line has significance. Terminating the line with a carriage return will cause the Assembler to re-initialize itself to PASS 1 at completion of the assembly; the Assembler is thus ready to accept another command string. Terminating the command with an ALT MODE will cause a return to the monitor at the end of assembly. In the XVM/RSX systems these line terminators have a different meaning. Termination with carriage return causes TDV to be called; termination with ALT MODE does not. In either case, the Assembler exits after executing the command line. If a command string error occurs, the entire command must be retyped.

5.4.1 Program File Name

To the right of the back arrow in the command string, one or more program file names may be required, depending upon the options used and the type of I/O devices. Where several names are needed, they are separated by commas.

Program names are required for files which are to be input from or output to directoried devices. The two proper forms for a file name are

```
filnam␣ext
    or
filnam
```

where

filnam = 1 to 6 character name
ext = 1 to 3 character extension

These may be formed from any of the legal printing characters shown in Appendix A and may appear in any order.

If the file name extension is omitted, the Assembler assumes SRC in default. Following are examples of single name command strings.

Examples:

User Command String	Assembler Interpretation	
	Name	Extension
←␣ABCDEF␣1ØØ↵	ABCDEF	1ØØ
←AB␣Ø11	AB	Ø11
←A↵	A	SRC
←ABCDEFG↵	ABCDEF	G
←ABCDEFG␣H↵	ABCDEF	H
←ABC␣␣VIA↵	ABC	SRC

The last three examples illustrate how the Assembler interprets improperly formed file names. If the file name is longer than six characters but is not followed by a space, the seventh, eighth and nineth characters are used as the extension. If it is followed by a space, characters beyond the sixth and before the space are ignored. If two spaces follow the file name, the extension is assumed to be SRC. In general, if too many characters are given the excess characters are ignored.

The extension name of the main program is output (unless the O option is present) as a special code in the relocatable binary file. This enables programmers to easily identify different versions of the same program by merely assigning unique extension names. If the P- option is utilized, the Linking Loader and UPDATE print out the source file names, including extension.

Regardless of the source file extension, such as TEST ØØ1, the binary file extension will be either BIN, meaning relocatable binary, or ABS, meaning absolute binary.

5.4.2 Options

Assembler options direct the course of the assembly. They describe the types of input and output desired. Option characters are listed to the left of the back arrow. They may be listed in any order and are typically not separated one from the other (although commas and spaces, which are ignored, may be used as separators). Option characters which appear more than once and invalid characters are ignored.

Examples:

Command	Meaning
B←FILE↵	Assemble FILE SRC and produce a binary object file.
BLS←NAME↵	Assemble NAME SRC and produce a binary object file and an assembly listing followed by a symbol table listing.
← PROG␣Ø1X↵	Assemble PROG Ø1X producing no output except a list of assembly errors, if any, on the listing device assigned to .DAT -12 (LUN 16 in XVM/RSX).

The following table shows the action and the default of the options.

Option	Action	Default Action
A	Print symbols at end of PASS 2 in alphanumeric sequence on listing device.	Symbols are not printed in alphanumeric sequence.
B	Generate a binary file to DAT -13 with extension BIN or ABS, as required. (LUN 17 in RSX).	A binary file is not generated.
C	Program areas that fall between unsatisfied conditionals are not printed. It is not necessary to type the L option if this option is used.	All source lines are printed.
E	This option enables the user to have any errors occurring during assembly printed on the console printer in addition to the device assigned to .DAT -12 (LUN 16 in RSX). The L or N switch should be used with the E option. This option is particularly useful to users who assign non-printing devices to .DAT -12.	Assembly errors are not printed on the console printer.
F	Read macro definition file from .DAT -14 (LUN 18 in RSX) during PASS 1. Terminate input with .EOT or CTRL D if Teletype[1] (CTRL D↵ if RSX).	No macro definition file is processed.
G	Print only the source line of a macro expansion. It is not necessary to type the L option.	Generate printouts for macro expansions and expandable pseudo-ops (e.g., .REPT).

[1] Teletype is a registered trademark of the Teletype Corporation.

Option	Action	Default Action
H	The H-option is used in conjunction with the A, V, or S options. User symbols are normally printed horizontally at the end of PASS 2, four symbols to a line. If the H-option is used the symbols will be printed one to a line.	Print symbols four to a line.
I	Ignore .EJECT's. The .EJECT pseudo-op is treated as a comment.	Skip to head of form when .EJECT is encountered.
L	Generate a listing file on the requested output device, DAT -12. (LUN 16 in RSX). If the output device is directoried, then the listing file extension will be LST.	A listing file is not generated (see options N,C).
N	Number each source line (decimal). If this option is used, it is not necessary to type the L option.	Source lines are not numbered.
O	Causes the assembler to omit the source extension and the linking loader code 33 from the binary file. This option must be used when assembling programs in the DOS or RSX systems to be run in ADSS or B/F.	Loader code 33 is included in the binary output.
P	Before assembly begins, read program parameters from DAT -10 (LUN 20 in RSX). Terminate input with .EOT or CTRL D (if Teletype). The parameter file is read only once; for this reason, only direct assignments may be used.	No parameters, begin assembly immediately after command string termination.
R	Identify the Assembler version number, print END PASS 1 and END PASS 2, and print the error count on the teleprinter (RSX only).	These items are not printed in order to speed up batch processing.
S	Same as selecting both A and V.	Symbols are not printed. (If neither option V, S nor A is requested, symbols are not printed.)
T	The T option causes a "Table of Contents table to be generated during PASS 1. The table will contain the page number and text of all assembled .TITLE statements in the program.	A table of contents is not generated at the head of the assembly listing.
V	Print symbols at end of PASS 2 in value sequence on listing device.	Symbols are not printed in value sequence.

Option	Action	Default Action
X	At completion of PASS 2, PASS 3 is loaded to perform the cross-referencing operation. At completion of PASS 3 the Assembler will call in PASS 1 and 2, to continue assembling programs. If the command string was terminated by an ALT MODE, control will return to the Monitor at the end of assembly. Without the N option the user would obtain a cross reference which would be effectively useless since the source lines of the listing are not numbered. The N option is automatically entered if you enter L and X.	A cross-reference is not provided and PASS 3 is not called in.
Z	The Z option is related to the macro definition file option F. Z has no effect if F is not also specified. F and Z are used in combination when the main program is segmented into two parts. The first part containing instructions other than simply macro definitions, must be read both during PASS 1 and PASS 2. This is the function of the Z option.	The F option, if specified, causes the Macro definition file to be read only during PASS 1.

