decsystem10

MACRO ASSEMBLER REFERENCE MANUAL

AA-C780C-TB

April 1978

This document describes the language elements of the MACRO-10 Assembler for the DECsystem-10.

SUPERSESSION/UPDATE INFORMATION: This document supersedes the document of the same name, Order No. DEC-10-LMCOA-B-D, published July 1977

OPERATING SYSTEM AND VERSION: TOPS-10, Version 6.03

SOFTWARE VERSION: MACRO-10, Version 53

To order additional copies of this document, contact the Software Distribution Center, Digital Equipment Corporation, Maynard, Massachusetts 01754

digital equipment corporation · maynard, massachusetts

First printing,	April 1967
Revised,	August 1967
Revised,	June 1969
Revised,	October 1969
Revised,	August 1970
Revised,	April 1972
Revised,	November 1974
Revised,	July 1977
Revised,	April 1978

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 only be used or copied in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by DIGITAL or its affiliated companies.

Copyright © 1967, 1978 by Digital Equipment Corporation

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.

The following are trademarks of Digital Equipment Corporation:

DIGITAL	DECsystem-10	MASSBUS
DEC	DECtape	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EDUSYSTEM	PHA
UNIBUS	FLIP CHIP	RSTS
COMPUTER LABS	FOCAL	RSX
COMTEX	INDAC	TYPESET-8
DDT	LAB-8	TYPESET-11
DECCOMM	DECSYSTEM-20	

CONTENTS

CONTENTS (CONT.)

CONTENTS (CONT.)

CONTENTS (CONT.)

CONTENTS (CONT.)

Page

TABLES

PREFACE

This manual is a reference for the programmer with some knowledge of assemblers and assembly languages.

Using the MACRO assembler effectively involves using other DECsystem-10 facilities: the monitor (TOPS-10), the LINK program, the CREF program, a debugging program, a text editor (SOS or TECO), and machine language. Therefore the following DECsystem-10 documents will prove useful:

Operating System Commands
AA-0916C-TB

Monitor Calls
AA-0974C-TB

LINK Reference Manual
AA-0988C-TB

SOS User's Guide
DEC-10-USOSA-A-D

Introduction to TECO
DEC-10-UTECA-A-D

DDT Dynamic Debugging Techique
DEC-10-UDDTA-A-D

GALAXY Batch Reference Manual
DEC-10-OGBRA-A-DN1

Hardware Reference Manual
EK-10/20-HR-001

CHAPTER 1

INTRODUCTION TO MACRO

MACRO is the symbolic assembler program for the DECsystem-10. The assembler reads a file of MACRO statements and composes relocatable binary machine instruction code suitable for loading by LINK, the system's linking loader.

MACRO is a statement-oriented language; statements are in free format and are processed in two passes. In processing statements, the assembler:

1. Interprets machine instruction mnemonics
2. Accepts symbol definitions
3. Interprets symbols
4. Interprets pseudo-ops
5. Accepts macro definitions
6. Expands macros on call
7. Assigns memory addresses
8. Generates a relocatable binary program file (.REL file) for input to LINK
9. Generates a program listing file showing source statements, the corresponding binary code, and any errors found
10. Generates a UNIVERSAL file that can be searched by other assemblies

In addition to translating machine instruction mnemonics and special-purpose operators called pseudo-ops, MACRO allows you to create your own language elements, called macros. In this way you can tailor the assembler's functions for each program.

Since the assembler is device independent, you can use any peripheral devices for input and output files. For example, you can use a terminal for your source program input, a line printer for your program listing output, and a disk for your binary program output.

MACRO programs must use the monitor for device-independent input/output services. (See the Monitor Calls manual.)

NOTES

The following conventions are used throughout this manual:

1. All numbers in the examples are octal unless otherwise indicated.

2. All numbers in the text are decimal unless otherwise indicated.

3. The name of the assembler, MACRO, appears in uppercase letters; references to user-defined macros appear in lowercase letters.

4. Examples sometimes show the code generated as it appears in the program listing file. This file is described in Section 6.1.

1.1 HOW THE ASSEMBLER OPERATES

MACRO is a 2-pass assembler; it reads your source program twice. On Pass 1, some symbolic addresses will not be resolved, if they refer to parts of the program not yet read. These symbolic references are entered in the symbol table and will be resolved on Pass 2.

The main purpose of Pass 1 is to build symbol tables and to make a rudimentary assembly of each source statement.

The first task of Pass 1 is initializing all impure data areas that MACRO uses (internally) for assembly. This area includes all dynamic storage areas and all buffer areas.

MACRO then reads a command string into memory. This command string contains specifications for the files to be used during assembly. After scanning the command string for proper syntax, MACRO initializes the specified output files.

As assembly begins, MACRO initiates a routine that retrieves source lines from the proper input file. If no such file is currently open, MACRO opens the next input file specified in the command string. Source lines are assembled as they are retrieved from input files.

Assembly Pass 2 performs the same steps as Pass 1. However, during Pass 2 MACRO writes the object code to the binary (and usually relocatable) output file; it also generates the program listing file, followed by the symbol table listing for the program.

MACRO can also generate a cross-referenced symbol table. (See Chapter 6.)

During Pass 2 MACRO also flags erroneous source statements with single-character error codes. (See Chapter 7.) These error codes appear in the program listing file.

The relocatable binary object file created during Pass 2 contains all binary code generated; this code is in a form suitable for loading by the LINK program. (See the LINK Reference Manual.)

MACRO processes relocation counters on both passes. If a labeled statement has a different relocation value on the second pass, MACRO generates a phase error.

1.2 ADDRESSES AND MEMORY

The address space of a DECsystem-10 program consists of 256K (1K = 1024 words), each word having 36 bits. Since the total number of storage locations is 2 to the 18th power, the address of a location can be expressed in 18 bits, or one halfword.

The left halfword of a storage location is bits 0 to 17; the right halfword is bits 18 to 35.

1.3 RELOCATABLE ADDRESSES

Normally the binary program generated by MACRO is relocatable. This means that when the program is loaded for execution, it can be loaded anywhere in physical memory. (The address for loading is selected at load time, and depends on what has already been loaded.)

Unless you specify otherwise, MACRO assembles your binary program beginning with address 0 (400000 for high-segment code). References to addresses within your program are therefore relative to 0 (400000 for the high segment), and must be changed at loading time. LINK does this by adding the load address to all such relative addresses, resolving them to absolute addresses.

For programs assembled with multiple PSECT counters, each PSECT begins with the relative address 0. At load time, each PSECT has its own relocation constant; PSECT origins must be selected carefully to avoid overlapping of PSECTs in memory.

CHAPTER 2

ELEMENTS OF MACRO

The character set recognized in MACRO statements includes all ASCII alphanumeric characters and 28 special characters (ASCII 040 through 137). Lowercase letters (ASCII 141 through 172) are treated internally as uppercase letters (ASCII 101 through 132).

MACRO also recognizes seven ASCII control codes: horizontal tab (011), linefeed (012), vertical tab (013), formfeed (014), carriage-return (015), CTRL/underscore (037), and CTRL/Z (032).

MACRO accepts any ASCII character in quoted text, or as text arguments to the ASCII and ASCIZ pseudo-ops.

NOTES

1. The line-continuation character (CTRL/_) is always effective.

2. Delimiters for certain pseudo-ops (such as ASCII, ASCIZ, and COMMENT) can be any nonblank, nontab ASCII character.

Characters and their codes are listed in Appendix A.

A MACRO program consists of statements made up of MACRO language elements. Separated into general types, these are:

1. Special characters
2. Numbers
3. Literals
4. Symbols
5. Expressions
6. MACRO-defined mnemonics
7. Pseudo-ops
8. Macros

The format of a MACRO statement is discussed in Chapter 4.

2.1 SPECIAL CHARACTERS

Characters and combinations that have special interpretations in MACRO are listed in Appendix B. These interpretations apply only in the contexts described. In particular, they do not apply within comment fields or text strings.

2.2 NUMBERS

The two properties of numbers that are important to MACRO are:

1. In what radix (base) the number is given

2. How the number should be placed in memory

You can control the interpretation of these properties by using pseudo-ops or special characters to indicate your choices.

2.2.1 Integers

MACRO stores an integer in its binary form, right justified in bits 1 to 35 of its storage word. If you use a sign, place it immediately before the integer. (If you omit the sign, the integer is assumed positive.) For a negative integer, MACRO first forms its absolute value in bits 1 to 35, then takes its two's complement. Therefore a positive integer is stored with 0 in bit 0, while a negative integer has 1 in bit 0.

The largest integer that MACRO can store is 377777 777777 (octal); the smallest (most negative) is 400000 000000 (octal).

2.2.2 Radix

The initial implicit radix for a MACRO program is octal (base 8). The integers you use in your program will be interpreted as octal unless you indicate otherwise.

You can change the radix to any base from 2 to 10 by using the RADIX pseudo-op. (See the pseudo-op RADIX in Chapter 3.) The new radix will remain in effect until you change it.

Without changing the prevailing radix, you can write a particular expression in binary, octal, or decimal. To do this, prefix the integer with ^B for binary, ^O for octal, or ^D for decimal. The indicated radix applies only to the single integer immediately following it.

NOTES

1. A single-digit number is always interpreted as radix 10. Thus 9 is seen as decimal 9, even if the current radix is 8.

2. In the notations for ^B, ^D, and ^O, the up-arrow in the text indicates the up-arrow character, not the CONTROL character.

For example, suppose the current radix is 8. Then you can write the decimal number 23 as:

27	octal (current radix)
^D23	decimal
^B10111	binary

But you cannot write decimal 23 as ^D45-22 since the ^D applies only to the first number, 45; the 22 is octal. However, you can write decimal 23 as ^D<45-22>.

2.2.3 Adding Zeros to Integers in Source Code

You can add zeros to an integer (multiply it by a constant) in your program by suffixing K, M, or G to it.

K adds 3 zeros	(K = "kilo-", thousands)
M adds 6 zeros	(M = "mega-", millions)
G adds 9 zeros	(G = "giga-", billions)

These zeros are suffixed before any conversion, so that in radix 10, 5K means 5000 decimal; in radix 8, 5K means 5000 octal, or 2560 decimal.

2.2.4 Fixed-Point Decimal Numbers

To indicate a fixed-point decimal number, prefix it with ^F, include a decimal point wherever you wish, and suffix Bn to show that you want to place the "assumed point" after bit n in the storage word. If you omit the decimal point, MACRO assumes that it follows the last digit. If you omit the Bn, MACRO assumes B35.

To handle the number, MACRO forms the integer part in a fullword register, and the fractional part in another fullword register. It then places the integer part (right justified) in bits 1 to n (n is from your Bn) of a binary word, and the fractional part (left justified) in the remaining bits. The integer part is truncated at the left, and the fractional part at the right. Bit 0 shows the sign of the number.

For example, ^F123.45B8 is formed in two registers as

```
000000 000173   (integer part, right justified)

346314 631462   (fractional part, left justified)
```

Since the Bn operator sets the assumed point after bit 8, the integer part is placed in bits 1 to 8, and the fractional part in bits 9 to 35. (The sign bit 0 is 0, showing a positive number.) Truncation is on the left and right, respectively, giving

```
173,346 314631
   ^
   |
assumed point
```

You can show a fixed-point decimal number as negative by placing a minus sign before the ^F. The absolute value of the negative number is formed in two registers as a positive number, then two's complemented. This complementing sets bit 0 to 1, showing that the number is negative.

NOTE

The binary number resulting from ^F does not show where the assumed point should be. You must keep track of this through your own programming conventions.

Examples:

000000	000173	^F123.45
000173	346314	^F123.45B17
346314	631462	^F123.45B-1
777777	777604	-^F123.45
777604	431463	-^F123.45B17
431463	146316	-^F123.45B-1

2.2.5 Floating-Point Decimal Numbers

In your program, a floating-point decimal number is a string of digits with a leading, trailing, or embedded decimal point and an optional leading sign. MACRO recognizes this as a mixed number in radix 10.

MACRO forms a floating-point decimal number with the sign in bit 0, a binary exponent in bits 1 to 8, and a normalized binary fraction in bits 9 to 35.

The normalized fraction can be viewed as follows: its numerator is the binary number in bits 9 to 35, whose value is less than 2 to the 28th power, but greater than or equal to 2 to the 27th power. Its denominator is 2 to the 28th power, so that the value of the fraction is always less than 1, but greater than or equal to 0. (This value is 0 only if the entire stored number is 0.)

The binary exponent is incremented by 128 so that exponents from -128 to 127 are represented as 0 to 255.

For a negative floating-point decimal number, MACRO first forms its absolute value as a positive number, then takes the two's complement of the entire word.

Examples:

The floating-point number 17. generates the binary

```
0   10 000 101   100 010 000 000 000 000 000 000 000
```

where bit 0 shows the positive sign, bits 1 to 8 show the binary exponent, and bits 9 to 35 show the proper binary fraction. The binary exponent is 133 (decimal), which after subtracting the added 128 gives 5. The fraction is equal to 0.53125 decimal. And 0.53125 times 2 to the 5th power is 17, which is the number given.

Similarly, 153. generates

```
0   10 001 000   100 110 010 000 000 000 000 000 000
```

while -153. generates

```
1   01 110 111   011 001 110 000 000 000 000 000 000
```

These two examples show that a negative number is two's complemented. Notice that since the binary fraction for a negative number always has some nonzero bits, the exponent field (taken by itself) appears to be one's complemented.

As in FORTRAN, you can write a floating-point decimal number with a suffixed E±n, and the number will be multiplied by 10 to the ±nth power. If the sign is missing, n is assumed positive.

Examples:

```
2840000.        can be written   284.E+4
2840000.        can be written   .284E7
     .0000284   can be written   .284E-4
     .0000284   can be written   284.E-7
```

Using this E notation with an integer (no decimal point) is not allowed, and causes an error. Therefore you can use 284.E4, but 284E4 is illegal.

NOTE

MACRO's algorithm for handling numbers given with the E notation is not identical to FORTRAN's. The binary values generated by the two translators may differ in the lowest order bits.

2.2.6 Binary Shifting

Binary shifting of a number with Bn sets the location of the rightmost bit at bit n in the storage word, where n is a decimal integer. The shift takes place after the binary number is formed. Any bits shifted outside the range (bits 0 to 35) of the storage word are lost.

For example, here are some numbers with their binary representations given in octal:

300000	000000	^D3B2
000000	042000	^D17B25
000001	000000	1B17
400000	000000	1B0
777777	777777	-1B35
000000	000001	1B35
000000	777777	-1B35

2.2.7 Underscore Shifting

You can also shift a number by using the underscore operator. (On some terminals this is a left-arrow.) If V is an expression with value n, suffixing _V to a number shifts it n bits to the left. (If n is negative, the shift is to the right.)

In an expression of the form W_V, W and V can be any expressions including symbols. The binary value of W is formed in a register, V is evaluated, and the binary of W is shifted V bits when placed in storage.

NOTE

> An expression such as -3.75E4_^D18 is legal, but the shift occurs after conversion to floating-point decimal storage format. Therefore the sign, exponent, and fraction fields are all shifted away from their usual locations. This is true also for other storage formats.

2.2.8 Querying the Position of a Bit Pattern

You can query the position of a bit pattern by prefixing ^L (up-arrow L) to an expression. This generates the number of leading zeros in the binary value of the expression. (^L0 generates 36 decimal.)

For example, suppose the current radix is 10. Then

```
^L153      generates  35  (29 decimal)

^L153.     generates  1

^L-153     generates  0

^L-153.    generates  0
```

In the first example, ^L153 generates 29 (decimal) because the binary representation of 153 decimal has its leftmost 1 in bit 28:

```
000 000 000 000 000 000 000 000 000 010 011 001
```

But in the second example, the binary form of 153. is in floating-point format (see Section 2.2.5),

```
010 001 000 100 110 010 000 000 000 000 000 000
```

and its leftmost 1 is in bit 1.

In both of the last two examples, ^L-153 and ^L-153. generate 0. This is because a negative number in any format sets bit 0 to 1.

2.3 LITERALS

A literal is a character string within square brackets inserted in your source code. MACRO stores the code generated by the enclosed string in a literal pool beginning with the first available literal storage location, and places the address of this location in place of the literal. The literal pool is normally at the end of the binary program. (See the pseudo-op LIT in Chapter 3.)

The statements

```
135 01 0 00 002016'              LDB T1,[POINT 6,.JBVER,17]
                                 LIT
22 06 0 00 000137
```

are equivalent to

```
135 01 0 00 002020'              LDB T1,PLACE
. . .
22 06 0 00 000137       PLACE:   POINT 6,.JBVER,17
```

A literal can also be used to generate a constant:

```
        PUSH 17,[0]             ;Generate zero fullword

        MOVE L,[3,,1]           ;Generate a word with 3 in
                                ; lefthalf and 14 in righthalf
```

Multiline literals are also allowed:

```
GETCHR: ILDB T2,T1                  ;Get a character
        CAIN T2,0                   ;Is it a null?
        JRST [MOVE T1,TXTPTR        ;Yes, retrieve pointer
              ILDB T2,T1            ;Get a new character
              CAIN T2,"?"           ;Is it a question mark?
              JRST [MOVE T1,TXTPT1      ;Yes, set alternate pointer
                    ILDB T2,T1          ;Get the message character
                    JRST GETHLP]        ;Go to help routine
              POPJ P,]              ;Not question mark, return
        POPJ P,                     ;Not a null, return
```

The text of a literal continues until a matching closing square bracket is found (unquoted and not in comment field).

A literal can include any term, symbol, expression, or statement, but it must generate at least one but no more than 99 words of data. A statement that does not generate data (such as a direct-assignment statement or a RADIX pseudo-op) can be included in a literal, but the literal must not consist entirely of such statements.

You can nest literals up to 18 levels. You can include any number of labels in a literal, but a forward reference to a label in a literal is illegal.

If you use a dot (.) in a literal to retrieve the location counter, remember that the counter is pointing at the statement containing the literal, not at the literal itself.

In nested literals, a dot location counter references a statement outside the outermost literal.

In the sequence

```
JRST [HRRZ AC1,V
      CAIE AC1,OP
      JRST .+1
      JRST EVTSTS]
SKIPE C
```

the expression .+1 generates the address of SKIPE C, not JRST EVTSTS.

Literals having the same value are collapsed in MACRO's literal pool. Thus for the statements:

```
PUSH P,[0]
PUSH P,[0]
MOVEI AC1,[ASCIZ /TEST1/]
```

the same address is shared by the two literals [0], and by the null word generated at the end of [ASCIZ /TEST1/]. Literal collapsing is suppressed for those literals that contain errors, undefined expressions, or EXTERNAL symbols.