5.4.3 Multiple Filename Commands

In the general case a command may require up to three file names, depending upon the options specified, to produce a single binary output file. As will be illustrated later on, the Assembler in XVM/RSX systems allows multiple assemblies to be specified in a single command, which may require more than three file names. For the other software systems, the limit is three. Names may be needed to specify parameter files, macro definition files and program files. The use of these names and the manner in which they are interpreted by the Assembler are described in the following paragraphs.

NOTE

> In the following descriptions any file which is processed by both PASS 1 and PASS 2 of the Assembler is also processed during PASS 3 if the cross-reference option (X) is specified.

NAME 1: PARAMETER FILE

If the P option is used and the device assigned to .DAT slot -10 (LUN 20 in XVM/RSX) has a directory, the first name is interpreted as being the parameter file name. The name of the file must be explicitly stated if it is on a directoried device. If the device assigned to the parameter file is non-directoried, the first name typed would follow the rules for name 2. The parameter file is passed over only once during PASS 1.

If the P option is not used, only two names are accepted by the command string processor. The first name then would follow the rules for name 2.

NAME 2: MACRO DEFINITION FILE

If the F option is used, the second name (or the first if the P option is not used) is interpreted as being the macro definition file or part one of a two part program (assuming the device assigned to .DAT -14 (LUN 18 in RSX) has a directory). If the device is non-directoried, the second file name (or first if the P-option is not used or doesn't require one) would follow the rules for name 3. The macro definition is normally passed over only once, during PASS 1. However, unlike the main program file, macro definitions on .DAT slot -14 are recorded in core during PASS 1. Hence, PASS 2 is unnecessary. If the Z option is used with the F option this file will be passed over twice, allowing source files in two parts on two different devices. The Z-switch has no effect if F is not specified.

If the F option is not used, the first name (second if P option is used) is interpreted as the file name of the program to be assembled.

The macro definition file may also be used as an additional parameter file. A second parameter file is useful where a program is conditionally assembled to produce different versions according to many assembly parameters.

NOTE

> The RSX MACRO does not contain definitions of system directives and I/O calls. MACRO definitions or RSX are in a file called RMC.v SRC, where v changes with each release.

NAME 3: PROGRAM FILE NAME (Name of the Program to be Assembled)

This file is processed from .DAT slot -11 (LUN 15 in RSX) and always by both PASS 1 and PASS 2. If the P and F options are not used and multiple names are typed, only the first name will be processed. If a binary output file is requested, it will be directed to .DAT slot -13 (LUN 17 in RSX). If either of the two devices has a directory, a file name must be specified. The binary file will assume the name of the program file and an extension of either BIN or ABS.

MULTIPLE NAME INTERPRETATION

Before processing, the Assembler uses the .FSTAT function (SEEK in RSX) to determine whether or not the named files are on the input devices. If not, the message "NAME ERROR" is typed. In all but the RSX and BOSS XVM systems the Assembler then expects the command string to be retyped. In RSX, the Assembler exits and calls TDV so that the command string can be given to TDV. In BOSS XVM the Assembler exits to the monitor. Assuming that enough names have been typed to satisfy the command string options, MACRO interprets the file names as follows:

a. Current name = NAME 1.

b. Was the P option used? If not, go to step f.

c. Is the device assigned to .DAT slot -10 (LUN 20 in RSX) directoried? If not, go to step f.

d. Use the current name (NAME 1) to .SEEK the parameter file via .DAT slot -10 (LUN 20 in RSX).

e. Current name = NAME 2.

f. Was the F option used? If not, go to step j.

g. Is the device assigned to .DAT slot -14 (LUN 18 in RSX) directoried? If not, go to step j.

h. Use the current name (NAME 1 or NAME 2) to .SEEK the MACRO definition file via .DAT slot -14 (LUN 18 in RSX).

i. Current name = NAME 3 (or NAME 2 in P option not used).

j. Use the current name (NAME 1 or NAME 2 or NAME 3) to .SEEK the program file via .DAT slot -11 (LUN 15 in RSX).

RULES FOR MULTIPLE NAMES IN THE COMMAND STRING

1. Initial blanks positioned after the back arrow are ignored.

2. Files are processed sequentially. The first name after the left arrow is the first file read, the second file is next and so on.

3. Once a string of legal name characters is started, a space has the following effect on a name.

   a. The first space delimits the proper name and indicates to the command string processor that the extension name is next. The proper name is defined as the first six characters of a file name, excluding the extension.

   b. Two consecutive blanks delimit the name. An extension of 'SRC' is implied if no extension was typed.

4. A comma or line terminator delimits the name. (Same as 3b above.)

5. Any name given after the third name is ignored, except in XVM/RSX. The XVM/RSX assembler allows multiple assemblies to be specified in a single command. Where the options require one, two or three file names, the command may contain multiples of one, two or three. Each such group of one, two or three names represents a single assembly.

RESTRICTIONS CAUSED BY MULTIPLE FILE INPUT (not relevant to XVM/RSX)

The .FSTAT system macro is used by the MACRO Assembler to determine whether or not the input device has a directory and whether or not the argument names are on the assigned devices. For this reason, only those I/O handlers which honor or which ignore the .FSTAT function may be used with MACRO. The "A" handlers for directoried devices (e.g., DTA, DKA) honor .FSTAT. The paper tape punch and reader handlers ignore .FSTAT, but the effect is as if they accept it. Device handlers which treat .FSTAT as illegal may not be used.

5.4.4 Examples of Commands for Segmented Programs

Below are typical assembly situations which illustrate the usage of some of the assembly options and show the resulting teleprinter output. The output for XVM/RSX differs slightly from what is shown. That is explained in section 5.3.

1. Segmented Program on Paper Tape

   A source main program is segmented onto three paper tapes to make loading in the reader easier. Tapes one and two terminate with an .EOT statement and tape three terminates with .END. All three segments are read from the primary input, .DAT -11 (LUN 15 in RSX). The command to MACRO to produce a binary program is:

   ```
   >B ← ANYNAM↵
   ```

   Note that tape 1 must be ready in the reader before the command string is entered. Were it not, the reader would return an end of tape condition anyway and erroneous results would be obtained. The resulting teleprinter output is shown below. The comments to the right are not part of the output; these are included here as explanatory remarks. User responses are underlined.

```
>B ← ANYNAM↵
 EOT                      /End of tape 1.
↑P↑P                      /Ready tape 2.  Type CTRL P.
 EOT                      /End of tape 2.
↑P↑P                      /Ready tape 3.  Type CTRL P.
 END OF PASS 1
↑P↑P                      /Ready tape 1.  Type CTRL P.
 EOT                      /End of tape 1.
↑P↑P                      /Ready tape 2.  Type CTRL P.
 EOT                      /End of tape 2.
↑P↑P                      /Ready tape 3.  Type CTRL P.
 SIZE=Ø12Ø3    NO ERROR LINES
```

2. Segmented Program on DECtape

A source main program cannot fit onto a single DECtape. It is split in two on two different DECtapes and given the same file name: MAIN SRC. The tape one file ends with .EOT; the tape two file ends with .END. The file names must be identical if both segments are to be read via the primary input, .DAT -11 (LUN 15 in RSX). Example 3 illustrates an alternate method. However, example 2 must be used if one also is to include a MACRO definition file, as in example 4. The following command to MACRO produces a binary program and the subsequent teleprinter output:

```
>B ← MAIN
 EOT                      /End of file 1.  Mount second
↑P↑P                      /DECtape on same unit.  Type
                          /CTRL P.
 END OF PASS 1            /End of file 2.  Mount first
↑P↑P                      /DECtape on same unit.  Type
                          /CTRL P.
 EOT                      /End of file 1.  Mount second
↑P↑P                      /DECtape on same unit.  Type
                          /CTRL P.
 SIZE=ØØ7Ø3    NO ERROR LINES
```

3. Segmented Program on Disk

This example is a variation of number 2. A two part main program resides on disk. It doesn't matter whether the two files are on the same or separate disk units. Part one terminates with .EOT; part 2, with .END. PART1 SRC will be read via the secondary input, .DAT -14 (LUN 18 in RSX); and PART 2 SRC will be read via the primary input, .DAT -11 (LUN 15 in RSX). The resultant binary file, produced by the following command to MACRO, will assume the name of the second (primary) file: PART2 BIN or PART2 ABS, as the case may be:

```
>BFZ ← PART1,PART2↵
 EOT                      /End of PART1 SRC.
 END OF PASS 1            /End of PART2 SRC.
 EOT                      /End of PART1 SRC.
 SIZE=Ø2ØØ3    NO ERROR LINES
```

Several points can be made about the differences between examples 2 and 3. First, note that CRTL P type in is not required unless input is from a device like paper tape. Next, note that example 2 is impractical on disk because it requires physically interchanging disks. Example 3 is not restricted to usage with disk, but can be used with other media as well.

4. Use of a Macro Definition File

MACDEF SRC, which terminates with .EOT, contains only macro definitions. It is read from the secondary input, .DAT -14 (LUN 18 in RSX). The user has a main program, USEMAC 002, which terminates with .END and which calls some of these macros but does not itself define them. This is just an example. It is perfectly legal for the main program to redefine macros which also appear in the macro definition file. USEMAC ØØ2 is read from the primary input, .DAT -11 (LUN 15 in RSX). Below is the appropriate command string to produce a binary program. Note that the F option without the Z option (see example 3) instructs the Assembler to read the first file (the Macro definition file) only during PASS 1.

```
>BF ←MACDEF,USEMAC ØØ2↵
EOT                              /End of MACDEF SRC.
END OF PASS 1                    /End of USEMAC ØØ2.
SIZE=Ø11Ø4   NO ERROR LINES
```

Note that EOT is not printed during PASS 2 because MACDEF SRC is read only during PASS 1. The preceding example assumes that the files are on directoried devices.

5. Parameter File on Paper Tape

A main program, MAIN SRC, which terminates with .END is conditionalized to produce different binary code based on the values or existence of certain assembly parameters. It is read via the primary input, .DAT -11 (LUN 15 in RSX), which, for this example, is assigned to DECtape. A paper tape containing parameter definitions (direct assignments) terminates with .EOT and is read via the auxiliary input, .DAT -1Ø (LUN 2Ø in RSX). The following command to MACRO produces a binary program:

```
>BP← MAIN↵
EOT                              /End of parameter tape.
END OF PASS 1                    /End of MAIN SRC.
SIZE=ØØ6Ø2   NO ERROR LINES
```

Note, although input is partly from paper tape, a CTRL P response is unnecessary because the parameter tape is read only during PASS 1.

6. Multiple File Assemblies in XVM/RSX

Using the Assembler in XVM/RSX, several assemblies, using the same set of options for each, may be specified in a single command. Unless the R option is used, no printout on the teleprinter will occur to signal the various stages of assembly. Below are listed two typical commands in RSX.

```
>MAC BL←P1,P2␣ØØ3,P3,P4↵
```

This requests four assemblies. A separate binary and listing are produced for P1 SRC, P2 ØØ3, P3 SRC and P4 SRC.

```
>MAC PB←PAR1,FIL1,PAR2,FIL2↵
```

This requests two assemblies. A separate binary is produced for FIL1 SRC and FIL2 SRC. The parameter file PAR1 SRC is applied to the assembly of FIL1 SRC and PAR2 SRC to that of FIL2 SRC.

5.5 ASSEMBLY LISTINGS

If the user requests a listing via the command string, the Assembler will produce an output listing on the requested output device. The top of the first page of the listing will contain the name of the program as given in the command string. The body of the listing will be formatted as follows:

Line No.	Error Flags	Location	Address Mode	Object Code	Address Type	Line Type	Source	Statement
XXXX	XXX	XXXXX	R	XXXXXX	R A E	*G *L *R *E	X	X

where:

Line Number = Each source line and comment line is numbered (decimal); generated lines are not included. Lines are not numbered unless the X or N option is specified.

Flags = Errors encountered by the assembler

Location = Relative or absolute location assigned to the object code.

Address Mode = Indicates the type of user address.

A = absolute
R = relocatable

Line Type = *G = Generated *L = Literal
*R = Repeated *E = External

Object Code = The contents of the location (in octal)

Address Type = Indicates the classification of the object code.

A = absolute
R = relocatable
E = external

The object codes assigned for literals and external symbols are listed following the program.

5.6 SYMBOL TABLE OUTPUT

At the end of PASS 2, the symbol table may be output to the listing .DAT -12 (LUN 16 in RSX) device. If the A option is used, the table will be printed in alphanumeric sequence; if the V option is used, the symbol table will be printed in numeric value sequence; if the S option is used, the symbol table will be output in both alphanumeric and numeric sequence. The format is as follows:

```
Symbol          Value          Type

SYMBL1          XXXXX           E
SYMBL2          XXXXX           R
DIRECT         XXXXXX           A
```

The Xs represent the octal value assigned to the symbol. This is the location where the symbol is defined, except for external symbols. For these, the value is the location of the transfer vector, whose contents are set at program load time with the actual value of the symbol. Note that for SYMBL1 and SYMBL2 there are five Xs but that there are six Xs for the symbol DIRECT. Symbols having six octal numbers to represent their values are the result of direct assignments.