2.4 SYMBOLS

MACRO symbols include:

1. MACRO-defined pseudo-ops (discussed in Chapter 3)
2. MACRO-defined mnemonics (discussed in Section 2.6)
3. User-defined macros (discussed in Chapter 5)
4. User-defined opdefs (discussed at OPDEF in Chapter 3)
5. User-defined labels (discussed in this section)
6. Direct-assignment symbols (discussed in Section 2.4.2.2)
7. Dummy-arguments for macros (discussed in Chapter 5)

MACRO stores symbols in three symbol tables:

1. Op-code table: machine instruction mnemonics and pseudo-ops
2. Macro table: macros, user-defined OPDEFs, and synonyms (See the SYN pseudo-op in Chapter 3.)
3. User symbol table: labels and direct-assignment symbols

An entry in one of these tables shows the symbol, its type, and its value.

Symbols are helpful in your programs because:

1. Defining a symbol as a label gives a name to an address. You can use the label in debugging or as a target for program control statements.
2. In revising a program, you can change a value throughout your program by changing a symbol definition.
3. You can give names to values to make computations clearer.
4. You can make values available to other programs.

2.4.1 Selecting Valid Symbols

Follow these rules in selecting symbols:

1. Use only letters, numerals, dots (.), dollar signs ($), and percent signs (%). MACRO will consider any other character (including a blank) as a delimiter.
2. Do not begin a symbol with a numeral.
3. If you use a dot for the first character, do not use a numeral for the second. Do not use dots for the first two characters; doing so can interfere with MACRO's created symbols. (See Section 5.5.2.)
4. Make the first six characters unique among your symbols. You can use more than six characters, but MACRO will use only the first six.

Examples:

VELOCITY	(legal, only VELOCI is meaningful to MACRO)
CHG.VEL	(legal, only CHG.VE is meaningful to MACRO)
CHG VEL	(illegal, looks like two symbols to MACRO)
1STNUM	(illegal, begins with a numeral)
NUM1	(legal)
.1111	(illegal, begins with dot-numeral)
..1111	(unwise, could interfere with created symbols)

2.4.2 Defining Symbols

You can define a symbol by making it a label or by giving its value in a direct-assignment statement. Labels cannot be redefined, but direct-assignment symbols can be redefined anywhere in your program.

You can also define special-purpose symbols called OPDEFs and macros using the pseudo-op OPDEF and the pseudo-op DEFINE. (See Chapter 3.)

2.4.2.1 **Defining Labels** - A label is always a symbol with a suffixed colon. A label is in the first (leftmost) field of a MACRO statement and is one of the forms:

ERRFOUND:	(MACRO uses only ERRFOU)
CASE1:	(legal label)
OK:CONTIN:	(legal; you can use more than one label at a location)
CASE2::	(double colon declares label INTERNAL; see Section 2.4.5.2)
CASE3:!	(colon and exclamation point suppresses output by debugger)
CASE4::!	(double colon and exclamation point declares label INTERNAL and suppresses output by debugger)

When MACRO processes the label, the symbol and the current value of the location counter are entered in the user symbol table. A reference to the symbol addresses the code at the label.

You cannot redefine a label to have a value different from its original value. A label is relocatable if the address it represents is relocatable; otherwise it is absolute.

2.4.2.2 **Direct Assignments** - You define a direct-assignment symbol by associating it with an expression. (See Section 2.5 for a discussion of expressions.) A direct assignment is in one of the forms:

symbol=expression	(symbol and value of expression are entered in user symbol table)
symbol==expression	(symbol and value of expression are entered in user symbol table, output by debugger is suppressed)
symbol=:expression	(symbol and value of expression are entered in user symbol table, symbol is declared INTERNAL; see Section 2.4.5.2)
symbol==:expression	(symbol and value of expression are entered in user symbol table, symbol is declared INTERNAL, output by debugger is suppressed)

You can redefine a direct-assignment symbol at any time; the new direct assignment simply replaces the old definition.

NOTE

If you assign a multiword value using direct assignment, only the first word of the value is assigned to the symbol. For example, A=ASCIZ /ABCDEFGH/ is equivalent to A=ASCIZ /ABCDE/, since only the first five characters in the string correspond to code in the first word.

2.4.3 Variable Symbols

You can specify a symbol as a variable by suffixing it with a number sign (#). A variable symbol needs no explicit storage allocation. On finding your END statement, MACRO assembles variables into locations following the literal pool.

You can assemble variables anywhere in your program by using the VAR pseudo-op. This pseudo-op causes all variables found so far to be assembled immediately. (Variables found after the VAR statement are assembled at the end of the program or at the next VAR statement.)

2.4.4 Using Symbols

When you use a symbol in your program, MACRO looks it up in the symbol tables. Normally MACRO searches the macro table first, then the op-code table, and finally the user symbol table. However, if MACRO has already found an operator in the current statement and is expecting operands, then it searches the user symbol table first.

You can control the order of search for symbol tables by using the pseudo-op .DIRECTIVE MACPRF.

2.4.5 Symbol Attributes

The value of a symbol is either relocatable or absolute. The relocatability of a label is determined by the relocatability of the address assigned to it. You can define either an absolute or a relocatable value for a direct-assignment symbol.

In addition, each symbol in your program has one of the following attributes: local, INTERNAL global, or EXTERNAL global. This attribute is determined when the symbol is defined.

2.4.5.1 **Local Symbols** - A local symbol is defined for the use of the current program only. You can define the same symbol to have different values in separately assembled programs. A symbol is local unless you indicate otherwise.

2.4.5.2 **Global Symbols** - A global symbol is defined in one program, but is also available for use in other programs. Its table entry is visible to all programs in which the symbol is declared global.

A global symbol must be declared INTERNAL in the program where it is defined; it can be defined in only one program. In other programs sharing the global symbol, it must be declared EXTERNAL; it can be EXTERNAL in any number of programs.

To declare a symbol as INTERNAL global, you can:

1. Use the INTERN pseudo-op.

   ```
   INTERN FLAG1
   ```

2. Insert a colon after = in a direct-assignment statement.

   ```
   FLAG2=:200

   FLAG3==:200
   ```

3. Use an extra colon with a label.

   ```
   FLAG4::
   ```

4. For subroutine entry points, use the ENTRY pseudo-op. (This pseudo-op does more than declare the symbol INTERNAL. See Chapter 3.)

   ```
   ENTRY FLAG5
   ```

To declare a symbol as an EXTERNAL global, you can:

1. Use the EXTERN pseudo-op.

   ```
   EXTERN FLAG6
   ```

2. Suffix ## to the symbol at any of its uses. (Doing this once is sufficient, but you can use ## with all references to the symbol.)

   ```
   FLAG7##
   ```

2.5 EXPRESSIONS

You can combine numbers and defined symbols with arithmetic and logical operators to form expressions. You can nest expressions by using angle brackets. MACRO evaluates each expression (innermost nesting levels first), and either resolves it to a fullword value, or generates a Polish expression to pass to LINK. (See Sections 2.5.3 and 2.5.4.)

2.5.1 Arithmetic Expressions

An arithmetic expression can include any number or defined symbol, and any of the following operators:

+ addition

- subtraction

* multiplication

/ division

These examples assume that WORDS, X, Y, and Z have been defined elsewhere:

```
MOVEI 3,WORDS/5

ADDI 12,<X+Y-Z>

ADDI 12,<<WORDS/5>+1>*5
```

2.5.2 Logical Expressions

A logical expression can include any number or defined symbol whose value is absolute, and any of the following operators:

& AND

! OR (inclusive OR)

^! XOR (exclusive OR)

^- NOT

The unary operation ^-A generates the fullword one's complement of the value of A.

Each of the binary operations &, !, and ^! generates a fullword by performing the indicated operation over corresponding bits of the two operands. For example, A&B generates a fullword whose bit 0 is the result of A's bit 0 ANDed with B's bit 0, and so forth for all 36 bits.

2.5.3 **Polish (Complex) Expressions**

MACRO cannot evaluate certain expressions containing relocatable values or EXTERNAL symbols. Instead MACRO generates special expressions called Polish expressions, which tell LINK how to resolve the values at load time. MACRO also generates Polish expressions to resolve inter-PSECT references.

For example, assume that A and B are externally defined symbols. Then MACRO cannot perform the operations A+B-3, but instead generates a special Polish block containing an expression to pass to LINK; the expression is equivalent to -+AB3. (See REL Block Type 11 in the LINK Reference Manual.) At load time, the values of A and B are available to LINK, and the expression is resolved.

NOTE

> If you have used reverse Polish notation with a calculator, you should notice that although MACRO's Polish expressions are similar, they are not reversed. (These notations are called Polish because they were invented by the Polish logician Jan Lukasiewicz.)

2.5.4 **Evaluating Expressions**

2.5.4.1 **Hierarchy of Operations** - MACRO has a hierarchy of operations in evaluating expressions. In an expression without nests (angle brackets), or within a nested expression, MACRO performs its operations in this effective order:

1. All unary operations and shifts: +, -, ^-, ^D, ^O, ^B, B (binary shift), _ (underscore shift), ^F, ^L, E, K, M, G. Zeros are added for K, M, and G before any other operation is performed.

2. Logical binary operations (from left to right): ! (OR), ^! (XOR), & (AND).

3. Multiplication and division (from left to right): *, /.

4. Addition and subtraction (binary operations): +, -.

You can override this hierarchy by using angle brackets to show what you want done first. For example, suppose you want to calculate the sum of A and B, divided by C. You cannot do this with A+B/C because MACRO will perform the division B/C first, then add the result to A. With angle brackets you can write the expression <A+B>/C, telling MACRO to add A and B first, then divide the result by C.

Expressions can be nested to any level. The innermost nest is evaluated first; the outermost, last. Some examples of legal expressions (assuming that A1, B1, and C are defined symbols) are:

```
A1+B1/5
<A1+B1>/5
^-A1&B1^!C
^B101M-^D98+6
```

NOTE

> An expression given in halfword notation (that is, lefthalf,,righthalf) has each half evaluated separately in a 36-bit register. Then the 18 low-order bits of each half are joined to form a fullword. For example, the expression <4,,6>/2 generates the value 000002 000003.

2.5.4.2 **Evaluating Expressions with Relocatable Values** - The value of an expression is usually either absolute or relocatable. Recall that relocatable values in your binary code will have the relocation constant added at load time by LINK.

Assume that A and B are relocatable symbols, and that X and Y are absolute symbols, and that the relocation constant is k. Let a+k and b+k be the values of A and B after relocation. Then A+X makes sense (to LINK) because it means <a+k>+X, which is the same as <a+X>+k, clearly relocatable.

Since X and Y are both absolute, any operation combining them gives an absolute result.

Now look at the expression A+B. This means <a+k>+<b+k>, which is the same as <a+b>+2k, neither absolute nor relocatable. Similarly, A*B means <a+k>*<b+k>, or <a*b>+<a+b>*k+k*k, again neither absolute nor relocatable. Such expressions cannot be evaluated by MACRO and are passed as Polish expressions to LINK.

More generally, you can see if an expression is absolute or relocatable by substituting relocated forms as above (for example, a+k), and separating it (if possible) into the form

    absolute+n*k

where absolute is an absolute expression. If n=0, the expression is absolute; if n=1, it is relocatable. If n is neither 0 nor 1, or if the expression cannot be put into the form above, then the expression is neither absolute nor relocatable. (Nevertheless, LINK will correctly evaluate the expression at load time.)

2.6 MACRO-DEFINED MNEMONICS

MACRO-defined mnemonics are words that MACRO recognizes and can translate to binary code. These mnemonics include:

1. Machine instruction mnemonics
2. I/O instruction mnemonics
3. I/O device code mnemonics
4. KL10 EXTEND instruction mnemonics
5. JRST and JFCL mnemonics
6. DECsystem-10 monitor call mnemonics
7. DECsystem-10 CALLI mnemonics
8. DECsystem-10 TTCALL mnemonics
9. DECsystem-10 MTAPE mnemonics
10. F40-switch-dependent mnemonics

Each type of mnemonic is discussed and tabulated in Appendix C. These mnemonics, together with MACRO's pseudo-ops and special characters, form the MACRO language.

CHAPTER 3

PSEUDO-OPS

A pseudo-op is a statement that directs the assembler to generate code or set switches to control assembly and listing of your program. For example, the pseudo-op RADIX does not generate code, but it tells MACRO how to interpret numbers in your program. The pseudo-op EXP generates one word of code for each argument given with it.

To use a pseudo-op in your program, follow it with a space or tab, and any required or optional arguments or parameters. The program examples in Appendix D show pseudo-ops used in context.

This chapter describes the use and functions of each pseudo-op (alphabetically). The headings included for each description, if applicable, are:

1. FORMAT
2. FUNCTION
3. EXAMPLES
4. OPTIONAL NOTATIONS
5. RELATED PSEUDO-OPS
6. COMMON ERRORS

Some entries under COMMON ERRORS cite single-character error codes (for example, M error). These codes are discussed in Section 8.2.

Many of the examples show some parts of the code assembled. The format and meaning of assembled code is discussed in Section 6.1.

ARRAY

FORMAT

ARRAY sym[expression]

expression = an integer value in the current radix, indicating the number of words to be allocated; the expression cannot be EXTERNAL, relocatable, or a floating-point decimal number, and its value must not be negative.

FUNCTION

Reserves a block of storage whose length is the value of the expression, and whose location is identified by the symbol. Storage is allocated along with other variable symbols in the program.

If the pseudo-op TWOSEG is used, ARRAY storage must be in the low segment. (See the VAR pseudo-op.)

The allocated storage is not necessarily zeroed.

If you use ARRAY in a PSECT, storage is allocated within that PSECT.

NOTE

Though the expression portion of an OPDEF must be in square brackets, this use of the brackets is completely unrelated to literals or literal handling.

EXAMPLES

```
ARRAY START[200]
ARRAY PLACE[1000]
ARRAY ERRS[2000]
```

OPTIONAL NOTATIONS

ARRAY sym1,sym2 [expression]

Both sym1 and sym2 have a length equal to the value of the expression.

RELATED PSEUDO-OPS

BLOCK, .COMMON, INTEGER, VAR

COMMON ERRORS

Using an EXTERNAL symbol for name or size of the array (E error).

ASCII

FORMAT

ASCII dtextd

d = delimiter; first nonblank character, whose second appearance terminates the text.

text = string of text characters to be entered.

FUNCTION

Enters ASCII text in the binary code. Each character uses seven bits. Characters are left justified in storage, five per word, with bit 35 in each word set to 0, and any unused bits in the last word set to 0.

EXAMPLES

```
105 122 122 117 122     ASCII /ERROR MESSAGE/
040 115 105 123 123
101 107 105 000 000

123 124 101 122 124     ASCII !STARTING AGAIN!
111 116 107 040 101
107 101 111 116 000

105 116 104 123 040     ASCII ?ENDS WITH ZEROS?
127 111 124 110 040
132 105 122 117 123
```

OPTIONAL NOTATIONS

Omit the space or tab after ASCII. This is not allowed if the delimiter is a letter, number, dot, dollar sign, or percent sign (that is, a possible symbol constituent), or if the ASCII value of the delimiter character is less than 040 or greater than 172.

Right justified ASCII can be entered by using double quotes to surround up to five characters; for example,

```
201 01 0 00 000101      MOVEI AC1,"A"
```

RELATED PSEUDO-OPS

ASCIZ, .DIRECTIVE FLBLST, RADIX50, SIXBIT

COMMON ERRORS

Using the delimiter character in the text string.

Missing the end delimiter (that is, attempting to use a carriage return as a delimiter).

Using more than 5 characters in a right-justified ASCII string, or more than 2 characters if in the address field (Q error).

Giving direct assignment of a long ASCII string value to a symbol (for example A=ASCII /ABCDEFGH/). Only the first word (five characters, left justified) is assigned.

Using ASCII when ASCIZ is required.

ASCIZ

FORMAT

ASCIZ dtextd

d = delimiter; first nonblank character, whose second appearance terminates the text.

text = string of text characters to be entered.

FUNCTION

Enters ASCII text exactly as in the pseudo-op ASCII, except that a trailing null character is guaranteed. That is, if the number of characters in text is a multiple of five, a fullword of zeros is generated.

EXAMPLES

```
105 122 122 117 122     ASCIZ /ERROR MESSAGE/
040 115 105 123 123
101 107 105 000 000

123 124 101 122 124     ASCIZ !STARTING AGAIN!
111 116 107 040 101
107 101 111 116 000

105 116 104 123 040     ASCIZ ?ENDS WITH ZEROS?
127 111 124 110 040
132 105 122 117 123
000 000 000 000 000
```

OPTIONAL NOTATIONS

Omit the space or tab after ASCIZ. This is not allowed if the delimiter is a letter, number, dot, dollar sign, or percent sign (that is, a possible symbol constituent), or if the ASCII value of the delimiter character is less than 040 or greater than 172.

RELATED PSEUDO-OPS

ASCII, .DIRECTIVE FLBLST, RADIX50, SIXBIT

COMMON ERRORS

Using the delimiter character in the text string.

Missing the end delimiter (that is, attempting to use a carriage return as a delimiter).

Giving direct assignment of a long ASCII string value to a symbol (for example A=ASCII /ABCDEFGH/). Only the first word (five characters, left justified) is assigned.

In a macro, using a delimiter character that interferes with recognition of a dummy-argument. For example, in the macro

```
DEFINE FOO(X)<
ASCIZ .X.
>
```

X is not seen as a dummy-argument because .X. is itself a valid symbol.

(Continued on next page)

ASCIZ (Cont.)

In the macro

```
DEFINE FOO(X)<
ASCIZ /X/
>
```

X is seen as a dummy-argument because the slash (/) is not valid in a symbol.

The macro

```
DEFINE FOO(X)<
ASCIZ .'X'.
```

uses the concatenation operator (') to assure recognition of X as a dummy-argument. (See Section 5.4 for a discussion on concatenating arguments.)

.ASSIGN

FORMAT

.ASSIGN sym1,sym2,increment

sym1 and sym2 = global symbols.

increment = expression with integer value.

FUNCTION

MACRO generates a REL Block Type 100. (See the LINK Reference Manual.) At the time the program is loaded into memory, assigns the value of sym2 to sym1, and adds increment to sym2.

The .ASSIGN pseudo-op is useful for assigning a block of storage in one module and providing another module with the symbols needed to reference that block.