The symbol table shows the type of symbol:

A = absolute
R = relocatable
E = external

Locations assigned to variables immediately follow the last object code producing statement in the assembled program. Locations assigned for literals not under .LTORG influence and transfer vectors are listed immediately following the variables; if no variables are used in the program, literals and transfer vectors immediately follow the program output.

```
PAGE   1      SAMPLF SRC      SAMPLE PROGRAM

   1                                           .TITLE SAMPLE PROGRAM
   2                                    /
   3                                    / SAMPLE SUBROUTINE, NOT CLAIMED TO WORK OR TO HAVE ANY PRACTICAL
   4                                    / VALUE, USED TO ILLUSTRATE THE OUTPUT ON AN ASSEMBLY LISTING.
   5                                    / THESE LINES ARE COMMENTS.
   6                                    /
   7                                    / THIS LISTING WAS OBTAINED USING BMACRO-15 IN DOS-15 WITH THE
   8                                    / FOLLOWING COMMAND OPTIONS TO MACRO:  LSX
   9                                    /
  10                    000005 A        OUT=5                             /.DAT SLOT 5.
  11                                           .IODEV   OUT
  12                                           .GLOBL   PRINT,SAVE,RESTOR
  13                                    /
  14                                           .IFUND   WIDTH             /CONDITIONAL ASSEMBLY.
  15                                           .DEC
  16                                    WIDTH=72                          /DECIMAL NUMBER.
  17                                           .OCT
  18                                           .ENDC
  19                    000040 A        BUFSIZ=WIDTH+4/5*2+2              /DIRECT ASSIGNMENT.
  20                                    /
  21            00000 R 000000 A        PRINT  0                          /SUBROUTINE ENTRY POINT.
  22            00001 R 040116 R               DAC      ACSAV#            /VARIABLE.
  23            00002 R 200123 R               LAC      (SAVBUF)          /LITERAL.
  24            00003 R 120122 E               JMS*     SAVE              /EXTERNAL CALL.
  25            00004 R 220116 R               LAC*     ACSAV             /BUFFER ADDRESS.
  26            00005 R 741200 A               SNA
  27     U      00006 R 600117 R               JMP      NOBUFF            /UNDEFINED SYMBOL (MISSPELLED).
  28     F      00007 R 040003 A               DAC      WRITE+3           /UNDEFINED SYMBOL BECAUSE OF
  29            00010 R 723777 A               AAC      -1                /2 FORWARD REFERENCES.
  30            00011 R 060124 R               DAC*     (10)              /AUTOINDEX REGISTER.
  31            00012 R 777740 A               LAW      -BUFSIZ
  32            00013 R 040115 R               DAC      COUNT
  33            00014 R 735000 A               CLX
  34            00015 R 220010 A        LOOP   LAC*     10
  35            00016 R 050055 R               DAC      BUF,X             /INDEX REGISTER REFERENCE.
  36            00017 R 440115 R               ISZ      COUNT
  37            00020 R 600015 R               JMP      LOOP
  38            00021 R 600024 R               JMP      CHANGE
  39            00022 R 200125 R        NOBUF  LAC      (ERRMSG)
  40     U      00023 R 040123 R               DAC      WRIT+3            /UNDEFINED (MISSPELLED).
  41            00024 R 740000 A        CHANGE NOP
  42                                           .INIT    OUT,1,0           /SYSTEM MACRO CALL.
                00025 R 001005 A *G            CAL+1*1000 OUT&777
                00026 R 000001 A *G            1
                00027 R 000000 A *G            0+0
                00030 R 000000 A *G            0
  43            00031 R 200126 R               LAC      (JMP AROUND)
  44            00032 R 040024 R               DAC      CHANGE
  45                                    /
  46                                           .EJECT                     /PAGE EJECT.
```

```
PAGE    2      SAMPLF SRC      SAMPLE PROGRAM

  47                  000033 R     WRITE=AROUND                            /FORWARD REFERENCE.
  48           00033 R             AROUND   .WRITE  OUT,2,XX,0             /SYSTEM MACRO CALL.
               00033 R 002005 A *G          CAL+2*1000 OUT&777
               00034 R 000011 A *G          11
               00035 R 740040 A *G          XX
                                *G          .DEC
               00036 R 000000 A *G          -0
  49                               /
  50                                        .WAIT   OUT                    /SYSTEM MACRO CALL.
               00037 R 000005 A *G          CAL OUT&777
               00040 R 000012 A *G          12
  51           00041 R 200123 R             LAC     (SAVBUF)
  52           00042 R 120121 E             JMS*    RESTOR                 /EXTERNAL CALL.
  53           00043 R 200116 R             LAC     ACSAV
  54           00044 R 620000 R             JMP*    PRINT
  55                               /
  56                               / THE NEXT LINE CONTAINS THREE STATEMENTS.
  57                               /
  58           00045 R 003002 A    ERRMSG  003002; 0; .ASCII /ERROR/<15>
               00046 R 000000 A
               00047 R 426452 A
               00050 R 247644 A
               00051 R 064000 A
               00052 R 000000 A
  59           00052 R                      .LOC    .-1                    /CHANGE LOCATION COUNTER.
  60           00052 R        A    SAVBUF   .BLOCK  3                      /MQ, XR AND LR.
  61           00055 R        A    BUF      .BLOCK  BUFSIZ
  62           00115 R 000000 A    COUNT    0
  63                               /
  64                               / FOLLOWING THE .END STATEMENT ARE THREE LOCATIONS (NOT SHOWN)
  65                               / FOR ONE VARIABLE (ACSAV) AND TWO UNDEFINED SYMBOLS (NOBUFF
  66                               / AND WRITE, THE LATTER BECAUSE OF A DOUBLE FORWARD REFERENCE).
  67                               / FOLLOWING THAT (SHOWN) ARE TWO EXTERNAL TRANSFER VECTORS
  68                               / AND FOUR LITERALS.
  69                               /
  70                   000000 A             .END
               00121 R 000121 E *E
               00122 R 000122 E *E
               00123 R 000052 R *L
               00124 R 000010 A *L
               00125 R 000045 R *L
               00126 R 600033 R *L
                   SIZE=00130        3 ERROR LINES
```

```
              SAMPLE SRC      SAMPLE PROGRAM

ACSAV    ØØ116 R     AROUND   ØØØ33 R     BUF      ØØØ55 R     BUFSIZ ØØØØ4Ø A
CHANGE   ØØØ24 R     COUNT    ØØ115 R     ERRMSG   ØØØ45 R     LOOP     ØØØ15 R
NOBUF    ØØØ22 R     NOBUFF   ØØ117 R     OUT     ØØØØØ5 A     PRINT    ØØØØØ R
RESTOR   ØØ121 E     SAVBUF   ØØØ52 R     SAVE     ØØ122 E     WIDTH  ØØØ11Ø A
WRIT     ØØ12Ø R     WRITE   ØØØØ33 R

PRINT    ØØØØØ R     OUT     ØØØØØ5 A     LOOP     ØØØ15 R     NOBUF    ØØØ22 R
CHANGE   ØØØ24 R     AROUND   ØØØ33 R     WRITE   ØØØØ33 R     BUFSIZ ØØØØ4Ø A
ERRMSG   ØØØ45 R     SAVBUF   ØØØ52 R     BUF      ØØØ55 R     WIDTH  ØØØ11Ø A
COUNT    ØØ115 R     ACSAV    ØØ116 R     NOBUFF   ØØ117 R     WRIT     ØØ12Ø R
RESTOR   ØØ121 E     SAVE     ØØ122 E


              SAMPLE CROSS REFERENCE

ACSAV    ØØ116     22     25     53
AROUND   ØØØ33     43     47     48*
BUF      ØØØ55     35     61*
BUFSIZ  ØØØØ4Ø     19*    31     61
CHANGE   ØØØ24     38     41*    44
COUNT    ØØ115     32     36     62*
ERRMSG   ØØØ45     39     58*
LOOP     ØØØ15     34*    37
NOBUF    ØØØ22     29*
NOBUFF   ØØ117     27
OUT     ØØØØØ5     10*    11     42     48     5Ø
PRINT    ØØØØØ     12     21*    54
RESTOR   ØØ121     12     52
SAVBUF   ØØØ52     23     51     6Ø*
SAVE     ØØ122     12     24
WIDTH   ØØØ11Ø     14     16*    19
WRIT     ØØ12Ø     4Ø
WRITE   ØØØØ33     28     47*
```

5.7 RUNNING INSTRUCTIONS

Once the Assembler has identified itself, it is ready to perform an assembly. Proceed as follows:

a. Place the source program to be assembled on the appropriate input device.

b. Type the command string.

5.7.1 Paper Tape Input Only

The following steps are required when the source program is encountered in the paper tape reader:

a. At the end of a source tape segment which is not terminated with a .END statement of at the beginning of PASS 2 or PASS 3, the Assembler types

   ↑P

b. Place the proper source tape in the reader.

c. In XVM/DOS type CTRL P to continue. For RSX, type CTRL P ↵.

5.7.2 Cross-Reference Output

At the end of PASS 2, PASS 3 will be performed by the Assembler for the cross-referencing operation if the X option is requested. At completion, the assembler will be restarted (except in RSX systems) to permit additional assemblies if the command string is terminated by a CARRIAGE RETURN (↵) entry.

When a cross reference output is requested, the symbols are listed in alphabetic sequence. The first address after the symbol is the location where the symbol is defined or its 6-digit value if it is a direct assignment. All subsequent locations represent the line number (decimal) where the symbol was referenced. The line number with the asterisk is that in which the symbol is defined. Leading zeroes are suppressed for the cross-reference symbol table. Nine line numbers are printed on one line and subsequent line numbers are continued on the next line.

Example:

```
PAGE      1        PRGA       CROSS REFERENCE

A         1        XXXXX      XXXXX*.. ....XXXXX
                   XXXXX      XXXXX
B         5000     XXXXX*
SYMBOL    100      XXXXX*
```

Cross referencing can be a useful tool even without the aid of a line printer. It is possible to put the source assembly listing with line numbers onto a directoried device, such as DECtape, and the cross reference table (by a separate assembly) on a teleprinter. Then, desired lines in the "LST" file can be accessed by using the EDITOR.

LIMITATIONS

A. Before cross reference output can begin, PASS3 of the Assembler must first have read the entire source file(s) and stored the reference line numbers in core memory. Should available core be too limited, the Assembler will output the following message to the listing:

   CORE EXHAUSTED AT LINE DDDD

   where D is a decimal digit. Then the Assembler outputs all the references found up to that point.

B. For programs with more than 9999 lines of source code, line numbers begin again at ØØØØ on line 1ØØØØ. In the cross-reference listing, 1ØØØØ is represented as :ØØØ, 11ØØØ as ;ØØØ, and so on. These special characters are simply those which follow the numerals in the ASCII character set (Appendix A). Below is a list of characters and their meanings.

:	1Ø
;	11
<	12
=	13
>	14
?	15

C. To conserve core space, PASS3 of the Assembler does not maintain a permanent symbol table. Consequently, if user defined symbols are identical to permanent symbols, references to the permanent symbols will be included in the cross reference. For example:

```
     LAC A
     TAD LAC
     .
     .
     .
 LAC 5
```

Three references to LAC will be listed.

D. Conditionals (.IFxxx through .ENDC) are treated during PASS 3 as if they are always satisfied. Consequently, although a conditional might not be satisfied during PASS1 and PASS2, references within to defined user symbols will appear in the cross-reference output.

Note that undefined symbols which are referenced in .IFDEF and .IFUND statements remain undefined; hence, these do not appear in the cross reference.

5.8 PROGRAM RELOCATION

The normal output from the MACRO XVM Assembler is a relocatable object program, which may be loaded into any part of memory regardless of which locations are assigned at assembly time. To accomplish this, the address portion of some instructions must have a relocation constant added to it. This relocation constant is added at load time by the Linking Loader, CHAIN or TKB; it is equal to the difference between the memory location that an instruction is actually loaded into and the location that was assigned to it at assembly time. The Assembler determines which storage words are relocatable (marking them with an R in the listing), which are absolute (making these non-relocatable words with an A) and which are external (marking these with an E).

The rules that the Assembler follows to determine whether a storage word is absolute or relocatable are as follows:

a. If the address is a number (not a symbol), the address is absolute.

b. If the address is a symbol which is defined by a direct assignment (i.e., =) and the righthand side of the assignment is a number, all references to the symbol will be absolute.

c. If a user symbol is defined within a block of coding that is absolute, the value of that symbol is absolute.

d. Variables, undefined symbols, external transfer vectors, and literals get the same relocation as was in effect when .END was encountered in PASS 1.

e. If the location counter (.LOC pseudo-op) references a symbol which is not defined in terms of a relocatable address, the symbol is absolute.

f. All others are relocatable.