EXAMPLES

```
.ASSIGN A,PC,5               ;Assigns the value of PC to A,
                             ; then redefines the value of
                             ; PC to be PC+5.

.ASSIGN ERR1,ERRS,ERNO       ;Assigns the value of ERRS to
                             ; ERR1, then redefines ERRS to
                             ; be ERRS plus the current
                             ; value of ERNO.
```

OPTIONAL NOTATIONS

.ASSIGN sym1,sym2

If the increment is missing, its value is 1.

COMMON ERRORS

Sym1 or sym2 not global.

Increment not defined at assembly time.

ASUPPRESS

FORMAT: ASUPPRESS

FUNCTION: Causes all local or INTERNAL symbols that are not referenced after the ASUPPRESS to be deleted from MACRO's symbol table at the end of Pass 2. These symbols will not be output to LINK, will not be available to the debugger, and will not appear in the symbol table in the program listing file.

If you use ASUPPRESS at the end of Pass 1, only those symbols defined or referenced in Pass 2 remain in MACRO's symbol table. This is useful for parameter files that define many more symbols than are actually used, since the unused symbols can be automatically deleted if they are defined in IF1 conditionals.

RELATED PSEUDO-OPS: PURGE, SUPPRESS

BLOCK

FORMAT

BLOCK expression

expression = an integer value in the current radix, indicating the number of words to be allocated; the expression cannot be EXTERNAL, relocatable, or a floating-point decimal number, and its value must not be negative.

FUNCTION

Reserves a block of locations whose length is the value of the expression. The location counter is incremented by this value. The allocated locations are not necessarily zeroed.

Note that the BLOCK pseudo-op does not generate or store code. Therefore it should not be used in a literal, since this will result in overwriting the reserved space during literal pooling.

If you use the BLOCK pseudo-op to reserve words meant for data storage, these words should be reserved in the low segment of a two-segment program.

EXAMPLES

```
002101' 200 02 0 00 400033'               MOVE  2,[XWD FRM,TO]
002102' 251 02 0 00 003010'               BLT 2,TOEND
......                                     . . .
002611'                             FRM:  BLOCK 100
002711'                             TO:   BLOCK 100
                    003010'               TOEND=.-1
```

OPTIONAL NOTATIONS

Use the pseudo-op Z inside literals.

RELATED PSEUDO-OPS

ARRAY, .COMMON, INTEGER, VAR

COMMON ERRORS

Relocatable expression (R error).

Floating-point or negative expression (A error).

Value of expression larger than 777777.

Expression contains EXTERNAL symbol (E error).

Expression contains nonexistent symbol (V error).

BLOCK used in literal (L error).

BYTE

FORMAT

BYTE bytedef ... bytedef

bytedef=(n)expression,...,expression

n = byte size in bits; n is a decimal expression in the range 1 to 36.

expression = value to be stored.

FUNCTION

Stores values of expressions in n-bit bytes, starting at bit 0 of the storage word. The first value is stored in bits 0 to n-1; the second in bits n to 2n-1; and so forth for each given value.

If a byte will not fit in the remaining bits of a word, the bits are zeroed and the byte begins in bit 0 of the next word. If a value is too large for the byte, it is truncated on the left.

If the byte size is 0 or is missing (empty parentheses), a zero word is generated.

EXAMPLES

```
          000002       VELOCY=2
05 00 00 01 05 02     BYTE (6)5,0,,101,5,VELOCY
```

generates the storage value 050000 010502. The two commas indicate a null argument; the 101 (octal) is too large for the byte size and is left truncated.

```
07 00 01 007 000   BYTE (6)7,0,1(9)7,0,1,"A"
001 101 000000
```

Notice that the code for "A" (101) is right justified in its 9-bit byte.

COMMON ERRORS

Byte size too big (A error).

Missing left or right parenthesis (A error).

Extraneous comma before left parenthesis; the comma generates a null byte.

Using an EXTERNAL symbol or EXTERNAL complex expression for n or expression.

COMMENT

FORMAT	COMMENT dtextd d = delimiter; the first nonblank character, whose second appearance terminates the text. text = text to be entered as a comment.
FUNCTION	Treats the text between the delimiters as a comment. The text can include a CR-LF to facilitate multiline comments, as shown below.
EXAMPLES	COMMENT /THIS IS A COMMENT THAT IS MORE THAN 1 LINE LONG/
OPTIONAL NOTATIONS	Omit the space or tab after COMMENT. This is not allowed if the delimiter is a letter, number, dot, dollar sign, or percent sign (that is, a possible symbol constituent), or if the ASCII value of the delimiter character is less than 040 or greater than 172. Use a semicolon (;) to make the rest of the line into a comment.
RELATED PSEUDO-OPS	REMARK
COMMON ERRORS	Using the delimiter character in the text string. Missing the end delimiter (that is, attempting to use a carriage return as a delimiter).

.COMMON

FORMAT

.COMMON symbol[expression]

symbol = name of a FORTRAN COMMON block.

expression = an expression having a positive integer value; this value defines the length of the COMMON block.

FUNCTION

Defines a FORTRAN or FORTRAN-compatible COMMON block. Causes the equivalent action of a FORTRAN labeled COMMON. (See the FORTRAN Programmer's Reference Manual.)

You can use .COMMON to define blank COMMON; to do this, use the symbol .COMM. as the name of the COMMON block. (Both FORTRAN and LINK recognize this as the name of blank COMMON.)

To define a COMMON block, MACRO generates a REL Block Type 20. (See the LINK Reference Manual.)

If used, the .COMMON pseudo-op must precede any MACRO statement that generates binary code, and must precede any other reference to the symbol name.

EXAMPLES

```
.COMMON DATA1[50]
```

OPTIONAL NOTATIONS

.COMMON symbol,...,symbol[expression]

defines a COMMON array for each symbol given. Each array has a length equal to the value of the expression.

RELATED PSEUDO-OPS

ARRAY, BLOCK, EXTERN, INTEGER

COMMON ERRORS

Missing left or right square bracket (A error).

Using a relocatable value or EXTERNAL symbol in expression.

.CREF

FORMAT	.CREF
FUNCTION	Resumes output of cross-referencing that was suspended by the .XCREF pseudo-op.
OPTIONAL NOTATIONS	Can apply to specific symbols to cancel a previous .XCREF on those symbols, as in .CREF symbol,...,symbol
RELATED PSEUDO-OPS	.XCREF
COMMON ERRORS	Specifying a nonexistent symbol (A error).

DEC

FORMAT — DEC expression,...,expression

FUNCTION — Defines the local radix for the line as decimal; the value of each expression is entered in a fullword of code. The location counter is incremented by 1 for each expression.

EXAMPLES

```
                RADIX 8
000000  000012  DEC 10,4.5,3.1416,6.03E-26,3
203440  000000
202622  077714
055452  456522
000000  000003
```

OPTIONAL NOTATIONS — Use the EXP pseudo-op and prefix ^D to each expression that must be evaluated in radix 10. In the example above, only the first expression, "10," has different evaluations in radix 8 and radix 10. Therefore an equivalent notation is

```
000000  000012  EXP ^D10,4.5,3.1416,6.03E-26,3
203440  000000
202622  077714
055452  456522
000000  000003
```

RELATED PSEUDO-OPS — EXP, RADIX, OCT

DEFINE

FORMAT

DEFINE macroname(darglist)<macrobody>

macroname = a symbolic name for the macro defined. This name must be unique among all macro, OPDEF, and SYN symbols.

darglist = a list of dummy-arguments.

macrobody = source code to be assembled when the macro is called.

FUNCTION

Defines a macro. (See Chapter 5.)

EXAMPLES

See Chapter 5.

RELATED PSEUDO-OPS

.DIRECTIVE (with .ITABM, .XTABM, or MACMPD arguments), IRP, IRPC, OPDEF, STOPI, SYN

COMMON ERRORS

Mismatched parentheses.

Mismatched angle brackets.

Using identical names for a macro and an OPDEF or SYN symbol (X error).

DEPHASE

FORMAT DEPHASE

FUNCTION Suspends the effect of a PHASE pseudo-op. Restores the location counter to its mode previous to the segment of PHASEd code.

For further details, see the pseudo-op PHASE.

EXAMPLES

```
400000'                                    RELOC 400000
000000                                     PHASE 0
000000  201 01 0 00 000000        TAG:     MOVEI 1,0
400001'                                    DEPHASE
400001' 254 00 0 00 000000'                JRST TAG
```

RELATED PSEUDO-OPS PHASE

.DIRECTIVE

FORMAT

.DIRECTIVE directive,...,directive

FUNCTION

Sets switches to enable or disable MACRO features. If a directive has a logical opposite, you can use NO as a prefix to reverse the directive. The directives are:

.ITABM - include spaces and tabs as part of passed arguments in macro call.

.XTABM - strip leading and trailing spaces and tabs from passed arguments in macro call. .XTABM is the default setting.

MACMPD - match paired delimiters in macro call. MACMPD is the default for assembly. It implies .XTABM and disables .ITABM. Using .DIRECTIVE NO MACMPD disables all quoting characters except angle brackets in macro arguments, and offers you a choice of .ITABM or .XTABM.

LITLST - list all binary code for literals in-line.

FLBLST - list only first line of binary code for multiline text. NO FLBLST is the default.

.OKOVL - allow overflow for arithmetic and for the pseudo-ops DEC, EXP, and OCT.

.EROVL - give an N error for arithmetic overflow. .EROVL is the default.

MACPRF - prefer macro definition of symbol over other definitions of the same symbol. This does not affect the searching of .UNV files.

SFCOND - suppress source listing for failing conditional assembly. The lines containing the opening and closing angle brackets are not suppressed.

.NOBIN - do not generate binary (.REL) file.

KA10 - enter KA10 as CPU type in header block of binary file.

KI10 - enter KI10 as CPU type in header block of binary file.

KL10 - enter KL10 as CPU type in header block of binary file.

EXAMPLES

```
.DIRECTIVE MACMPD,.NOBIN
```

COMMON ERRORS

Using NO with a directive that does not have a logical opposite.

END

FORMAT

END expression

expression = an optional operand that specifies the address of the first instruction to be executed; can be EXTERNAL.

FUNCTION

Must be the last statement in a MACRO program. Statements after END are ignored. The starting address is optional and normally is given only in the main program. (Since subprograms are called from the main program, they need not specify a starting address.)

When the assembler first encounters an END statement, it terminates Pass 1 and begins Pass 2. The END terminates Pass 2 on the second encounter, after which the assembler simulates XLISTed LIT and VAR statements beginning at the current location. (In a PSECTed program, the LIT and VAR statements are simulated for each PSECT.)

EXAMPLES

```
END START
```

START is a label at the starting address.

OPTIONAL NOTATIONS

Use the END statement to specify a transfer word in some output file formats. (See pseudo-ops RIM, RIM10, and RIM10B in Appendix E.)

RELATED PSEUDO-OPS

PRGEND

COMMON ERRORS

Failing to end a text string or literal with a closing delimiter; MACRO cannot see the END statement.

Including an END statement in a source file when it is not the last file in a group of files you want assembled as a single program.

Closing the input file immediately after the characters "END" with no following carriage return.

.ENDPS

FORMAT	.ENDPS
FUNCTION	Suspends use of the relocation counter associated with the current PSECT. If the current PSECT is nested in other PSECTs, the relocation counter for the next outer PSECT is activated. Otherwise, the relocation counter for the blank PSECT is activated. MACRO generates a REL Block Type 22. (See the LINK Reference Manual.) For a complete discussion of PSECTs and their handling, see Section 9.1.3.
OPTIONAL NOTATIONS	Give the name of the current PSECT with the .ENDPS pseudo-op. For example, .ENDPS A causes MACRO to verify that A is the name of the current PSECT; if not, an error message is issued.
RELATED PSEUDO-OPS	LOC, .ORG, .PSECT, RELOC, TWOSEG

ENTRY

FORMAT

ENTRY symbol,...,symbol

symbol = name of an entry point in a library subroutine.

FUNCTION

Defines each symbol in the list following the ENTRY pseudo-op as an INTERNAL symbol and places them in a REL Block Type 4 at the beginning of the .REL output file. If this .REL file is later included in an indexed library of subroutines, then the symbol will also be included in a REL Block Type 14 at the beginning of the library. (Except for this, ENTRY is equivalent to INTERN.)

If LINK is in library search mode, a subroutine will be loaded if the program to be executed contains an undefined global symbol that matches a name in the library entry list for that program.

Since library subroutines are external to programs using them, the calling program must list them in EXTERN statements.

EXAMPLES

If the MATRIX subroutine is a library subroutine, it must contain the statement

```
ENTRY MATRIX
```

in order to make the symbol MATRIX available to other programs. In addition, it must define the symbol MATRIX as a label at the address where execution of the call is to begin:

```
MATRIX:
```

RELATED PSEUDO-OPS

INTERN, EXTERN

COMMON ERRORS

Not defining the symbol in the program.

Purging an ENTRY symbol in Pass 2 only. The ENTRY symbol is normally output at the beginning of Pass 2; a PURGE of an ENTRY symbol must occur in Pass 1 to be effective.

EXP

FORMAT EXP expression,...,expression

FUNCTION Enters the value of each expression (in the current radix) in a fullword of code.

EXAMPLES

```
        000003   X=3
        000101   HALF=101
        000004   B=4
        000002   A=2
000000  000003   EXP X,4,^D65,HALF,B+362-A
000000  000004
000000  000101
000000  000101
000000  000364
```

RELATED PSEUDO-OPS DEC, OCT

EXTERN

FORMAT

EXTERN symbol,...,symbol

FUNCTION

Identifies symbols as being defined in other programs. EXTERNAL symbols cannot be defined within the current program.

At load time, the value of an EXTERNAL symbol is resolved by LINK if you load a module that defines the symbol as an INTERNAL symbol. (If you do not load such a module, LINK gives an error message for the undefined EXTERNAL symbol.)

An EXTERNAL symbol cannot be used for any program values affecting address assignment (such as arguments to LOC or RELOC).

For a discussion of global symbols and their resolution by LINK, see Section 2.4.5.2.

EXAMPLES

```
EXTERN SQRT,CUBE,TYPE
```

OPTIONAL NOTATIONS

Suffix ## to the symbol. This declares the symbol EXTERNAL, and eliminates the need for the EXTERN pseudo-op. Most programmers who use the ## notation do so at all occurrences of the symbol to show at each site that the symbol is EXTERNAL.

For example, the two statements

```
EXTERN A
ATWO=A*2
```

can be simplified to

```
ATWO=A##*2
```

RELATED PSEUDO-OPS

INTERN, ENTRY, UNIVERSAL

COMMON ERRORS

Attempting to declare a symbol as EXTERNAL after its first use has made it local (by default) or INTERNAL (by declaration).

Declaring a symbol as EXTERNAL in a program that searches a UNIVERSAL file that gives a conflicting definition.

.HWFRMT

FORMAT	.HWFRMT
FUNCTION	Causes binary code to be listed in halfword format.
EXAMPLES	see below
OPTIONAL NOTATIONS	Use the /G switch described in Table 7-1.
RELATED PSEUDO-OPS	.MFRMT

```
200 01 0 02 000002        MOVE 1,2(2)
                          .HWFRMT
200042  000002            MOVE 1,2(2)
```

.IF

FORMAT: .IF expression,qualifier,<code>

FUNCTION: Gives criterion and code for conditional assembly. The code is assembled if:

qualifier is AND	expression is
ABSOLUTE	absolute
ASSIGNMENT	a direct-assignment symbol
ENTRY	a symbol given in ENTRY pseudo-op
EXTERNAL	an EXTERNAL symbol
INTERNAL	an INTERNAL or ENTRY symbol
GLOBAL	a global symbol
LABEL	a label
LOCAL	a local symbol
LRELOCATABLE	a lefthalf relocatable symbol
MACRO	a macro name
NEEDED	an undefined but referenced symbol
NUMERIC	numeric
OPCODE	an opcode
OPDEF	a symbol defined by OPDEF pseudo-op
REFERENCED	a symbol already in the symbol table
RELOCATABLE	a relocatable symbol
RRELOCATABLE	a righthalf relocatable symbol
SYMBOL	a symbol (instead of a number)
SYNONYM	a symbol defined by SYN pseudo-op

NOTE

If the expression has different properties in Pass 1 and Pass 2, the number of words of code generated may be different for each pass.

EXAMPLES:

```
.IF FOO,MACRO,<FOO>
```

OPTIONAL NOTATIONS: Abbreviate qualifier up to unique initial letters. For example, you can abbreviate OPCODE to OPC, but not to OP, since OPDEF has the same first two letters.

Omit the comma preceding the left angle bracket.

RELATED PSEUDO-OPS: .DIRECTIVE SFCOND, .IFN, IFx group

COMMON ERRORS: Omitting the comma between expression and qualifier.

Mismatching angle brackets.

Misplacing the .IF statement in such a way that the property given by the qualifier is different in Pass 1 and Pass 2. For example, the following code generates phase errors in Pass 2:

```
        .IF FOO,OPDEF,<JFCL>
        OPDEF FOO[JRST]
NXTLAB: END
```

.IFN

FORMAT
.IFN expression,qualifier,<code>

FUNCTION
Gives criterion and code for conditional assembly. The code is assembled if:

qualifier is AND	expression IS NOT
ABSOLUTE	absolute
ASSIGNMENT	a direct-assignment symbol
ENTRY	a symbol given in ENTRY pseudo-op
EXTERNAL	an EXTERNAL symbol
INTERNAL	an INTERNAL or ENTRY symbol
GLOBAL	a global symbol
LABEL	a label
LOCAL	a local symbol
LRELOCATABLE	a lefthalf relocatable symbol
MACRO	a macro name
NEEDED	an undefined but referenced symbol
NUMERIC	numeric
OPCODE	an opcode
OPDEF	a symbol defined by OPDEF pseudo-op
REFERENCED	a symbol already in the symbol table
RELOCATABLE	a relocatable symbol
RRELOCATABLE	a righthalf relocatable symbol
SYMBOL	a symbol (instead of a number)
SYNONYM	a symbol defined by SYN pseudo-op

NOTE

If the expression has different properties in Pass 1 and Pass 2, the number of words of code generated may be different for each pass.

EXAMPLES

```
.IFN FOO,OPDEF,<OPDEF FOO[270B8]>
```

OPTIONAL NOTATIONS
Abbreviate qualifier up to unique initial letters. For example, OPCODE can be abbreviated to OPC, but not to OP, since OPDEF has the same first two letters.

Omit the comma preceding the left angle bracket.