The following table depicts the manner in which the Assembler handles expressions which contain both absolute and relocatable elements.

(A=absolute, R=relocatable)

A+A=A	A-R=R	R+R=R and flagged as possible error
A-A=A	R+A=R	R-R=A
A+R=R	R-A=R	

If multiplication or division is performed on a relocatable symbol, it will be flagged as a possible relocation error.

If a relocatable program exceeds 4K, and the assembler is a page mode version, the following warning message will be typed at the end of PASS 2:

```
PROG > 4K
```

5.9 SYSTEM ERROR CONDITIONS AND RECOVERY PROCEDURES

5.9.1 XVM/DOS and BOSS XVM

See the XVM/DOS User's Manual, Appendix D or the XVM/DOS Keyboard Command Guide, Appendix C for descriptions of IOPS error messages.

5.9.2 XVM/RSX

Printout	Recovery Procedure
MAC-I/O ERROR LUN xx yyyyyy	is produced on LUN 3: xx represents the Logical Unit Number (decimal) and yyyyyy the octal Event Variable value indicating the cause of the error. See the XVM/RSX System Manual for the meaning of the error Event Variables. Control is automatically returned to TDV.

5.9.3 Restart Control Entries (DOS only)

CTRL P	Restart Assembler, if running
CTRL C	Return to Monitor

5.10 ERROR DETECTION BY THE ASSEMBLER

MACRO XVM examines each source statement for possible errors. The statement which contains the error will be flagged by one or several letters in the left-hand margin of the line, or, if the lines are numbered, between the line number and the location. The following table shows the error flags and their meanings.

Flag	Meaning
A	Error in direct symbol table assignment - assignment ignored
B	1. Memory bank error (program segment too large) 2. Page error - the location of an instruction and the address it references are on different memory pages (error in page mode only)
C	A .ENDC appears before an unsatisfied .IFxxx.
D	Statement contains a reference to a multiply-defined symbol - the first value is used.
E	1. Symbol not found in user's symbol table during PASS 2 2. Operator combined with its operand may produce erroneous results
F	Forward reference - symbol value is not resolved by PASS 2
I	Line ignored: 1. Relocatable pseudo-op in .ABS program 2. Redundant pseudo-op 3. .ABS pseudo-op in relocatable program 4. .ABS pseudo-op appears after a line has been assembled

5. A second .LOCAL pseudo-op appears before a matching .NDLOC pseudo-op
6. An .NDLOC appears without an associated .LOCAL pseudo-op
7. Too many .LTORG pseudo-ops (more than 8)
8. .IODEV pseudo-op in .ABS or .FULL program
9. Illegal statement within .CBS and .CBE

L Literal error:

1. Phase error - literal encountered in PASS 2 does not equal any literal found in PASS 1
2. Nested literal (a literal within a literal)

M Multiple symbol definition - first value defined is used

N Error in number usage (digit 8 or 9 used under .OCT influence)

P Phase error:

1. PASS 1 symbol value not equal to PASS 2 symbol value (PASS 2 value ignored)
2. A tag defined in a local area (.LOCAL pseudo-op is also defined in a non-local area

Q Questionable line:

1. Line contains two or more sequential operators (e.g., LAC A+*B)
2. Bad line delimiter - address field not terminated with a semicolon, carriage return or a comment
3. Bad argument in .REPT pseudo-op
4. Unrecognizable symbol with .ABS(P) pseudo-op

R Possible relocation error

S Symbol error - illegal character used in tag field

U Undefined symbol

W Line overflow during macro expansion

X Illegal use of macro name or index register

1. Unmatched .IFxxx and .ENDC
2. Unmatched .DEFIN and .ENDM
3. Unmatched .CBS and .CBE

In addition to flagged lines, there are certain conditions which will cause assembly to be terminated prematurely.

Message	Meaning
SYNTAX ERR	Bad command string, control returns to TDV (RSX only)
?	Bad command string, retype (not RSX)
NAME ERROR	File named in command string not found. In DOS, the Assembler will restart and accept another command string. RSX MACRO will return to TDV. BOSS will return to the Monitor.
TABLE OVERFLOW	Too many symbols and/or macros
CALL OVERFLOW	Too many embedded macro calls
CORE EXHAUSTED AT LINE nnn	PASS 3 error - too many symbol references

APPENDIX A

CHARACTER SET

Printing Character	7-bit ASCII	6-bit Trimmed ASCII	Printing Character	7-bit ASCII	6-bit Trimmed ASCII
@	100	00	Form Feed	014	
A	101	01	Carriage Return	015	
B	102	02	ALT MODE (ESC)	175	
C	103	03	Rubout	177	
D	104	04	(Space)	040	40
E	105	05	!	041	41
F	106	06	"	042	42
G	107	07	#	043	43
H	110	10	$	044	44
I	111	11	%	045	45
J	112	12	&	046	46
K	113	13	'	047	47
L	114	14	(	050	50
M	115	15	)	051	51
N	116	16	*	052	52
O	117	17	+	053	53
P	120	20	,	054	54
Q	121	21	-	055	55
R	122	22	.	056	56
S	123	23	/	057	57
T	124	24	0	060	60
U	125	25	1	061	61
V	126	26	2	062	62
W	127	27	3	063	63
X	130	30	4	064	64
Y	131	31	5	065	65
Z	132	32	6	066	66
[*	133	33	7	067	67
\	134	34	8	070	70
]*	135	35	9	071	71
↑*	136	36	:*	072	72
←*	137	37	;	073	73
Null	000		<	074	74
Horizontal Tab	011		=	075	75
Line Feed	012		>	076	76
Vertical Tab	013		?	077	77

*Illegal as source, except in a comment or text. Any characters not in this table are illegal to MACRO XVM and are flagged and ignored.

APPENDIX B

PERMANENT SYMBOL TABLE

Operate

OPR	740000
NOP	740000
CMA	740001
CML	740002
OAS	740004
RAL	740010
RAR	740020
IAC	740030
HLT	740040
XX	740040
SMA	740100
SZA	740200
SNL	740400
SML	740400
SKP	741000
SPA	741100
SNA	741200
SZL	741400
SPL	741400
RTL	742010
RTR	742020
SWHA	742030
CLL	744000
STL	744002
CCL	744002
RCL	744010
RCR	744020
CLA	750000
TCA	740031
CLC	750001
LAS	750004
LAT	750004
GLK	750010
LAW	760000

EAE

EAE	640000
LRS	640500
LRSS	660500
LLS	640600
LLSS	660600
ALS	640700
ALSS	660700
NORM	640444
NORMS	660444
MUL	653122
MULS	657122
DIV	640323
DIVS	644323
IDIV	653323
IDIVS	657323
FRDIV	650323
FRDIVS	654323
CLAC	641000
LACQ	641002
LACS	641001
CLQ	650000
ABS	644000
GSM	664000
OSC	640001
OMQ	640002
CMQ	640004
LMQ	652000

IOT

IOT	700000
IORS	700314
DRK	703304
DBR	703344
IOF	700002
ION	700042
CAF	703302
RES	707742

Memory Reference

CAL	000000
DAC	040000
JMS	100000
DZM	140000
LAC	200000
XOR	240000
ADD	300000
TAD	340000
XCT	400000
ISZ	440000
AND	500000
SAD	540000
JMP	600000

Automatic Priority Interrupt

RPL	705512
SPI	705501
ISA	705504

Index Instructions Which Take an Immediate Nine-bit Operand

AAC	723000
AXR	737000
AXS	725000

Mode Switching

EBA	707764
DBA	707762

Permanent Symbol Table

Index and Limit Register Instructions Which do not use Operands	
CLLR	736000
PAL	722000
PAX	721000
PLA	730000
PLX	731000
PXA	724000
PXL	726000
CLX	735000

Index Register Value	
X	10000

APPENDIX C

MACRO CHARACTER INTERPRETATION

Character		Function
Name	Symbol	
Space	␣	Field delimiter. Designated by ␣ in this manual.
Horizontal tab	⇥	Field delimiter. Designated by ⇥ in this manual.
Semicolon	;	Statement terminator
Carriage return	↵	Statement terminator
Plus	+	Addition operator (two's complement)
Minus	-	Subtraction operator (addition of two's complement)
Asterisk	*	Multiplication operator or indirect indicator
Slash	/	Division operator or comment initiator
Ampersand	&	Logical AND operator
Exclamation point	!	Inclusive OR operator
Back slash	\	Exclusive OR operator
Opening parenthesis	(	Initiate literal
Closing parenthesis	)	Terminate literal
Equals	=	Direct Assignment
Opening angle bracket	<	Argument delimiter
Closing angle bracket	>	Argument delimiter
Comma	,	An argument delimiter in macro definitions or an exclusive OR operator
Question mark	?	Created symbol designator in macros
Quotation mark	"	Text string indicator
Apostrophe	'	Text string indicator
Number Sign	#	Variable indicator
Dollar sign	$	Real argument continuation
Line feed	non-printing	not applicable
Form feed	non-printing	
Vertical tab	non-printing	
Commercial At	@	Concatenation operator in macro definitions

Macro Character Interpretation

Character		Function
Name	Symbol	
Null	Blank Character	Ignored by the Assembler
Delete	Blank Character	Ignored by the Assembler

Illegal Characters

Only those characters listed in the preceding table are legal in MACRO XVM source programs, all other characters will be ignored and flagged as errors. The following characters, although illegal as source, may be used within comment lines and in text preceded by .ASCII or .SIXBT pseudo-ops.

Character Name	Symbol
Left bracket	[
Right bracket	]
Up arrow	↑
Left arrow	←
Colon	:

APPENDIX D

SUMMARY OF MACRO XVM PSEUDO-OPS

Pseudo-op (Section)	Format	Function
.ABS (3.2.1) .ABSP (3.2.1)	⇥.ABS ⇥NLD↵ ⇥.ABSP⇥NLD↵	Object program is output in absolute, blocked, checksummed format for loading by the Absolute Binary Loader. Not supported in RSX.
.ASCII (3.3.1)	label*⇥.ASCII⇥/text/<octal>↵	Input text strings in 7-bit ASCII code, with the first character serving as delimiter. Octal codes for nonprinting control characters are enclosed in angle brackets.
.BLOCK (3.11)	label*⇥.BLOCK⇥exp↵	Reserves a block of storage words equal to the expression. If a label is used, it references the first word in the block.
.CBC (3.5.4)	⇥.CBC⇥displacement,value↵	Initialize a word of a common block to a constant.
.CBD (3.5.1)	label*⇥.CBD⇥NAME,/size↵	Sets up a COMMON area having the name and size specified. The first element in the COMMON area is also given (base address).
.CBE (3.5.5)	⇥.CBE↵	End of common block initialization section.
.CBDR (3.5.2)	label*⇥.CBDR⇥displacement↵	Enters the starting address of the last common block specified in a .CBD plus the argument into the location of the .CBDR.

[*] All pseudo-ops shown with a label generate binary output code.

Summary of MACRO XVM Pseudo-ops

Pseudo-op (Section)	Format	Function
.CBS (3.5.3)	⇥.CBS⇥name [, size]↵	Start common block initialization section.
.DBREL (3.2.3)	⇥.DBREL↵	Disable bank mode relocation.
.DEC (3.10)	⇥.DEC↵	Sets prevailing radix to decimal.
.DEFIN (3.4)	⇥.DEFIN⇥macroname,args↵	Defines macros.
.DSA (3.16)	label* ⇥.DSA␣exp↵	Generates a transfer vector for the specified symbol.
.EBREL (3.2.3)	⇥.EBREL↵	Enable bank mode relocation.
.EJECT (3.1.2)	⇥.EJECT↵	Skip to head of form on listing device.
.END (3.12)	⇥.END⇥start↵	Must terminate every source program. START is the address of the first instruction to be executed.
.ENDC (3.6)	⇥.ENDC↵	Terminates conditional coding in .IF statements.
.ENDM (3.4)	⇥.ENDM↵	Terminates the body of a macro definition.
.EOT (3.13)	⇥.EOT↵	Must terminate physical program segments, except the last, which is terminated by .END.
.ETC (3.4)	⇥.ETC⇥args,args↵	Used in macro definition to continue the list of dummy arguments on succeeding lines.
.FULL (3.2.2) .FULLP (3.2.2)	⇥.FULL ↵ ⇥.FULLP↵	Produces absolute, unblocked, unchecksummed binary object programs. Used only for paper tape output. Not supported in RSX.
.GLOBL (3.14)	⇥.GLOBL⇥sym,sym,sym↵	Used to declare all internal and external symbols which reference other programs.

*All pseudo-ops shown with a label generate binary output code.