RELATED PSEUDO-OPS
.DIRECTIVE SFCOND, .IF, IFx group

COMMON ERRORS
Omitting the comma between expression and qualifier.

Mismatching angle brackets.

Misplacing the .IFN statement in such a way that the property given by the qualifier is different in Pass 1 and Pass 2. For example, the following code generates phase errors in Pass 2:

```
        .IFN FOO,OPDEF,<JFCL>
        OPDEF FOO[JRST]
NXTLAB: END
```

IFx group

FUNCTION

Gives criterion and code for conditional assembly. A symbol or expression used to define the conditions for assembly must be defined before MACRO reaches the conditional statement. If the value of such a symbol or expression is not the same on both assembly passes, a different number of words of code may be generated, and a phase error can occur.

The forms of the IF pseudo-op are listed below; in the first six forms, n is the value of the given expression.

IFE expression,<code> - assemble code if $n=0$.

IFN expression,<code> - assemble code if $n \neq 0$.

IFG expression,<code> - assemble code if $n>0$.

IFGE expression,<code> - assemble code if $n \geq 0$.

IFL expression,<code> - assemble code if $n<0$.

IFLE expression,<code> - assemble code if $n \leq 0$.

IF1 <code> - assemble code on Pass 1.

IF2 <code> - assemble code on Pass 2.

IFDEF symbol,<code> - assemble code if the symbol is defined as user-defined, an opcode, or a pseudo-op.

IFNDEF symbol,<code> - assemble code if the symbol is not defined as user-defined, an opcode, or a pseudo-op. Code is also assembled if the symbol has been referenced, but is not yet defined. This can occur during Pass 1.

IFIDN <string1><string2>,<code> - assemble code if the strings are identical.

IFDIF <string1><string2>,<code> - assemble code if the strings are different.

NOTES

1. For IFIDN and IFDIF, the assembler compares the two strings (interpreted as ASCII) character by character.

2. The IFIDN and IFDIF pseudo-ops usually appear in macro definitions, where one or both strings are dummy-arguments.

(Continued on next page)

IFx group (Cont.)

IFB <string>,<code> - assemble code if the string contains only blanks and tabs.

IFNB <string>,<code> - assemble code if the string does not contain only blanks and tabs.

EXAMPLES

```
%%CC==%%CC+1          ;Increment character count
IFG %%CC-5,<%%CC==0   ;Word overflowed?
%%WC==%%WC+1>         ;Yes, to next word
```

OPTIONAL NOTATIONS

Omit angle brackets enclosing code for single-line conditionals.

Omit the comma preceding the code if the code is enclosed in angle brackets.

For IFIDN, IFDIF, IFB, and IFNB only: use a nonblank, nontab character other than < as the initial and terminal delimiters for a string (as in pseudo-ops ASCII and ASCIZ). You can then include angle brackets in the string.

RELATED PSEUDO-OPS

.DIRECTIVE SFCOND, .IF, .IFN

COMMON ERRORS

Comparison string too large (A error).

Mismatched angle brackets.

EXTERNAL symbol used for comparison (E error).

String not properly delimited.

Missing comma with single-line conditional.

INTEGER

FORMAT

INTEGER symbol,...,symbol

symbol = the name of a location to be reserved.

FUNCTION

Reserves storage locations at the end of the program on a one-per-given-symbol basis. The symbols are equivalent to variable symbols.

For a two-segment program, INTEGER storage must be in the low segment.

EXAMPLES

```
INTEGER A,B,C
```

OPTIONAL NOTATIONS

Reserve a single storage location by suffixing a number sign (#) to a symbol in the operand field. For example,

```
ADD 3,TEMP#
```

is equivalent to

```
INTEGER TEMP
ADD 3,TEMP
```

RELATED PSEUDO-OPS

ARRAY, BLOCK, .COMMON, VAR

INTERN

FORMAT	INTERN symbol,...,symbol
FUNCTION	Declares each given symbol to be INTERNAL global; therefore its definition, which must be in the current program, is available to other programs at load time. Each such symbol must be defined as a label, a variable, or a direct-assignment symbol. MACRO builds a list of symbol definitions that will be available to other programs at load time. OPDEF symbols can be declared INTERNAL, and thus be made available to other programs at load time. However, if the current program has another symbol (besides the OPDEF symbol) of the same name, the INTERNAL declaration will apply to that symbol rather than to the OPDEF symbol.
EXAMPLES	INTERN SQUARE,CBROOT,TYPE2
OPTIONAL NOTATIONS	TAG:: ;INTERNAL label VALUE=:expression ;INTERNAL direct assignment
RELATED PSEUDO-OPS	EXTERN, ENTRY
COMMON ERRORS	Failing to define an INTERNAL symbol in the current program. Using INTERN for a library entry point (when ENTRY is required).

IOWD

FORMAT

IOWD expl,exp2

expl, exp2 = expressions.

FUNCTION

Generates one I/O transfer word in a special format for use in BLKI and BLKO and all five pushdown instructions (ADJSP, PUSH, POP, PUSHJ, POPJ). The left half of the assembled word contains the 2's complement of the value of expl, and the right half contains the value exp2-1.

EXAMPLES

The following line shows how IOWD 6,^D256 places -6 (octal 777772) in the left halfword and 256 (octal 377) in the right halfword:

```
777772  000377              IOWD 6,^D256
```

The following lines show IOWD STL,STK used in a literal. The LIT pseudo-op then shows the code generated in the literal pool.

```
        000017              P==17
        000001              AC1==1
        000100              STL==100
                       STK: BLOCK STL
200 17 0 00 001053'         MOVE P,[IOWD STL,STK]
261 17 0 00 000001          PUSH P,AC1
254 00 0 00 001054'         JRST END
      . . .                 . . .
                            LIT
777700  000001
104 00 0 00 000170     END: HALTF
```

OPTIONAL NOTATIONS

XWD -expl,exp2-1

-expl,,exp2-1

COMMON ERRORS

Using a relocatable expression for expl (R error).

IRP

FORMAT

IRP darg,<code>

darg = one of the dummy-arguments of the enclosing macro definition. (You can use IRP only in the body of a macro definition.)

FUNCTION

Generates one expansion of code for each subargument of the string that replaces darg. Each occurrence of darg within the expansion is replaced by the subargument currently controlling the expansion. (See Section 5.6.)

Concatenation and line continuation are not allowed across end-of-IRP, since a carriage return and linefeed are appended to each expansion. See the example below.

EXAMPLES

```
                                LALL
              000000            Z=0
              000001            ANSWER=1
              000002            Q=2
              000003            X=3
              000004            Y=4
                                DEFINE SUM(A,B)<
                                        MOVEI Q,0
                                        IRP A,<ADD Q,A>
                                        MOVEM Q,B
                                >
                                SUM (<X,Y,Z>,ANSWER)^
201 02 0 00 000000                      MOVEI Q,0
                                        IRP
140 02 0 00 000003              ADD Q,X
140 02 0 00 000004              ADD Q,Y
140 02 0 00 000000              ADD Q,Z

202 02 0 00 000001                      MOVEM Q,ANSWER
```

RELATED PSEUDO-OPS

IRPC, STOPI

COMMON ERRORS

IRP NOT IN A MACRO (A ERROR).

Argument is not a dummy symbol (A error).

Argument is a created symbol (A error).

Mismatched angle brackets.

IRPC

FORMAT

IRPC darg,<code>

darg = one of the dummy-arguments of the enclosing macro definition. (IRPC can only be used in the body of a macro definition.)

FUNCTION

Generates one expansion of code for each character of the string that replaces darg. Each occurrence of darg within the expansion is replaced by the character currently controlling the expansion. (See Section 5.6.)

Concatenation and line continuation are not allowed across end-of-IRPC, since a carriage return and linefeed are appended to each expansion. See the example below.

EXAMPLES

```
                              DEFINE A(B)<IRPC B,<ASCIZ \B\>>
                              A(STRING)^IRPC
123 000 000 000 000           ASCIZ \S\
124 000 000 000 000           ASCIZ \T\
122 000 000 000 000           ASCIZ \R\
111 000 000 000 000           ASCIZ \I\
116 000 000 000 000           ASCIZ \N\
107 000 000 000 000           ASCIZ \G\
```

RELATED PSEUOD-OPS

IRP, STOPI

COMMON ERRORS

IRPC NOT IN A MACRO (A ERROR).

Argument is not a dummy symbol (A error).

Argument is a created symbol (A error).

Mismatched angle brackets.

LALL

FORMAT	LALL
FUNCTION	Causes the assembler to print in the program listing file everything that is processed, including all text and macro expansions. Since XALL is the default, you must use LALL if you want full macro expansions listed. This can be helpful in debugging a program. LALL does not produce comments in a macro expansion if the comments are preceded by double semicolons (;;). This is because such comments are not stored.
OPTIONAL NOTATIONS	Use the /E switch described in Table 7-1.
RELATED PSEUDO-OPS	LIST, SALL, XALL, XLIST

.LINK

FORMAT	.LINK chain-number,store-address,chain-address chain-number = a positive integer expression that associates the link with others having the same number. store-address = a symbol giving the store address for this entry in the chain. chain-address = an optional integer expression giving the address of this entry in the chain. If you omit the chain-address, MACRO generates a 0 and LINK uses the store-address as the chain-address.
FUNCTION	Generates static chains at load time. MACRO generates a REL Block Type 12. (See the LINK Reference Manual for a full discussion of LINK's handling of these chains.)
EXAMPLES	See the LINK Reference Manual (REL Block Type 12) for extensive examples of using .LINK and .LNKEND.
RELATED PSEUDO-OPS	.LNKEND
COMMON ERRORS	Chain-number not absolute (A error). EXTERNAL expression for store-address or chain-address (E error).

LIST

FORMAT	LIST
FUNCTION	Resumes listing following an XLIST statement. The LIST function is implicitly contained in the END statement.
OPTIONAL NOTATIONS	Use the /L switch described in Table 7-1.
RELATED PSEUDO-OPS	LALL, SALL, XALL, XLIST

LIT

FORMAT LIT

FUNCTION Assembles literals beginning at the current address. The literals assembled are those found since the previous LIT, or since the beginning of the program, whichever is later. The location counter is incremented by 1 for each word assembled.

In a PSECTed program, LIT assembles only literals in the current PSECT.

A literal found after the LIT is not affected. It will be assembled at the next following LIT, or at the END statement, whichever is earlier.

At the END statement, unassembled literals are placed in open-ended storage after the end-of-program. If data is also to be entered in open-ended storage, literals stored there may be overwritten. (See Appendix F for a discussion of storage allocation.) This possibility is avoided by using LIT before the END statement.

Assembling literals with LIT also produces a listing of their binary code. Literals unassembled at the END are XLISTed.

Literals having the same value are collapsed in MACRO's literal pool. Thus for the statements:

```
PUSH P,[0]
PUSH P,[0]
MOVEI AC1,[ASCIZ /TEST1/]
```

the same address is shared by the two literals [0], and by the null word generated at the end of [ASCIZ /TEST1/]. Literal collapsing is suppressed for those literals that contain errors, undefined expressions, or EXTERNAL symbols.

NOTES

1. If the code immediately preceding a LIT does not cause a transfer of execution control to some other location, execution will "fall into" the literal pool, producing unpredictable results.

2. In a file containing PRGEND pseudo-ops, only one LIT is permitted in each module before the last one. The last module (containing the END statement), or any file without PRGENDs, can contain multiple LITs.

(Continued on next page)

LIT (Cont.)

EXAMPLES

```
400046' 200 00 0 00 400050'     MOVE 0,[XWD 1,3]
400047' 047 00 0 00 000041      GETTAB 0,
400050'                         LIT
400050' 000001  000003
```

RELATED PSEUDO-OPS

.DIRECTIVE LITLST, END, PRGEND, VAR

COMMON ERRORS

Assembling literals so that some are collapsed on Pass 1, but not on Pass 2. For example, in the following lines, the literals [A] and [B] are collapsed on Pass 1 since they have the same value; but on Pass 2 their values are different and they are not collapsed. This produces a phase error for the label FOO.

```
        IF1,<A=5
              B=5>
        IF2,<A=5
            B=4>
        MOVE AC,[A]
        MOVE AC,[B]
        LIT
FOO:
```

However, literals that have different values in Pass 1 but the same value in Pass 2 do not produce a phase error. For example, the following code generates two words of literal storage in Pass 1. During Pass 2 the values of [A] and [B] are collapsed, but nevertheless MACRO generates two words of literal storage to avoid a phase error at the label FOO.

```
        MOVE AC1,[A]
        MOVE AC1,[B]
        LIT
        A=5
        B=5
FOO:
```

.LNKEND

FORMAT	.LNKEND chain-number,store-address
FUNCTION	Ends a static chain generated at load time. See the <u>LINK Reference Manual</u> (REL Block Type 12) for extensive examples of using .LINK and .LNKEND.)
RELATED PSEUDO-OPS	.LINK
COMMON ERRORS	Chain-number not absolute (A error). EXTERNAL expression for store-address (E error).

LOC

FORMAT

LOC expression

expression = an optional operand whose value gives the address at which sequential address assignment is to continue.

FUNCTION

Sets the location counter to the value of the expression and begins assigning absolute addresses to the instructions and data following the LOC instruction.

If no address is specified, the location counter is restored to its value previous to the last LOC pseudo-op or RELOC-RELOC sequence. (See example below.) If no previous LOC pseudo-op was encountered, the assumed address is 0.

To switch to relocatable address mode, use the pseudo-op RELOC. If no argument is specified, RELOC (in this context) restores the location counter to its value previous to the LOC pseudo-op or LOC-LOC sequence. (An implicit RELOC 0 begins each program.)

If an entire program is to be assigned absolute locations, a LOC statement must precede all instructions and data.

Note that, unlike RELOC-RELOC sequences, typically used to switch between segments in a two-segment program, LOC-LOC sequences cannot be successfully interrupted and then resumed. This is demonstrated in the example below.

EXAMPLES

```
400000'                    TWOSEG 400000   ;Set up hiseg
000000'                    RELOC           ;Back to lowseg
000010                     LOC 10          ;Set up LOC-LOC
000010  000000  000001     DEC 1,2
000011  000000  000002
000100                     LOC 100
000100  000000  000003     DEC 3,4
000101  000000  000004
000012                     LOC
000102                     LOC
000000'                    RELOC           ;Resume RELOC-
                                           ; RELOC
;But we can't resume LOC-LOC
400000'                    RELOC
000102                     LOC
000102                     LOC
400000'                    RELOC           ;But RELOC-
                                           ; RELOC is fine
000102                     LOC
```

RELATED PSEUDO-OPS

RELOC, .ORG, TWOSEG

COMMON ERRORS

Using an EXTERNAL expression for the address expression (E error).

.MFRMT

FORMAT	.MFRMT
FUNCTION	Causes multiformat listing of binary code. The type of instruction assembled determines this format. (See Section 6.1.) .MFRMT is the default setting.
OPTIONAL NOTATIONS	Use the /F switch described in Table 7-1.
RELATED PSEUDO-OPS	.HFRMT

MLOFF

FORMAT

MLOFF

FUNCTION

Terminates each literal at end-of-line even if no closing square bracket is found. This pseudo-op is intended only to maintain compatibility of programs written for very old versions of MACRO.

EXAMPLES

This example shows how MLOFF can be used to interpret [1234 as [1234].

```
                          MLOFF
000000  402001'           [1234
000000  402001'           [1234]
```

OPTIONAL NOTATIONS

Use the /O switch described in Table 7-1.

RELATED PSEUDO-OPS

MLON

MLON

FORMAT	MLON
FUNCTION	Suspends the effect of an earlier MLOFF pseudo-op, thereby enabling the use of multiline literals. MLON is the default setting.
RELATED PSEUDO-OPS	MLOFF

.NODDT

FORMAT	.NODDT symbol,...,symbol
FUNCTION	Suppresses debugger output of each given symbol. Each symbol must have been previously defined. Symbols suppressed with .NODDT can include OPDEF symbols.
EXAMPLES	.NODDT CALL,FJRST,F
OPTIONAL NOTATIONS	Use == for direct-assignment symbols. (See Section 2.4.2.2.) Use :! for label symbols. (See Section 2.4.2.1.)
RELATED PSEUDO-OPS	PURGE
COMMON ERRORS	Using .NODDT with an undefined symbol argument.

NOSYM

FORMAT NOSYM

FUNCTION Suppresses listing of the symbol table in the program listing file.

Suppressing the listing of symbol tables is useful for a library file containing many PRGENDs.

OCT

FORMAT
OCT expression,...,expression

FUNCTION
Defines the local radix for the line as octal; the value of each expression is entered in a fullword of code. The location counter is incremented by 1 for each expression.

EXAMPLES

```
000000  000001          OCT 1,2,20,100
000000  000002
000000  000020
000000  000100
```

OPTIONAL NOTATIONS
Use the EXP pseudo-op and prefix ^O to each expression that must be evaluated in radix 8. In the example above, only the third and fourth expressions, "20,100," could have different evaluations in different radixes. Therefore an equivalent notation is:

```
000000  000001          EXP 1,2,^O20,^O100
000000  000002
000000  000020
000000  000100
```

RELATED PSEUDO-OPS
DEC, EXP, RADIX

OPDEF

FORMAT

OPDEF symbol[expression]

FUNCTION

Defines the symbol as an operator equivalent to expression, giving the symbol a fullword value. When the operator is later used with operands, the accumulator fields are added, the indirect bits are ORed, the memory addresses are added, and the index register addresses are added.

An OPDEF can be declared INTERNAL, using the INTERN pseudo-op. However, if a symbol of the same name exists, the INTERNAL declaration will apply only to that symbol, and not to the OPDEF.

NOTES

1. If you use a relocatable symbol in defining an OPDEF, the value of the symbol may not be the same for all references to the OPDEF.
2. Though the expression portion of an OPDEF must be in square brackets, this use of the brackets is completely unrelated to literals or literal handling.

EXAMPLES

```
200062  000010           OPDEF CAL [MOVE 1,@SYM(2)]
200 02 1 04 000014       CAL 1,BOL(2)
```

The CAL statement is equivalent to:

```
200 02 1 04 000014       MOVE 2,@SYM+BOL(4)
```

RELATED PSEUDO-OPS

DEFINE, SYN

COMMON ERRORS

OPDEF of macroname or SYN symbol (A error).

No code generated by statement in square brackets (A error).

Missing square brackets (A error).

.ORG

FORMAT .ORG address

FUNCTION Sets the location counter to the address and causes the assembler to assign absolute or relocatable addresses depending on the mode of the argument. If A is relocatable, then .ORG A is equivalent to RELOC A; if A is absolute, then .ORG A is equivalent to LOC A.