Summary of MACRO XVM Pseudo-ops

Pseudo-op (Section)	Format	Function
.IFxxx (3.6)	→|.IFxxx→|exp↵	If a condition is satisfied, the source coding following the .IF statement and terminating with an .FNDC statement is assembled.
.IODEV (3.15)	→|.IODEV→|.DAT numbers↵	Specifies .DAT slots and associated I/O handlers required by this program. Not supported in RSX.
.LOC (3.9)	→|.LOC→|exp↵	Sets the location counter to the value of the expression.
.LOCAL (3.7)	→|.LOCAL↵	Allows deletion of certain symbols from the user symbol table.
.LST (3.1.3)	→|.LST↵	Continue requested assembly listing output of source lines. Lines between .NOLST and .LST are not listed.
.LTORG (3.8)	→|.LTORG↵	Allows the user to specifically state where literals are to be stored.
.NDLOC (3.7)	→|.NDLOC↵	Terminates deletion of certain symbols from the user symbol table contained between .LOCAL and .NDLOC.
.NOLST (3.1.3)	→|.NOLST↵	Terminates requested assembly listing output of source lines of code contained between .NOLST and .LST.
.OCT (3.10)	→|.OCT↵	Sets the prevailing radix to octal. Assumed at start of every program.
.REPT (3.17)	→|.REPT→|count,n↵	Repeats the object code of the next object code generating instruction Count times. Optionally, the generated word may be incremented by n each time it is repeated.

Summary of MACRO XVM Pseudo-ops

Pseudo-op Section)	Format	Function
.SIXBT (3.3.2)	label⇥.SIXBT⇥/text/<octal>↵	Input text strings in 6-bit trimmed ASCII, with first character as delimiter. Numbers enclosed in angle brackets are truncated to one 6-bit octal character.
.SIZE (3.18)	label⇥.SIZE↵	MACRO outputs the address of last location plus one occupied by the object program.
.TITLE (3.1.1)	⇥.TITLE⇥any text string↵	Causes the Assembler to accept up to 50_{10} typed characters. During source program assembly operations, a .TITLE causes a form feed code to be output to place the text starting with .TITLE at the top of a page.

APPENDIX E

SOURCE LISTING OF THE ABSOLUTE BINARY LOADER

```
           /***ABSOLUTE BINARY LOADER ***
           /      .FULL
7ØØØØ4     CLOF=7ØØØØ4
7ØØ112     RRB=7ØØ112
7ØØ144     RSB=7ØØ144
7ØØ1Ø1     RSF=7ØØ1Ø1
Ø1772Ø     LDSTRT=1772Ø
7Ø33Ø2     BINLDR CAF                 /CLEAR FLAGS
7ØØØØ4            CLOF                /CLOCK OFF
7ØØØ12            IOF+1Ø              /INTERRUPT OFF
7Ø55Ø4            ISA                 /TURN OFF API
74ØØØØ     LODMOD NOP                 /(EBA), (DBA), (NOP)
7Ø77Ø2            7Ø77Ø2              /PDP-9 COMPATIBILITY (EEM)
Ø17726     LDNXBK=17726
157775            DZM  LDCKSM         /CHECKSUMMING LOCATION
117753            JMS  LDREAD
Ø57776            DAC  LDSTAD         /GET STARTING ADDRESS
7411ØØ            SPA                 /BLOCK HEADING OR
617747            JMP  LDXFR          /START BLOCK
117753            JMS  LDREAD
Ø57777            DAC  LDWDCT         /WORD COUNT (2'S COMPLEMENT)
117753            JMS  LDREAD
Ø17736     LDNXWD=17736
117753            JMS  LDREAD
Ø77776            DAC* LDSTAD         /LOAD DATA INTO APPROPRIATE
457776            ISZ  LDSTAD         /MEMORY LOCATIONS
457777            ISZ  LDWDCT         /FINISHED LOADING
617736            JMP  LDNXWD         /NO
357775            TAD  LDCKSM
7402ØØ            SZA                 /LDCKSM SHOULD CONTAIN Ø
74ØØ4Ø            HLT                 /CHECKSUM ERROR HALT
617726            JMP  LDNXBK         /PRESS CONTINUE TO IGNORE
Ø17747     LDXFR=17747
Ø57777            DAC  LDWDCT
457777            ISZ  LDWDCT
617763            JMP  LDWAIT         /EXECUTE START ADDRESS
74ØØ4Ø            HLT                 /NO ADDRESS ON .END STATEMENT
Ø17753     LDREAD=17753               /MANUALLY START USER PROGRAM
```

Source Listing of the Absolute Binary Loader

```
000000          0
700144          RSB
357775          TAD  LDCKSM
057775          DAC  LDCKSM
700101          RSF
617757          JMP  LDREAD+4
700112          RRB
637753          JMP* LDREAD
       /THE LAST FRAME OF EVERY .ABS(P) PROG IS GARBAGE.
017763 LDWAIT=17763
117753          JMS  LDREAD         /PASS OVER LAST FRAME (PDP-9
637776          JMP* LDSTAD         /COMPATIBILITY).
006235 ENDLDR=.
003500 HRMWD   003500; 0            /HEADER
000000
000261          261;    277         /HRM START
000277
000320          320;    0
000000
017775 LDCKSM=17775
017776 LDSTAD=17776
017777 LDWDCT=17777
       /        .END BINLDR
       /*** END OF LOADER ***
```

INDEX

MACRO XVM Assembler Language Manual
DEC-XV-LMALA-A-D

Please cut along this line.

READER'S COMMENTS

NOTE: This form is for document comments only. Problems with software should be reported on a Software Problem Repcrt (SPR) form.

Did you find errors in this manual? If so, specify by page.

____
____
____
____
____
____

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

____
____
____
____
____
____

Is there sufficient documentation on associated system programs required for use of the software described in this manual? If not, what material is missing and where should it be placed?

____
____
____
____
____
____

Please indicate the type of user/reader that you most nearly represent.

- □ Assembly language programmer
- □ Higher-level language programmer
- □ Occasional programmer (experienced)
- □ User with little programming experience
- □ Student programmer
- □ Non-programmer interested in computer concepts and capabilities

Name ______________________ Date ______________

Organization ______________________________

Street ______________________________

City ______________ State __________ Zip Code ______________
or
Country

If you require a written reply, please check here. □

-- Fold Here --

-- Do Not Tear - Fold Here and Staple --

FIRST CLASS
PERMIT NO. 33
MAYNARD. MASS.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

Postage will be paid by:

digital

Software Communications
P. O. Box F
Maynard, Massachusetts 01754

digital

digital equipment corporation

PRINTED IN U.S.A.