.ORG with no address sets the location counter to the value it had immediately before the last LOC, RELOC, or .ORG.

EXAMPLES

```
400000'             RELOC 400000    ;Set up some labels
400000'   RELAD1:
000000'             RELOC 0
000000'   RELAD2:
000100              LOC 100
000100    ABSAD1:
400100              LOC 400100
400100    ABSAD2:
000100              LOC ABSAD1      ;Set counter to ABSAD1
                                    ; and begin absolute
                                    ; address assignment.
400000'             RELOC RELAD1    ;Set counter to RELAD1
                                    ; and begin relative
                                    ; address assignment.
400100              .ORG ABSAD2     ;Set counter to ABSAD2
                                    ; and begin absolute
                                    ; address assignment.
400000'             .ORG            ;Set counter to value
                                    ; immediately before
                                    ; last LOC, RELOC, or
                                    ; .ORG, and begin
                                    ; address assignment
                                    ; in appropriate mode.
000000'             .ORG RELAD2     ;Set counter to RELAD2
                                    ; and begin absolute
                                    ; address assignment.
400000'             .ORG            ;Set counter to value
                                    ; immediately before
                                    ; last LOC, RELOC, or
                                    ; .ORG, and begin
                                    ; address assignment
                                    ; in appropriate mode.
000000'             .ORG            ;Set counter to value
                                    ; immediately before
                                    ; last LOC, RELOC, or
                                    ; .ORG, and begin
                                    ; address assignment
                                    ; in appropriate mode.
```

RELATED PSEUDO-OPS LOC, RELOC, TWOSEG

COMMON ERRORS Using an EXTERNAL symbol or complex EXTERNAL expression for the address expression.

PAGE

FORMAT	PAGE
FUNCTION	Causes the assembler to list the current line and then skip to the top of the next listing page. The subpage number is incremented, but the page number is not.
OPTIONAL NOTATIONS	A formfeed character (CTRL/L) in the input text has a similar effect, but increments the page number and resets the subpage number.

PASS2

FORMAT PASS2

FUNCTION Switches the assembler to Pass 2 processing for the remaining code. All code preceding this statement will have been processed by Pass 1 only; all following code by Pass 2 only.

You can use PASS2 to reduce assembly time during debugging; you can also use PASS2 to omit the second pass for a UNIVERSAL file containing only symbol definitions (OPDEFs, macros, and direct assignments).

EXAMPLES Testing a macro defined in the Pass 1 portion:

```
IFE  NON,<
        PRINTX ?HORRIBLE ERROR
        PASS2
        END
>
```

stops assembly if NON = 0.

PHASE

FORMAT

PHASE address

address = an integer expression; cannot be an EXTERNAL symbol.

FUNCTION

Assembles part of a program so that it can be moved to other locations for execution. To use this feature, the subroutine is assembled at sequential relocatable or absolute addresses along with the rest of the program, but the first statement before the subroutine is PHASE, followed by the address of the first location of the block into which the subroutine is to be moved prior to execution. All address assignments in the subroutine are in relation to the address argument. The subroutine is terminated by DEPHASE, which restores the location counter.

EXAMPLES

In the following example, which is the central loop in a matrix inversion, a block transfer instruction moves the subroutine LOOP into accumulators 11 to 15 for execution. (This results in faster execution on KA10 and KI10 processors.)

```
002000' 200 00 0 00 402002'MAIN:  MOVE [XWD LOOPX,LOOP]
002001' 251 00 0 00 000015        BLT LOOP+4
002002' 254 00 0 00 000011        JRST LOOP
000011                     LOOPX: PHASE 11
000011  210 02 0 03 000002 LOOP:  MOVN AC,A(X)
000012  160 02 0 00 000100        FMP AC,MPYR
000013  142 02 0 04 000002        FADM AC,A(Y)
000014  365 03 0 00 000011        SOJGE X,.-3
000015  254 00 0 00 002000'       JRST MAIN
002010'                           DEPHASE
```

The label LOOP represents accumulator 11, and the .-3 in the SOJGE instruction represents accumulator 11.

Note that the code inside the PHASE-to-DEPHASE program segment is loaded into the address following the previous relocatable code; all labels inside the segment, however, have the address corresponding to the phase address. Thus the phased code, if it contains control transfers other than skips, cannot be executed until it has been moved (for example, by a BLT instruction) to the address for which it was assembled.

RELATED PSEUDO-OPS

DEPHASE

COMMON ERRORS

Using an EXTERNAL symbol or complex EXTERNAL expression as the address (E error).

POINT

FORMAT

POINT bytesize,address,bitplace

FUNCTION

Generates a byte pointer word for use with the machine language mnemonics ADJBP, LDB, IBP, ILDB, and IDBP.

Bytesize gives the decimal number of bits in the byte, and is assembled in bits 6 to 11 of the storage word. Address gives the location of the byte word, and is assembled in bits 13 to 35. Bitplace gives the position (in decimal) of the rightmost bit of the byte. MACRO places the value 35 minus bitplace in bits 0 to 5 of the storage word.

If the address is indirect, bit 13 is set. If the address is indexed, the index is placed in bits 14 to 17. The default bytesize is 0. The default bitplace is -1, so that the byte increment instructions IBP, ILDB, and IDBP will begin at the left of the address word.

EXAMPLES

```
36 06 0 00 000000        POINT 6,0,5
44 06 0 00 000100        POINT 6,100
```

COMMON ERRORS

Bytesize or bitplace not given in decimal.

Bytesize or bitplace not absolute.

Bytesize or bitplace EXTERNAL.

PRGEND

FORMAT

PRGEND

FUNCTION

Replaces the END statement for all except the last program of a multiprogram assembly. PRGEND closes the local symbol table for the current module.

You can use PRGEND to place several small programs into one file to save space and disk accesses. The resulting binary file can be loaded in search mode. (See the LINK Reference Manual.)

Using PRGEND requires extra memory for assembly, since the tables for each program must be saved for Pass 2. Functionally, however, PRGEND is identical to END, except that PRGEND does not end the current assembly pass.

NOTE

1. PRGEND is not allowed in macros or PSECTs.
2. PRGEND clears the TWOSEG pseudo-op.
3. Like END, PRGEND causes assembly of all unassembled literals and variable symbols.
4. In a file containing PRGENDs, using more than one LIT pseudo-op in any but the last program produces unpredictable results.

OPTIONAL NOTATIONS

Give an argument with PRGEND, specifying the start address for the program. See the END pseudo-op for a discussion of this argument and its meaning.

RELATED PSEUDO-OPS

END, LIT, VAR

COMMON ERRORS

Failing to end a text string, REPEAT, conditional code, DEFINE, or literal with a closing delimiter; MACRO cannot see any following PRGEND or END.

Confusing multiprogram and multifile assemblies. A multiprogram assembly involves multiple programs separated by PRGENDs. A multifile assembly always involves multiple files separated by end-of-file. The two types of assemblies are not mutually exclusive.

PRINTX

FORMAT

PRINTX text

FUNCTION

Causes text to be output during assembly. On Pass 1 the text is output to the terminal and the listing device. On Pass 2 the text is output to the terminal, but only if the terminal is not the listing device.

PRINTX is frequently used to output conditional information and, in very long assemblies, to report progress of the assembler through Pass 1.

EXAMPLES

```
PRINTX ASSEMBLER HAS REACHED POINT NOWGO

IFGE .-1000,<PRINTX CODE MORE THAN 1P>
```

.PSECT

FORMAT

.PSECT name/attribute,origin

name = a valid symbol giving the name of the PSECT.

attribute = either CONCATENATE or OVERLAID.

origin = an expression giving an address for the PSECT origin.

FUNCTION

Specifies the relocation counter to be used for the code following. MACRO generates a REL Block Type 23. (See the LINK Reference Manual.)

Do not use PRGEND and .PSECT in the same file. MACRO will treat the first PRGEND as an END statement and ignore any following source code.

For a complete discussion of PSECTS and their handling, see Section 9.1.3.

EXAMPLES

```
.PSECT A/CONCATENATE,0

.PSECT FIRST/OVERLAID,1000
```

OPTIONAL NOTATIONS

Omit attribute (defaults to CONCATENATE).

RELATED PSEUDO-OPS

.ENDPS, LOC, .ORG, RELOC, TWOSEG

COMMON

Using TWOSEG and .PSECT in the same module.

Using HISEG and .PSECT in the same module.

PURGE

FORMAT

PURGE symbol,...,symbol

symbol = an assigned symbol, a label, an operator, or a macro name.

FUNCTION

Deletes symbols from the symbol tables. Normally used at the end of a program to conserve storage and to delete symbols for the debugger. Purged symbol table space is reused by the assembler.

If you use the same symbol for both a macro name or OPDEF and a label, a PURGE statement deletes the macro name or OPDEF. Repeating the instruction then purges the label.

Purging a symbol that is EXTERNAL or undefined suppresses any error messages associated with it.

EXAMPLES

```
000040  000001          LABEL:  1,1
                        PURGE LABEL
```

RELATED PSEUDO-OPS

.NODDT, XPUNGE

RADIX

FORMAT

RADIX expression

expression = decimal value from 2 to 10

FUNCTION

Sets the radix to the value of expression. An implicit RADIX 8 statement begins every MACRO program.

All numerical expressions that follow (up to the next RADIX pseudo-op) are interpreted in the given radix unless another local radix is indicated. (A different local radix for the line can be indicated by the DEC or OCT pseudo-ops; a different local radix for an expression can be indicated by ^B, ^D, or ^O. See Section 2.2.2.)

Ordinarily, numbers outside the range of the given radix are not interpreted. For example, in radix 8, the number 99 causes an error. However, a single-digit number is interpreted in any case. For example, in radix 8, the number 9 is recognized as octal 11.

EXAMPLES

```
                        RADIX 10
000000  000012          EXP 10
                        RADIX 8
000000  000010          EXP 10
```

OPTIONAL NOTATIONS

Use one of the following prefixes to change the radix for a single expression: ^B for binary, ^O for octal, ^D for decimal.

RELATED PSEUDO-OPS

DEC, OCT

COMMON ERRORS

Using a relocatable expression (A error).

Using an external expression (E error).

Giving a radix argument not in the range 2 to 10 decimal (A error).

Misusing numbers in a given radix; for example, in the statements

```
RADIX 10
RELOC 400000
```

MACRO treats the number 400000 as decimal.

RADIX50

FORMAT

RADIX50 code,symbol

FUNCTION

Packs the symbol into bits 4 to 35 of the storage word, with the code in bits 0 to 3.

The "50" in RADIX50 is octal, so that the radix in decimal is 40. The 40 characters permitted in symbols are the "digits" of the RADIX50 symbol expression. Thus a symbol is seen by RADIX50 as a "6-digit" number in base 40, converted to binary, and placed in bits 4 to 35 in storage.

The code expression for RADIX50 is a number in the range 0 to 74 octal. Its binary equivalent should end with two zeros (that is, the octal should end with 0 or 4), since the two low-order bits will not be stored. The four high-order bits are placed in bits 0 to 3 in storage.

See Appendix A for the octal values of RADIX50 characters.

EXAMPLES

```
126633   472376   RADIX50 10,SYMBOL
466633   472376   RADIX50 44,SYMBOL
```

OPTIONAL NOTATIONS

The mnemonic SQUOZE can be used in place of RADIX50.

RADIX50 ,symbol (code is taken as zero).

RELATED PSEUDO-OPS

SQUOZE

COMMON ERRORS

RADIX50 code not absolute (A error).

RADIX50 code does not end with 0 or 4 (Q error).

RELOC

FORMAT

RELOC expression

expression = an optional operand that specifies the address at which sequential address assignment is to continue.

FUNCTION

Sets the location counter to the value of expression, and begins assigning relocatable addresses to the instructions and data that follow.

In a PSECTed program, RELOC sets the location counter for the current PSECT.

If no address is specified, the location counter is restored to its value before the last RELOC, or before the last LOC-LOC sequence, whichever is later. (See the first example below.) If no previous RELOC or LOC-LOC sequence was encountered, the location counter is set to 0.

An implicit RELOC 0 begins every MACRO program. To switch to absolute address mode, use the pseudo-op LOC.

Note that RELOC-RELOC sequences (typically used to switch between segments in a two-segment program) can be interrupted and then resumed. This is demonstrated in the first example below.

EXAMPLES

```
400000'                          TWOSEG 400000 ;Set up hiseg
000000'                          RELOC         ;Back to lowseg
000000' 000000  000001           DEC 1,2
000001' 000000  000002
400000'                          RELOC         ;Back to hiseg
400000' 255 00 0 00 000000       JFCL
000137                           LOC 137       ;Deposit version
                                               ; in absolute 137
000137  000100  000001           XWD 100,1
400001'                          RELOC         ;Back to hiseg
                                               ; where left off
400001' 254 00 0 00 400000'      JRST .-1
000002'                          RELOC         ;Back to lowseg
```

RELATED PSEUDO-OPS

LOC, .ORG, TWOSEG

(Continued on next page)

RELOC (Cont.)

COMMON ERRORS

Using an EXTERNAL symbol or complex EXTERNAL expression as the address.

Returning to the wrong segment when using RELOC with TWOSEG. The last four lines of the following example show how this can occur:

```
400000'                         TWOSEG
400000'                         RELOC 400000 ;Sets first RELOC
                                             ; counter to
                                             ; 400000'
000000'                         RELOC 0      ;Saves 400000',
                                             ; sets to 000000'
400000'                         RELOC        ;Swaps counters
400000' 000000  000001          EXP 1,2      ;Enter values here
400001' 000000  000002
000000'                         RELOC        ;Swaps again
000000' 000000  000003          EXP 3,4      ;More values here
000001' 000000  000004
400002'                         RELOC        ;Swaps again
400000'                         RELOC 400000 ;Lost counter
                                             ; to 000002'
400002'                         RELOC        ;Swaps again
400000'                         RELOC        ;Swaps again
400000' 000000  000001          EXP 1        ;Overwrites 400000'
```

REMARK

FORMAT	REMARK text
FUNCTION	Text is a comment.
EXAMPLES	REMARK I CAN SAY ANYTHING HERE.
OPTIONAL NOTATIONS	A comment line can also begin with a semicolon.
RELATED PSEUDO-OPS	COMMENT
COMMON ERRORS	Continuing REMARK text to next line without using the continuation character (CTRL/underscore).

REPEAT

FORMAT

REPEAT expression,<code>

expression = the repeat index, which gives the number of times to repeat assembly of the code given; the repeat index can be any expression having a nonnegative integer value.

FUNCTION

Generates the code given in angle brackets n times. REPEAT statements can be nested to any level.

Line continuation is not allowed across end-of-REPEAT, since a carriage return and linefeed are appended to each expansion of the code.

Note that REPEAT 0,<code> is logically equivalent to a false conditional, and REPEAT 1,<code> is logically equivalent to a true conditional.

EXAMPLES

```
                      000000          COUNT=0
                              TABLE:  REPEAT 4,<COUNT
                              COUNT=COUNT+1>
002020' 000000        000000  COUNT
                      000001  COUNT=COUNT+1
002021' 000000        000001  COUNT
                      000002  COUNT=COUNT+1
002022' 000000        000002  COUNT
                      000003  COUNT=COUNT+1
002023' 000000        000003  COUNT
                      000004  COUNT=COUNT+1

                              REPEAT 3,<.>
002024' 000000        002024' .
002025' 000000        002025' .
002026' 000000        002026' .
```

RELATED PSEUDO-OPS

DEFINE, IRP, IPRC

COMMON ERRORS

No comma after n (A error).

Using an EXTERNAL symbol or complex EXTERNAL expression as the repeat index.

Mismatching angle brackets.

.REQUEST

FORMAT
.REQUEST filespec

FUNCTION
Causes the specified file to be loaded only to satisfy a global request; that is, the file is loaded in library search mode. (See Chapter 7 for a discussion of files.)

The filespec must not include a file extension. If you specify a path, only the project-programmer number is allowed; SFDs are not allowed.

MACRO generates a REL Block Type 17. (See the LINK Reference Manual.)

EXAMPLES

```
.REQUEST DSK:MACROS
.REQUEST MACROS
```

OPTIONAL NOTATIONS
DSK: is the default device.

Your default path at load time is the default path.

RELATED PSEUDO-OPS
.REQUIRE, .TEXT

.REQUIRE

FORMAT

.REQUIRE filespec

FUNCTION

Causes the specified file to be loaded automatically, independent of any global requests. (See Chapter 7 for discussion of files.)

The filespec must not include a file extension. If you specify a path, only the project-programmer number is allowed; SFDs are not allowed.

MACRO generates a REL Block Type 16. (See the LINK Reference Manual.)

EXAMPLES

```
.REQUIRE DSK:MACROS
.REQUIRE MACROS
.REQUIRE SYS:MACREL
```

OPTIONAL NOTATIONS

DSK: is the default device.

Your default path at load time is the default path.

RELATED PSEUDO-OPS

.REQUEST, .TEXT

SALL

FORMAT	SALL
FUNCTION	Causes suppression of all macro and repeat expansions and their text; only the input file and the binary generated will be listed. SALL can be nullified by either XALL or LALL. Using SALL generally produces the tidiest listing file.
OPTIONAL NOTATIONS	Use the /M switch described in Table 7-1.
RELATED PSEUDO-OPS	LALL, LIST, XALL, XLIST

SEARCH

FORMAT

SEARCH tablename(filename),...,tablename(filename)

FUNCTION

Defines a list of symbol tables for MACRO to search if a symbol is not found in the current symbol table. A maximum of ten tables can be specified. Tables are searched in the order specified.

When the SEARCH pseudo-op is seen, MACRO checks its internal UNIVERSAL table for a memory-resident UNIVERSAL of the specified name. (See the UNIVERSAL pseudo-op for further discussion of memory-resident UNIVERSAL tables and use of the /U switch.)

If no such entry is found in the UNIVERSAL table, MACRO reads in the symbol table using the given file specification. If no file specification is given, MACRO reads tablename.UNV from the default path. If no such file is found, MACRO then tries UNV:tablename.UNV and SYS:tablename.UNV, in that order.

When all the specified files are found, MACRO builds a table for the search sequence. If MACRO cannot find a given symbol in the current symbol table, the UNIVERSAL tables are searched in the order specified. When the symbol is found, it is moved into the current symbol table. This procedure saves time (at the expense of core) on future references to the same symbol.

A UNIVERSAL file can search other UNIVERSAL files, provided all names in the search list have been assembled.

The internal table of UNIVERSAL names is cleared on each run (.R MACRO) or START command, but is not cleared when MACRO responds with an asterisk.

In a PSECTed program, all UNIVERSAL symbols belong to the blank PSECT.

EXAMPLES

```
SEARCH MONSYM,MACSYM
```

OPTIONAL NOTATIONS

Omit the filename and its enclosing parentheses. MACRO then looks on DSK:, UNV:, and SYS: (in that order) for tablename.UNV.

RELATED PSEUDO-OPS

UNIVERSAL

COMMON ERRORS

Not purging a macro that redefines itself (P error). If a macro is found in a universal file, the definition is copied into the current macro table and the auxiliary table is not searched on Pass 2. Thus, a macro that redefines itself can cause P errors similar to enclosing the macro by IF1. Such macros should be purged before Pass 2.

SIXBIT

FORMAT

SIXBIT dtextd

d = delimiter; first nonblank character, whose second appearance terminates the text.

FUNCTION

Enters strings of text characters in 6-bit format. Six characters per word are left justified in sequential storage words. Any unused bits are set to zero.

Lowercase letters in SIXBIT text strings are treated as uppercase. Otherwise, only the SIXBIT character set is allowed. (See Appendix A for SIXBIT characters and their octal codes.)

EXAMPLES

```
64 45 70 64 00 63   SIXBIT \TEXT STRING\
64 62 51 56 47 00

644570  640000      EXP SIXBIT /TEXT/
```

OPTIONAL NOTATIONS

Omit the space or tab after SIXBIT. This is not allowed if the delimiter is a letter, number, dot, dollar sign, or percent sign (that is, a possible symbol constituent), or if the ASCII value of the delimiter character is less than 040 or greater than 172.

Right-justified SIXBIT can be entered by using single quotes to surround up to six characters; for example,

```
006251  475064      'RIGHT'
```

RELATED PSEUDO-OPS

ASCII, ASCIZ, .DIRECTIVE FLBLST

COMMON ERRORS

Using the delimiter character in the text string.

Missing the end delimiter (that is, attempting to use a carriage return as a delimiter).

Using more than six characters in a right-justified SIXBIT string, or more than three characters if in the address field (Q error).

Using non-SIXBIT characters in the text string.

SQUOZE

FORMAT	SQUOZE code,symbol
FUNCTION	SQUOZE is a mnemonic for RADIX50.
EXAMPLES	126633 472376 RADIX50 10,SYMBOL 126633 472376 SQUOZE 10,SYMBOL
OPTIONAL NOTATIONS	RADIX50 code,symbol SQUOZE ,symbol (code is taken as 0).
RELATED PSEUDO-OPS	RADIX50
COMMON Errors	Code not absolute (A error). Code does not end with 0 or 4 (Q error).

STOPI

FORMAT STOPI

FUNCTION Ends an IRP or IRPC before all subarguments or characters are used. The current expansion is completed, but no new expansions are started. STOPI can be used with conditionals inside IRP or IRPC to end the repeat if the given condition is met.

EXAMPLES

```
                    LALL
                    DEFINE ONETWO(A)<
                            IRP A,<IFIDN<A><ONE>,<STOPI
                                            EXP 1>>
                            IRP A,<IFIDN<A><TWO>,<STOPI
                                            EXP 2>>
                    >
                    ONETWO <A,B,D>^
                            IRP
                    IFIDN<A><ONE>,<STOPI
                                            EXP 1>
                    IFIDN<B><ONE>,<STOPI
                                            EXP 1>
                    IFIDN<D><ONE>,<STOPI
                                            EXP 1>

                            IRP
                    IFIDN<A><TWO>,<STOPI
                                            EXP 2>
                    IFIDN<B><TWO>,<STOPI
                                            EXP 2>
                    IFIDN<D><TWO>,<STOPI
                                            EXP 2>

                    ^
                    ONETWO <A,ONE,B,ONE,TWO>^
                            IRP
                    IFIDN<A><ONE>,<STOPI
                                            EXP 1>
                    IFIDN<ONE><ONE>,<STOPI
000000  000001                              EXP 1>

                            IRP
                    IFIDN<A><TWO>,<STOPI
                                            EXP 2>
                    IFIDN<ONE><TWO>,<STOPI
                                            EXP 2>
                    IFIDN<B><TWO>,<STOPI
                                            EXP 2>
                    IFIDN<ONE><TWO>,<STOPI
                                            EXP 2>
                    IFIDN<TWO><TWO>,<STOPI
000000  000002                              EXP 2>
```

RELATED PSEUDO-OPS IRP, IRPC

COMMON ERRORS STOPI not inside IRP or IRPC.

SUBTTL

FORMAT SUBTTL subtitle

FUNCTION Defines a subtitle (of up to 80 characters) to be printed at the top of each page of the listing file until the end-of-listing or until another SUBTTL statement is found.

The initial SUBTTL usually appears on the second line of the first page of the input file, immediately following the TITLE statement.

For subsequent SUBTTL statements, the following rule applies: if the new SUBTTL is on the first line of a new page, then the new subtitle appears on that page; if not, the new subtitle appears on the next page.

NOTE

The statements

```
. . .
PRGEND
TITLE FOO
SUBTTL BAR
```

do not cause BAR to appear as the subtitle on the first page of the listing of FOO.

SUBTTL affects only the listing file, and subtitles can be changed as often as desired.

EXAMPLES

```
SUBTTL THIS SECTION CONTAINS DEVICE-DEPENDENT ROUTINES
```

RELATED PSEUDO-OPS TITLE

SUPPRESS

FORMAT

SUPPRESS symbol,...,symbol

FUNCTION

Turns on a suppress bit in the symbol table for the specified symbols. The suppress bit will be turned off for any symbol later referenced in the program. Symbols whose suppress bits are on at the end of assembly are not listed in the symbol table, but will be listed in any tables built by CREF unless they are XCREFed.

When an appended parameter file (as opposed to a UNIVERSAL file) is used in an assembly, many symbols may be defined but never used. These take up space in the binary file and complicate listing of the file.

Unused and unwanted symbols can be removed from tables by SUPPRESS or ASUPPRESS. These pseudo-ops control the suppress bit in each entry of the symbol table; if the bit is on, the symbol in that location is not output.

RELATED PSEUDO-OPS

ASUPPRESS

COMMON ERRORS

Attempting to suppress an undefined symbol.

SYN

FORMAT

SYN sym1,sym2

sym1 = a defined symbol.

sym2 = a symbol to be defined as synonymous with sym1.

FUNCTION

Defines sym2 as synonymous with sym1.

If sym1 is defined as both a label and an operator, sym2 assumes the label definition.

EXAMPLES

The following are legal SYN statements:

```
SYN X,K
SYN FAD,ADD
SYN END,XEND
```

To turn XLIST into a null operator,

```
DEFINE .XL < >
SYN .XL,XLIST
```

To restore its operation,

```
PURGE XLIST
```

RELATED PSEUDO-OPS

DEFINE, OPDEF

COMMON ERRORS

Missing symbol (A error).

Unknown symbol - first operand not defined (A error).

Missing comma (A error).

Using a variable as one of the symbol arguments (A error).

TAPE

FORMAT TAPE

FUNCTION Causes the assembler to begin assembling the program contained in the next source file in the MACRO command string.

EXAMPLES (Interactive)

```
.R MACRO
*DSK:BINAME,LPT:=TTY:,DSK:MORE
PARAM=6
TAPE
;THIS COMMENT WILL BE IGNORED
^Z
```

This sets PARAM to 6 and assembles the remainder of the program from the source file DSK:MORE. Since MACRO is a two-pass assembler, the TTY: file must be repeated for Pass 2.

```
[MCREP1 END OF PASS 1]
PARAM=6
TAPE
^Z
```

Note that all text after the TAPE pseudo-op is ignored.

TEXT

FORMAT

.TEXT dtextd

d = delimiter; first nonblank character, whose second appearance terminates the text.

FUNCTION

Generates an ASCIZ REL Block Type for LINK and inserts the text string directly into the .REL file output as a separate block. (See the LINK Reference Manual.)

The text inserted in the .REL file is interpreted as a command string for LINK. Therefore a MACRO program loaded by user commands to LINK can contain additional LINK commands, carried out when the MACRO program is loaded.

EXAMPLES

.TEXT '/SET:.HIGH.:500000'

OPTIONAL NOTATIONS

Omit the space or tab after .TEXT. This is not allowed if the delimiter is a letter, number, dot, dollar sign, or percent sign (that is, a possible symbol constituent), or if the ASCII value of the delimiter character is less than 040 or greater than 172.

RELATED PSEUDO-OPS

.REQUEST, .REQUIRE

COMMON ERRORS

Using the delimiter character in the text string.

Missing the end delimiter (that is, attempting to use a carriage return as a delimiter).

TITLE

FORMAT

TITLE title

FUNCTION

Gives the program name and a title to be printed at the top of each page of the program listing.

The first characters (up to six characters, or up to the first non-RADIX50 character) are the program name. This name is used when debugging with DDT to gain access to the program's symbol table.

The entire text of the title is printed on each page of the program listing.

Only one TITLE statement is allowed in a module; programs with PRGEND statements can use one TITLE statement for each module.

A TITLE statement can appear anywhere in the program; it usually appears as the first line of the program.

If no TITLE statement is used, the assembler inserts the program name ".MAIN".

EXAMPLES

```
TITLE FLOATING-POINT NUMBER PACKAGE
```

The program name is FLOATI; the words FLOATING-POINT NUMBER PACKAGE will appear at the head of each page and subpage of the listing.

RELATED PSEUDO-OPS

SUBTTL, UNIVERSAL

COMMON ERRORS

Using more than one TITLE in a program.

Using TITLE and UNIVERSAL in the same module (M error).

TWOSEG

FORMAT

TWOSEG expression

expression = any expression giving a nonnegative value as the beginning of the program high segment; cannot be EXTERNAL.

FUNCTION

Directs MACRO to assemble a two-segment program with the high segment beginning at the given address. MACRO sets the location counter to the given address, and generates a REL Block Type 3, which tells LINK to expect two segments. (The address is reduced to the next lower multiple of 2000 (octal). If this result is 0, the address defaults to 400000.)

Only one TWOSEG pseudo-op is allowed in a program.

High-segment code is controlled by using RELOC with a value at least as large as the TWOSEG address. Low-segment code is controlled by smaller RELOC values.

NOTE

Using TWOSEG without an argument sets the beginning address for the high segment to 400000. However, this does not set the location counter to 400000.

EXAMPLES

```
        TWOSEG
        RELOC 0
DATA:   BLOCK 10000       ;Low segment
        RELOC 400000
START:  EXIT 0            ;High segment
```

RELATED PSEUDO-OPS

LOC, .ORG, RELOC

COMMON ERRORS

Using an EXTERNAL symbol or complex EXTERNAL expression as the address argument.

Using TWOSEG more than once in a program (Q error).

Generating relocatable code before the TWOSEG pseudo-op (Q error).

Using PSECT and TWOSEG in the same program.

UNIVERSAL

FORMAT

UNIVERSAL tablename

FUNCTION

Declares the symbol table of the current program available to other programs, and stores the given tablename in MACRO's internal UNIVERSAL table. The tablename is also taken as the program name, and appears in the heading of each page of the listing file.

When an END or PRGEND statement is found, the symbol table is placed immediately after the assembler's pushdown stacks and buffers. In addition to this memory-resident copy of the UNIVERSAL symbol table, the file tablename.UNV is generated. (This file can be suppressed by the /U switch described in Table 7-1.)

UNIVERSAL files can be used to generate data, but are more commonly used to generate symbols, macros, and OPDEFs. The symbols and OPDEFs generated in a UNIVERSAL program need not be declared INTERNAL, since its local symbols are available to accessing programs. (See the SEARCH pseudo-op.)

Memory-resident UNIVERSAL symbol tables are cleared on each run (.R MACRO) or START, but are not cleared when MACRO responds with an asterisk. This saves redundant lookups when many programs search a common set of UNIVERSALs.

Note that if a sequence of programs (or even one program) searches more than ten UNIVERSAL symbol tables, a SEARCH table overflow occurs. This overflow forces reinitialization of the assembler by a run (.R MACRO) or START command.

For a UNIVERSAL program that does not generate data (that is, it has only symbol, macro, and OPDEF definitions), you can save time by using 1-pass assembly. However, such a file must not contain forward references to symbol definitions.

A UNIVERSAL file cannot contain PSECTs.

(Continued on next page)

UNIVERSAL (Cont.)

NOTES

1. For COMPILE-class commands, the existence of the file tablename.REL may prevent recompilation of the UNIVERSAL file tablename.MAC. To avoid this, force compilation of the .MAC file by including /COMPIL in the command string.

2. Generally, a UNIVERSAL file need not be reassembled when referencing programs are assembled with newer versions of MACRO. However, if the UNIVERSAL's assembler version is newer than the program's, you may get the MCRUVS message, indicating skewed UNIVERSAL versions. In this case, reassembly or one or both files is required (using the same assembler version).

EXAMPLES

```
UNIVERSAL S1
START=765
AC1=1
F=0
END
```

RELATED PSEUDO-OPS: SEARCH, TITLE

COMMON ERRORS: Using TITLE and UNIVERSAL in the same module (M error).

VAR

FORMAT
VAR

FUNCTION
Causes variable symbols (defined in previous statements by suffixing the number sign (#), or by ARRAY or INTEGER statements) to be assembled as BLOCK statements. This has no effect on subsequent definitions of symbols of the same type.

If the VAR statement does not appear in the program, all variables are stored at the end of the program.

If the pseudo-op TWOSEG is used, the variables reserved by an array statement must be assigned to the low segment; thus a RELOC back to the low segment is required before using the VAR pseudo-op.

EXAMPLES

```
402003' 201 01 0 01 000000      ADD2:   MOVEI 1,0(1)
402004' 202 01 0 00 402012'             MOVEM 1,FIRST#
402005' 201 02 0 02 000000              MOVEI 2,0(2)
402006' 202 02 0 00 402013'             MOVEM 2,SECOND#
402007' 140 01 0 00 000002              ADD 1,2
402010' 200 01 0 00 402014'             MOVE 1,SUM#
402011' 263 17 0 00 000000              POPJ 17,
001052'                                 RELOC
001052'                                 VAR
001055'                                 BLOCK 2
```

RELATED PSEUDO-OPS
ARRAY, BLOCK, INTEGER

XALL

FORMAT	XALL
FUNCTION	Resumes standard listing after previous LALL or SALL. (XALL is the default among these three.)

XALL suppresses all lines of the program listing file that do not generate binary code.

XALL does not suppress REPEAT expansions.

NOTE

> Under XALL only one listing line is output for each source line generating binary code in a macro expansion. Occasionally, a single line of a macro definition expands into several lines of listing text. When this occurs, part of a binary-generating source line may not be listed.
>
> You can avoid this by temporarily setting the listing mode to LALL (list all) or SALL (suppress all) around such lines.

RELATED PSEUDO-OPS	LALL, LIST, SALL, XLIST
OPTIONAL NOTATIONS	Use the /X switch described in Table 7-1.

.XCREF

FORMAT	.XCREF symbol,...,symbol
FUNCTION	Suspends output of cross-referencing for the specified symbols. References to these symbols between this statement and the next .CREF or the end of the program will not appear in the cross-reference listing.
OPTIONAL NOTATIONS	.XCREF If no symbol names are specified, MACRO suspends cross-referencing for all symbols.
RELATED PSEUDO-OPS	.CREF
COMMON ERRORS	Specifying a nonexistent symbol (A error).

XLIST

FORMAT XLIST

FUNCTION Suspends output to the program listing file. This output occurs only in Pass 2; XLIST does not affect Pass 1. To resume output, use the pseudo-op LIST.

EXAMPLES The following sequence of code shows an XLIST pseudo-op suppressing listing of literals:

```
        EXIT    ;End of program
        XLIST   ;Don't list literals
        LIT
        LIST
        END
```

This sequence of code lists as:

```
401023' 104 00 0 00 000170  HALTF   ;End of program
                            XLIST   ;Don't list literals
                            LIST
                            END
```

Note that the high-segment break will be greater than 401023' because the literals are assembled after the HALTF.

RELATED PSEUDO-OPS LALL, LIST, SALL, XALL

OPTIONAL NOTATIONS Use the /S switch described in Table 7-1.

XPUNGE

FORMAT	XPUNGE
FUNCTION	Deletes all local symbols during Pass 2. This reduces the size of the .REL file and speeds up loading. XPUNGE should immediately precede the END statement.
RELATED PSEUDO-OPS	PURGE

XWD

FORMAT

XWD lefthalf,righthalf

FUNCTION

Enters two halfwords in a single storage word. Each half is formed in a 36-bit register, and the low-order 18 bits are placed in the halfword. The high-order bits are ignored.

XWD statements are used to set up pointer words for block transfer instructions. Block transfer pointer words contain two 18-bit addresses; the left half is the starting location of the block to be moved, and the right half is the first location of the destination.

EXAMPLES

```
402017' 200 02 0 00 403040'       MOVE 2,[XWD FROM1,TO1]
402020' 251 02 0 00 403035'       BLT 2,TOEND1
......                            . . .
402636'                   FROM1:  BLOCK 100
402736'                   TO1:    BLOCK 100
                403035'           TOEND1=.-1
```

OPTIONAL NOTATIONS

lefthalf,,righthalf

BYTE (18)lefthalf,righthalf

COMMON ERRORS

Using halfword with absolute value larger than 18 bits (Q error).

Using two commas between the arguments to XWD. For example, XWD A,3 is correct; XWD A,,3 is incorrect.

Z

FORMAT Z accumulator, address

FUNCTION Z is treated as if it were the null machine language mnemonic. An instruction word is formed with zeros in bits 0 to 8. The rest of the word is formed from the accumulator and address. (See Section 4.7.1.)

EXAMPLES

```
403036' 000 00 0 00 000000     Z
403037' 000 01 0 04 000002     Z 1,2(4)
```

CHAPTER 4

MACRO STATEMENTS AND STATEMENT PROCESSING

A MACRO statement has one or more of the following: a label, an operator, one or more operands, and a comment. The general form of a MACRO statement is:

```
label:  operator operand,operand    ;comment
```

A carriage return ends the statement.

NOTES

1. Direct-assignment statements receive special handling. (See Section 2.4.2.2.)

2. Processing of macros is not discussed here because a macro call produces a text substitution. After substitution, the text is processed as described in this chapter. Macros are discussed in Chapter 5.

4.1 LABELS

A label is always a symbol with a suffixed colon. (See Section 2.4.2.1.) The assembler recognizes a label by finding the colon. If a statement has labels (you can use more than one), they must be the first elements in the statement.

A label can be defined only once; its value is the address of the first word of code generated after it.

Since a label gives an address, the label can be either absolute or relocatable. A label is a local symbol by default. You can declare a label INTERNAL global or EXTERNAL global. (See Section 2.4.5.)

4.2 OPERATORS

After processing any labels, the assembler views the following nonblank, nontab characters as a possible operator. An operator is one of the following:

1. A MACRO-defined mnemonic. All mnemonics are listed in Appendix C, and are discussed in the Hardware Reference Manual.

2. A user-defined operator. (See the pseudo-op OPDEF in Chapter 3.)

3. A pseudo-op. (See Chapter 3.)

If the characters found do not form one of the above, then MACRO views them as an expression.

An operator is ended by the first non-RADIX50 character: if it is ended by a blank or tab, operands may follow; if it is ended by a semicolon, there are no operands and the comment field begins; if it is ended by a carriage return, the statement ends and there are no operands or comments.

4.3 OPERANDS

After processing labels and the operator, if any, the assembler views as operands all characters up to the first unquoted semicolon or carriage return. Commas delimit the operands.

The operator in a statement determines the number (none, one, two or more) and kinds of permitted or required operands. Any expected operand not found is interpreted as null. An operand can be any expression or symbol appropriate for the operator.

4.4 COMMENTS

The first unquoted semicolon in a statement begins the comment field. You can use any ASCII characters in a comment; however, angle brackets in a comment may produce unpredictable results. You can continue a comment to the next line by typing CTRL/ , followed by a carriage return.

If the first nonblank, nontab character in a line is a semicolon, the entire line is a comment. You can also enter a full line of comment with the pseudo-op REMARK, or a multiline comment with the pseudo-op COMMENT. (See Chapter 3.)

Comments do not affect binary program output.

4.5 STATEMENT PROCESSING

MACRO processes your program as a linear stream of data. During Pass 1, MACRO may find references to symbols not yet defined. These symbols are entered in the user symbol table. Whenever a symbol is defined, it is entered in the table with its value, so that on Pass 2 all definitions can be found in the table. The values then replace the symbols in the binary code generated.

NOTE

> Delayed definition is allowed only for labels and direct-assignment symbols. A symbol that contributes to code generation (for example, an OPDEF, a macro, or a REPEAT index) must be defined before any reference to it.

Statement processing proceeds as follows:

1. Labels are found and entered in the user symbol table.
2. The next characters up to the first unquoted semicolon, blank, tab, comma, or equal sign are processed.
   a. Equal sign: the characters form a symbol, and the following characters form an expression. The symbol and the value of the expression are entered in the user symbol table.
   b. Other delimiter: the characters form an expression or an operator. If an operator, it is found in a table and assembled. If an expression, its value is assembled.
3. If the operator takes operands, the next characters up to the first unquoted semicolon or carriage return form operands. Unquoted commas delimit operands. For each operand, leading and trailing blanks and tabs are ignored. Operands are evaluated and assembled for the given operator.
4. The first unquoted semicolon ends processing of the line. Any further characters up to the first carriage return are comment.
5. The first unquoted carriage return ends the statement. Any following characters begin a new statement.

4.6 ASSIGNING ADDRESSES

MACRO normally (and by default) assembles statements with relocatable addresses. Assembly begins with the zero storage word and proceeds sequentially. Each time MACRO assembles a word of binary code, it increments its location counter by 1.

A mnemonic operator generates one word of binary code. Direct-assignment statements and some pseudo-ops do not generate code. Some pseudo-ops generate more than one word of code.

You can control address assignment by setting the assembler's location counter using the pseudo-ops LOC and RELOC. (See Section 9.1.)

You can also reference addresses relative to the location counter by using the dot symbol (.). For example, the expression .-1 used as an address refers to the location immediately preceding the current location.

In revising MACRO programs, you can cause an incorrect address to be assembled by adding or removing statements within the range of a .+n expression. For example, in the sequence

```
000000' 332 00 0 01 000000      SKIPE 0(AC)
000001' 254 00 0 00 001020'     JRST GOTONE
000002' 344 01 0 00 000000'     AOJA AC,.-2
```

the expression .-2 gives the address of the SKIPE statement. If you revise this sequence by inserting a statement, you should change the expression to .-3 so that it still refers to the correct statement.

```
000000' 332 00 0 01 000000      SKIPE 0(AC)
000001' 254 00 0 00 001020'     JRST GOTONE
000002' 350 00 0 00 000014      AOS NULCNT       ;Added line
000003' 344 01 0 00 000000'     AOJA AC,.-3      ;Changed line
```

For this reason, use great care with such expressions other than .+1 and .-1. Using labels avoids this problem entirely.

4.7 MACHINE INSTRUCTION MNEMONICS AND FORMATS

There are two kinds of machine instruction mnemonics: primary and input/output. Primary instructions generate binary code in primary instruction format; input/output instructions generate binary code in input/output instruction format.

4.7.1 Primary Instructions

A primary instruction is in one of the forms

```
mnemonic accumulator,address
mnemonic accumulator,
mnemonic address
```

where mnemonic is a machine instruction mnemonic, accumulator is an accumulator register address, and address is a memory address. The memory address can be modified by indexing, indirect addressing, or both.

A complete list of machine instruction mnemonics and their octal codes is given in Appendix C, and these mnemonics are discussed in the Hardware Reference Manual.

The accumulator address gives the address of a register, and can be any expression or symbol whose value is an integer in the range 0 to 17 octal.

The memory address gives a location in memory, and can be any expression or symbol whose value is an integer in the range 0 to octal 777777.

You can modify the memory address by indirect addressing, indexed addressing, or both. For indirect addressing, prefix an at sign (@) to the memory address in your program. For indexed addressing, suffix an index register address in parentheses to the memory address in your program. This address can be any expression or symbol whose value is an integer in the range 1 to octal 17.

NOTE

> To assemble the index, MACRO places the index register address in a fullword of storage, swaps its halfwords, and then adds the swapped word to the instruction word.

For an example of a primary instruction (assuming that AC17, TEMP, and XR have the octal values 17, 100, and 3, respectively), the statement

```
ADD AC17,@TEMP(XR)
```

generates the binary code

```
instruction          indirect                     memory
   code                 bit                        address

010 111 000    1 111     1     0 011     000 000 000 001 000 000

            accumulator        index
                              register
```

which appears in the program listing as

```
270 17 1 03 000100       ADD AC17,@TEMP(XR)
```

The mnemonic ADD has the octal code 270, and this is assembled into bits 0 to 8. The accumulator goes into bits 9 to 12. Since the @ appears with the memory address, bit 13 is set to 1. The index register goes into bits 14 to 17. Finally, the memory address is assembled into bits 18 to 35.

If any element is missing from a primary instruction, zeros are assembled in its instruction word field.

4.7.2 Mnemonics With Implicit Accumulators

A few mnemonics set bits in the accumulator field as well as in the instruction field. Therefore these mnemonics do not take accumulator operands, and are of the form

```
mnemonic address
```

These mnemonics and their octal codes are listed in Table C-5 in Appendix C.

For example, the mnemonic JFOV gives the octal code 25504; JFCL gives 255. Therefore both give the opcode 255 in bits 0 to 8, but JFOV also sets the accumulator bits (9 to 12) to binary 0001. This makes JFOV 100 equivalent to JFCL 1,100:

```
255 01 0 00 000100      JFOV 100
255 01 0 00 000100      JFCL 1,100
```

4.7.3 Input/Output Instructions

An input/output statement in your program resembles a primary instruction statement except that the first operand gives a device number instead of an accumulator. The general format is:

```
mnemonic device,address
```

In an input/output instruction, the indirect, index, and address fields (bits 13 to 35 inclusive) are assembled exactly as in a primary instruction.

Unlike a primary instruction word, however, an input/output word has a split instruction code in bits 0 to 2 (always set to 111 binary) and 10 to 12, and a device code in bits 3 to 9. The device code can be any expression or symbol giving a valid device code for your system.

(MACRO-defined I/O instruction mnemonics and device code mnemonics are listed in Tables C-2 and C-3 in Appendix C.)

For example (assuming that NVR has the octal value 1037), the statement

```
DATAI CDR,@NVR(4)
```

generates the binary code

```
              device            indirect                     memory
               code               bit                        address

     111    001 001 1    00 1    1    0 100    000 000 001 000 011 111

           instruction                index
               code                 register
```

which appears in the listing as

```
7 114 04 1 04 001037'    DATAI CDR,@NVR(4)
```

The octal code for the mnemonic DATAI is 70004, which is written in bits 0 to 14. The octal device code 114 (for card reader) is then overwritten in bits 3 to 9. The @ in the statement sets bit 13 to 1. The index register and memory address are placed in bits 14 to 17 and 18 to 35, as in a primary instruction.

4.7.4 Extended Instructions

The KL10 Extended Instruction Set is a multifunction instruction set that performs character-string editing, decimal-to-binary conversion, string move with left or right justification, string move with offset or translation, and string compare.

The Extended Instruction Set consists of a single KL10 instruction (EXTEND, octal 123) and a set of 16 extended operators. (See the Supplement to the Hardware Reference Manual.)

The KL10 EXTEND instruction mnemonics are listed in Table C-4 in Appendix C.

CHAPTER 5

USING MACROS

A macro is a sequence of statements defined and named in your program. When you call a macro (by invoking its name in your program), the sequence of statements from its definition is generated in line, replacing the call. A macro can have arguments.

By using macros with arguments, you can generate passages of code that are similar, but whose differences are controlled by the passed arguments. This saves repetition in building a source file.

5.1 DEFINING MACROS

Before you can call a macro, you must define it. You can also redefine a macro if you wish; the new definition simply replaces the old one.

To define (or redefine) a macro, use the pseudo-op DEFINE:

```
DEFINE macroname (darglist)<macrobody>
```

where macroname is the name of the macro, darglist is an optional list of dummy-arguments, and macrobody is a sequence of statements.

The macroname is a symbol; you must follow the rules for valid symbols in selecting a macroname. (See Section 2.4.1.)

The optional dummy-argument list can give one or more dummy-argument symbols through which values are passed to the sequence of statements. If a macro definition has dummy-arguments, they must be enclosed in parentheses. Use commas as delimiters between dummy-arguments. For each dummy-argument, leading and trailing spaces and tabs are ignored.

The macrobody is the sequence of statements you want to generate when you call the macro. The macrobody must be enclosed in angle brackets.

Here is an example of a macro definition:

```
DEFINE VMAG (WHERE,LENG) <
        ;Vector length routine
        MOVE 0,WHERE        ;Get first
                            ; component
        FMP 0               ;Square it
        MOVE 1,WHERE+1      ;Get second
                            ; component
        FMP 1,1             ;Square it
        FAD 1               ;Add square
                            ; of second
        MOVE1,WHERE+2       ;Get third
                            ; component
        FMP 1,1             ;Square it
        FAD 1               ;Add square
                            ; of third
        PUSHJ 17,FSQRT      ;Floating SQRT
                            ; routine
        MOVEM LENG          ;Store the
                            ; length
>
```

NOTE

Comments in a macro use storage. If you begin a comment with a double semicolon, the comment is listed in the definition but not stored for listing with expansions.

5.2 CALLING MACROS

You can call a macro by putting its name in your program. Recall that you must define the macro before you can call it. You can use the macroname as a label, an operator, or an operand.

If the macro's definition has dummy-arguments, the macro call can have arguments. The arguments passed to the macro are inserted into the defined sequence of statements as it is generated. The first passed argument replaces the first dummy-argument; the second passed argument replaces the second dummy-argument; this treatment continues for each argument passed. Any missing arguments are passed as nulls (zeros) or filled in by default arguments (see Section 5.5).

NOTE

If FOO is a macro with four dummy-arguments, the call FOO A,,C passes A and C as the first and third arguments. The second argument is passed as nulls; it is not considered missing and cannot be replaced by a default argument. The fourth argument is missing and will be replaced by a default argument if one has been defined; otherwise it is passed as nulls. (See Section 5.5.1.)

After argument substitution, the defined sequence of statements replaces the macroname and argument list in the source text. For example, suppose you have defined VMAG(A,B) as shown in Section 5.1 above, and VMAG appears in your program as

```
        LALL
        P7=245
        VLEN=11
        PLACE=15
TAG1:   MOVE 1,P7       ;Get P7
        MOVEM PLACE     ;Put it in PLACE
TAG2:   VMAG PLACE,VLEN
TAG3:   MOVE 1,VLEN     ;Get length
```

Then the code to be assembled is:

```
                                 LALL
        000245                   P7=245
        000011                   VLEN=11
        000015                   PLACE=15
200 01 0 00 000245       TAG1:   MOVE 1,P7         ;Get P7
202 00 0 00 000015               MOVEM PLACE       ;Put it in PLACE
                         TAG2:   VMAG PLACE,VLEN^
                                 ;Vector length routine
200 00 0 00 000015               MOVE 0,PLACE      ;Get first
                                                   ; component
160 00 0 00 000000               FMP 0             ;Square it
200 01 0 00 000016               MOVE 1,PLACE+1    ;Get second
                                                   ; component
160 01 0 00 000001               FMP 1,1           ;Square it
140 00 0 00 000001               FAD 1             ;Add square
                                                   ; of second
200 01 0 00 000017               MOVE 1,PLACE+2    ;Get third
                                                   ; component
160 01 0 00 000001               FMP 1,1           ;Square it
140 00 0 00 000001               FAD 1             ;Add square
                                                   ; of third
264 00 0 00 001007'              JSR FSQRT         ;Floating SQRT
                                                   ; routine
202 00 0 00 000011               MOVEM VLEN        ;Store
                                                   ; length
                         ^
200 01 0 00 000011       TAG3:   MOVE 1,VLEN       ;Get length
```

Notice that the macro definition has the dummy-arguments A and B in the macrobody. The call VMAG PLACE,VLEN causes PLACE to replace each appearance of A, and VLEN to replace each appearance of B.

NOTES

1. Under LALL, when the text of a macrobody is listed at call, it is enclosed in up-arrows (^).

2. Under XALL, the beginning of the text of a macrobody is marked by an up-arrow; the ending is marked by an up-arrow only if the last line of the macrobody generates binary code.

5.2.1 Macro Call Format

In a macro call, delimit the macroname with one or more blanks or tabs.

If the macro has arguments, the first nonblank, nontab character begins the argument list. Each argument ends with a comma, a carriage return, or a semicolon. These three characters cannot be used within arguments unless enclosed by special quoting characters. (See Section 5.2.2.)

Leading and trailing spaces and tabs are stripped from each argument unless they are within special quoting characters. Embedded spaces and tabs are not stripped.

You can continue an argument to the next line by using CTRL/underscore. Otherwise an unquoted carriage return or semicolon ends the argument and the argument list. An unquoted semicolon also begins the comment field.

5.2.2 Quoting Characters in Arguments

The special quoting characters for macro argument handling are:

< > angle brackets
( ) parentheses
[ ] square brackets
" " quote marks

NOTE

Single quote marks (apostrophes) are not special quoting characters.

Any character, including the semicolon (;), enclosed in special quoting characters is treated as a regular character. If one of the special quoting characters is to be passed as a regular character, it must be enclosed by different special quoting characters.

Here are the rules for macro argument handling. In the examples, FOO is assumed to be a defined macro:

1. The special quoting characters are not argument delimiters. They only tell the assembler to treat the enclosed characters as regular characters.

   FOO C<A,B> has one argument: C<A,B>.

   FOO C,D<A,B> has two arguments: C and D<A,B>.

2. With the two exceptions explained below, special quoting characters are always included in passed arguments.

   FOO A,(B,C) has two arguments: A and (B,C).

   FOO [XWD 1,L1]-1(AC) has one argument: [XWD 1,L1]-1(AC).

   FOO "(",0 has two arguments: "(" and 0.

   Exception 1: If the first character of the argument list is a left parenthesis, then it and its matching right parenthesis delimit the argument list. They are not treated as special quoting characters and are not included in passed arguments. All nested quoting characters except angle brackets are disabled. After stripping the outer parentheses, angle brackets are handled as described in Exception 2 below.

   FOO (A,B,C) has three arguments: A, B, and C.

   FOO (?LENGTH >132) has one argument: ?LENGTH >132.

   FOO ([A,B]) has two arguments: [A and B].

   FOO (<A,B>) has one argument: A,B.

   Exception 2: If a left angle bracket is the first character of the argument list, or the first character after an unquoted comma, then it and its matching right angle bracket are treated as special quoting characters, but are not included in passed arguments.

   FOO <A,B>,C has two arguments: A,B and C.

   FOO C,<A,B> has two arguments: C and A,B.

You can alter this argument handling by using the pseudo-op .DIRECTIVE with MACMPD, .ITABM, and .XTABM. (See Chapter 3.)

NOTE

To pass special characters in a macro call, we suggest defining the macro so that the delimiters are part of the passed argument. For example, use

```
DEFINE T1 (A) <OUTSTR [ASCIZ A]>
```

rather than

```
DEFINE T2 (A) <OUTSTR [ASCIZ \A\]>
```

The call T1 ">>" will work, but T2 ">>" will not.

5.2.3 Listing of Called Macros

You can control the listing of called macros by using the pseudo-ops XALL, SALL, and LALL. LALL causes macro expansions to be listed in full; XALL suppresses part of the listing; LALL suppresses all of the listing. The default among these three is XALL.

The following example shows the action of these pseudo-ops on macro listings:

```
                      DEFINE FOO (N)<
                      IFE N,<2>
                      IFN N,<1>
                      >
                      SALL
000000  000002        FOO(0)
000000  000001        FOO(1)
                      XALL
                      FOO(0)^
000000  000002        IFE 0,<2>
                      FOO(1)^
000000  000001        IFN 1,<1>
                      LALL
                      FOO(0)^
000000  000002        IFE 0,<2>
                      IFN 0,<1>
                      ^
                      FOO(1)^
                      IFE 1,<2>
000000  000001        IFN 1,<1>
                      ^
```

5.3 NESTING MACRO DEFINITIONS

You can nest macro definitions. That is, you can define a macro within the body of another macro definition. Notice, however, that the nested macro is not defined to the assembler until the nesting macro is called.

Here is an example:

```
DEFINE PERSON (A) <
        DEFINE CHILD (B) <
                DEFINE GRANDCHILD (C) <
                        EXP A,B,C>
        >
>
```

Until the DEFINE PERSON statement is assembled, calls to PERSON, CHILD, and GRANDCHILD are illegal. These macros are not yet defined to the assembler.

When the DEFINE PERSON statement is reached and assembled, PERSON can be called, but not CHILD or GRANDCHILD. The call PERSON 1 generates the text

```
PERSON 1^
        DEFINE CHILD (B) <
                DEFINE GRANDCHILD (C) <
                        EXP 1,B,C>
        >
^
```

thus defining CHILD to the assembler. The following call CHILD 2 generates the text

```
CHILD 2^
                DEFINE GRANDCHILD (C) <
                        EXP 1,2,C>
        ^
```

and GRANDCHILD is defined to the assembler. Finally, a call to GRANDCHILD 3 generates

```
                        GRANDCHILD 3^
000000  000001                                EXP 1,2,3^
000000  000002
000000  000003
```

Notice the result of a subsequent call to CHILD 10. The text

```
CHILD 10^
                DEFINE GRANDCHILD (C) <
                        EXP 1,10,C>
        ^
```

is generated, and this definition replaces the old definition of GRANDCHILD; the definitions of PERSON and CHILD are not changed. After this, the call GRANDCHILD 3 generates

```
                        GRANDCHILD 3^
000000  000001                                EXP 1,10,3^
000000  000010
000000  000003
```

NOTE

Using multiple angle brackets for a passed argument preserves the argument as one unit. For example passing the argument <<A,B,C>> to nested macros causes the outer macro to pass <A,B,C> as one argument; the first nested macro passes A, B, and C as three arguments.

5.4 CONCATENATING ARGUMENTS

The apostrophe (') is the concatenation operator for macro calls. If you insert an apostrophe immediately before or after a dummy-argument in the body of a macro, the assembler removes it at call. This removal joins (concatenates) the passed argument to the neighboring character in the generated text.

(One application of this concatenation is shown under COMMON ERRORS for the ASCIZ pseudo-op.)

If the apostrophe precedes the dummy-argument, the passed argument is suffixed to the preceding character; if the apostrophe follows the dummy-argument, the passed argument is prefixed to the following character.

You can use more than one apostrophe with a dummy-argument. In this case only apostrophes next to the dummy-argument will be removed (at most one from each side). Other apostrophes are treated as regular characters in the macrobody. The following example shows the treatment of apostrophes on both sides of the dummy-argument, and of double apostrophes.

```
DEFINE O (PREFIX,MIDFIX) <
        DEFINE OCOMP (SUFFIX) <
                PREFIX'O'MIDFIX''SUFFIX>
>
```

Now the call O A,J generates

```
DEFINE OCOMP (SUFFIX) <
        AOJ'SUFFIX>
```

because when the assembler replaces PREFIX with A, the apostrophe following is removed to form AO. When J replaces MIDFIX, the preceding apostrophe and first following apostrophe are removed to form AOJ'SUFFIX.

Now the call OCOMP LE generates

```
                              OCOMP LE^
343 00 0 00 000000                      AOJLE^
```

since the apostrophe is removed to join AOJ to LE.

5.5 DEFAULT ARGUMENTS AND CREATED SYMBOLS

Ordinarily, an argument missing from a macro call is passed as nulls. For example, the macro defined by

```
DEFINE WORDS (A,B,C) <
        EXP A,B,C>
```

when called by WORDS 1,1 generates three words containing 1, 1, and 0, respectively.

```
                              WORDS 1,1^
000000  000001                        EXP 1,1,^
000000  000001
000000  000000
```

You can, however, alter this handling by specifying default values other than nulls, or by using created symbols.

5.5.1 Specifying Default Values

If you want a missing argument to default to some value other than nulls, you can specify the default value in your DEFINE statement. Do this by inserting the default value in angle brackets immediately after the dummy-argument. For example, the macro defined by

```
DEFINE WORDS (A,B<222>,C<333>)<
        EXP A,B,C>
```

when called by WORDS 1,1 generates three words containing 1, 1, and 333, respectively.

```
                          WORDS 1,1^
000000  000001                    EXP 1,1,333^
000000  000001
000000  000333
```

NOTE

An argument passed as nulls by consecutive commas is not considered missing and cannot invoke a default value. Therefore missing arguments can occur only at the end of the list of passed arguments.

5.5.2 Created Symbols

A symbol used as a label in a macrobody must be different for each call of the macro (since duplicate labels are not allowed). Therefore for each call a different symbol for the label must be passed as an argument.

If you do not refer to such a label from outside the macro, you can simply let the assembler provide a new label for each call. This label is called a created symbol, and is of the form ..nnnn where nnnn is a 4-digit number.

To use a created symbol in place of a passed argument, use the percent sign (%) as the first character of the dummy-argument in your DEFINE statement. The assembler then creates a symbol for use in the macro expansion if that argument is missing from a call to the macro. If you provide an argument in the call, the passed argument overrides the created symbol.

NOTES

1. A null argument (indicated by two adjacent delimiters) is not treated as missing.

2. Avoid using symbols of the form ..nnnn, since they could interfere with created symbols.

The following example shows a macro defined with a created symbol, the macro called using the created symbol, and the macro called overriding the created symbol:

```
DEFINE COMPAR (TEST,SAVE,INDEX,%HERE) <
%HERE:   MOVE SAVE,TEST
         SETZ INDEX,
         CAME SAVE,TABLE(INDEX)
         JRST %HERE
>
COMPAR T1,T2,T3^
..0001:  MOVE T2,T1
         SETZ T3,
         CAME T2,TABLE(T3)
         JRST ..0001
COMPAR T1,T2,T4,HERE1^
HERE1:   MOVE T2,T1
         SETZ T4
         CAME T2,TABLE(T4)
         JRST HERE1
```

5.6 INDEFINITE REPETITION

The pseudo-ops IRP, IRPC, and STOPI give a convenient way to repeat all or part of a macro; you can change arguments on each repetition if you wish, and the number of repetitions can be computed at assembly time. You can use these three pseudo-ops only within the body of a macro definition.

To see how IRP works, assume the macro definition

```
DEFINE DOEACH (A) <
        IRP A,<A>>
```

The call DOEACH <ALPHA,BETA,GAMMA> produces the code

```
        000200          ALPHA=200
        000300          BETA=300
        000400          GAMMA=400
                        DOEACH <ALPHA,BETA, GAMMA>^
                                IRP
000000  000200          ALPHA
000000  000300          BETA
000000  000400          GAMMA
                        ^
```

because each subargument passed to IRP generates one repetition of the code. Notice that the range of IRP must be enclosed in angle brackets.

NOTE

Using angle brackets in the call to DOEACH is critical, since they make the string ALPHA,BETA,GAMMA a single argument for IRP. IRP then sees the commas as delimiting subarguments.

IRPC is similar to IRP, but an argument passed to IRPC generates one repetition for each character of the argument.

STOPI ends the action of IRP or IRPC after assembly of the current expansion. You can use STOPI with a conditional assembly to calculate a stopping point during assembly. For example:

```
                    ;Enter value of 111 for each radix from 2 to K

                    DEFINE CONV1 (L) <
                            RADIX L ;Set radix
                            111     ;Evaluate and enter
                            RADIX 8 ;Back to radix 8
                    >

                    DEFINE CONVERT (A) <
                            IRP A,<IFE K-A,<STOPI> ;Still OK?
                                    CONV1 A> ;CONV1
                    >

        000004      K=4
                    CONVERT <2,3,4,5,6,7,8,9>^
                            IRP
                    IFE K-2,<STOPI> ;Still OK?
                                    CONV1 2^
                            RADIX 2 ;Set radix
000000  000007              111     ;Evaluate and enter
                            RADIX 8 ;Back to radix 8
                    ^
                    IFE K-3,<STOPI> ;Still OK?
                                    CONV1 3^
                            RADIX 3 ;Set radix
000000  000015              111     ;Evaluate and enter
                            RADIX 8 ;Back to radix 8
                    ^
                    IFE K-4,<STOPI> ;Still OK?
                                    CONV1 4^
                            RADIX 4 ;Set radix
000000  000025              111     ;Evaluate and enter
                            RADIX 8 ;Back to radix 8
                    ^
                            ;CONV1
                    ^
```

5.7 ALTERNATE INTERPRETATIONS OF CHARACTERS PASSED TO MACROS

The normal argument passed by a macro call is simply the string of characters given with the call. MACRO offers three alternate interpretations of the passed argument.

If you prefix a backslash (\) to an expression argument, the argument passed is the ASCII numeric character string giving the value of the expression.

If you prefix a backslash-apostrophe (\') to an expression argument, the argument passed is the string whose value is the SIXBIT string with the integer value of the expression.

If you prefix a backslash-quotemark (\") to an expression argument, the argument passed is the string whose value is the ASCII string with the integer value of the expression.

To show how these work, the following example defines a macro to print the argument passed. Then four different arguments are passed using the various argument interpretations.

```
                      LALL
                      DEFINE LOOKIE (ARG) <
                      REMARK  The passed argument is:  ARG  >

                      LOOKIE 60^
                      REMARK  The passed argument is:  60  ^

                      LOOKIE \60^
                      REMARK  The passed argument is:  60  ^

                      LOOKIE \'60^
                      REMARK  The passed argument is:  P  ^

                      LOOKIE \"60^
                      REMARK  The passed argument is:  0  ^

        000060        Z=60

                      LOOKIE Z^
                      REMARK  The passed argument is:  Z  ^

                      LOOKIE \Z^
                      REMARK  The passed argument is:  60  ^

                      LOOKIE \'Z^
                      REMARK  The passed argument is:  P  ^

                      LOOKIE \"Z^
                      REMARK  The passed argument is:  0  ^

635170  425164        ZZ='SIXBIT'

                      LOOKIE ZZ^
                      REMARK  The passed argument is:  ZZ  ^

                      LOOKIE \ZZ^
                      REMARK  The passed argument is:  635170425164  ^

                      LOOKIE \'ZZ^
                      REMARK  The passed argument is:  SIXBIT  ^
```

```
203234  162311        ZZZ='ASCII'

                      LOOKIE ZZZ^
                      REMARK  The passed argument is:  ZZZ  ^

                      LOOKIE \ZZZ^
                      REMARK  The passed argument is:  203234162311  ^

                      LOOKIE \'ZZZ^
                      REMARK  The passed argument is:  ASCII  ^
```

CHAPTER 6

ASSEMBLER OUTPUT

MACRO can generate three kinds of output files:

1. A program listing (.LST) file
2. A binary program (.REL) file
3. A UNIVERSAL (.UNV) file

6.1 THE PROGRAM LISTING FILE

MACRO outputs the program listing file to the device you specify, usually your terminal or a disk file. You can control the form of the program listing by using the pseudo-ops .DIRECTIVE FLBLST, .DIRECTIVE SFCOND, LIST, XLIST, LALL, XALL and SALL. (See Chapter 3.) All MACRO programs begin with the implicit pseudo-ops LIST and XALL.

The listing has a heading at the top of each page and subpage. The first line gives the program name, the assembler version, the time and date of assembly, and the page number. The second line gives the program filename (including extension), the date and time of creation, and an optional program subtitle.

Example:

```
TIMER   MACRO %53(711)    10:07  27-APR-77       PAGE 2
TIMER   MAC        27-AUG-77       10:06          MACDEF
```

The listing has up to 55 lines per page. You can change this by using the L switch; /nnL specifies nn lines per page. A formfeed (CTRL/L) in your program begins a new page and increments the page number. If the linecount exceeds lines-per-page before a formfeed is found, a subpage number is formed. For example, the subpages following page 6 are 6-1, 6-2, and so forth. A formfeed would begin page 7.

The five columns in the program listing give:

1. The CREF line number (if the program was assembled with the CREF switch on).

2. The line sequence number (if the input file is sequenced).

3. The 6-digit octal address of the storage word, usually a sequential location assignment.

   ```
   400066'
   400067'
   400070'
   ```

   An apostrophe (') after the address shows that it is relocatable.

   For a PHASE pseudo-op, the phased address is given.

   For a BLOCK pseudo-op, only the address of the first word is given.

   For a program with PSECTS, the 2-digit PSECT number of the current PSECT immediately follows the address. For example,

   ```
   000100'02
   ```

   For a LOC or RELOC pseudo-op, only the address to which the location counter is set is given; the next word of code will be assembled at that address.

4. The assembled binary code (if any) in one of eight formats.

   Fullword: all zeros with number sign (000000000000#), showing that a fullword Polish fixup is required for the word of code.

   Halfword: two 18-bit bytes. Each halfword can be followed by an apostrophe (') to indicate that it is relocatable, or by a pound sign (#) to indicate that a Polish fixup is required for it. When you use the .HWFRMT pseudo-op, all code is listed in halfword format.

   Instruction: 9-bit op-code; 4-bit accumulator code; 1-bit indirect code; 4-bit index; 18-bit address.

   Input/output: 3-bit I/O code; 7-bit device code; 3-bit operand; 1-bit indirect code; 4-bit index; 18-bit address.

   Byte pointer: 6-bit byte position; 6-bit byte size; 1 unused bit; 1-bit indirect code; 4-bit index; 18-bit address.

   ASCII: five 7-bit bytes; one unused bit.

   SIXBIT: six 6-bit bytes.

BYTE: binary representation of specified bytes. Bytes appear on the program listing only to the extent that available horizontal space permits. For example, 36 1-bit bytes cannot be represented as individual bytes on the listing. Any halfword byte containing an address can be flagged by an apostrophe (') or by a pound sign (#). See the halfword format above.

OPDEF or assignment: one or two 18-bit bytes, as needed.

These examples show some code in each format:

```
000056'  000000000000#                B=A+C

000057' 000001  000017'               1,,TAG1          ;Halfword
000060' 000017  000001                AC17,,1          ;Halfword

000061' 255 01 0 00 000100            JFOV 100         ;Instruction
000062' 255 01 0 00 000100            JFCL 1,100       ;Instruction

000063' 7 114 04 1 04 001037'         DATAI CDR,@NVR(4)  ;I/O
000064' 7 110 20 1 05 000004          CONO CDP,@4(5)   ;I/O

000065' 21 06 0 00 000067'     P1:    POINT 6,B1,18    ;Byte pointer
000066' 44 10 0 00 000070'     P2:    POINT 8,B2       ;Byte pointer

000067' 07 00 01 000000        B1:    BYTE (6)7,0,1    ;Byte
000070' 006 004 002 000 00     B2:    BYTE (8)6,4,2,0  ;Byte

        017000  000000                OPDEF Z1[17B8]   ;OPDEF
        026000  000000                OPDEF Z2[26B8]   ;OPDEF

000071' 061 062 063 064 065           ASCII /12345/    ;ASCII
000072' 101 102 103 104 105           ASCII \ABCDE\    ;ASCII

000073' 21 22 23 24 25 26             SIXBIT /123456/  ;SIXBIT
000074' 41 42 43 44 45 46             SIXBIT \ABCDEF\  ;SIXBIT
```

An apostrophe (') shows the code as relocatable. The examples show relocatable values in the right half of some words. The left half can also be relocatable.

An asterisk (*) shows a symbol to be EXTERNAL or undefined.

A number sign (#) shows that a Polish expression is required to resolve the value.

5. Source statements and comments.

If the assembler finds errors in a line of text, it suffixes one or more letters to the sequence number as error codes. These error codes are discussed in Chapter 8. A code is not repeated for multiple errors of the same type in a line.

At the end of the listing, the assembler gives the total number of errors, followed by break addresses. The program break is the largest relocatable address assembled, plus 1. The absolute break is the largest absolute address assembled. The high-segment break is the largest high-segment address assembled. For a program with PSECTs, the break for each PSECT is also given.

The listing gives CPU time in the form mm:ss.sss where mm is minutes and ss.sss is seconds. Core used is given in K's; one K is 1024 words (2000 octal).

In the symbol table at the end of the listing, some symbols may have the following codes:

ent	result of ENTRY pseudo-op
ext	EXTERNAL symbol
int	INTERNAL symbol
pol	defined in terms of EXTERNAL symbols
sen	suppressed result of ENTRY pseudo-op
sex	suppressed EXTERNAL symbol
sin	suppressed INTERNAL symbol
spd	suppressed for debugger
udf	undefined symbol

If you use the /C switch with MACRO, you can generate three additional tables in the program listing. The /C switch directs MACRO to generate the listing file in a format suitable for input to CREF, the cross-referencing program. This is a .CRF file rather than the usual .LST file.

After assembly, the .CRF file can be used as input to CREF, and the output is the cross-referenced .LST file. This file contains the program listing and symbol table as described above. In addition, it has a cross-referenced symbol table, a table of macros and OPDEFs, and, if you use the /O switch with CREF, a cross-referenced table of opcodes and pseudo-ops.

The cross-referenced symbol table lists each user-defined symbol (except macros, OPDEFs, and SYN symbols), and lists the sequence number of each line containing the symbol.

The table of macros and OPDEFs shows each reference to macros, OPDEFs, and SYN symbols.

The opcode table shows each reference to MACRO-defined opcodes and pseudo-ops, giving the sequence number of each line containing the opcode or pseudo-op.

6.2 THE BINARY PROGRAM FILE

MACRO outputs the binary program file to the device you specify, usually a storage device. The default device is a disk. Most of the file is the binary expansion of your program instructions. These instructions are formatted into groups called REL Blocks; each block is labeled so that LINK can recognize it. Details of this formatting and labeling are discussed in the LINK Reference Manual.

A relocatable binary program file can be stored on any input/output device. The output format is not related to either block types or logical divisions of the device.

6.3 THE UNIVERSAL FILE

THE UNIVERSAL file is output only if the source file contains the UNIVERSAL pseudo-op. (See the discussions at UNIVERSAL in Chapter 3 and in Section 9.2.)

A UNIVERSAL file contains only symbols and definitions. These definitions are available to any program, and can be obtained by using the SEARCH pseudo-op.

CHAPTER 7

USING THE ASSEMBLER

To assemble a MACRO program, use one of the following:

1. The operating system command COMPILE. (See the Monitor Calls manual for details.)

2. The $MACRO card for the BATCH program. (See the GALAXY Batch Reference Manual.)

3. The MACRO command level.

To assemble a program in the command level of MACRO, type R MACRO to the system. The system then runs MACRO, which responds with an asterisk (*):

```
.R MACRO
*
```

Then define files for MACRO by typing a command of the form

```
relfile,listfile=sourcefile,...,sourcefile
```

where:

relfile is a filespec for the binary program output file.

listfile is a filespec for the program listing output file.

each sourcefile is a filespec for a source program input file; MACRO assembles source files in the order given.

The default device for each file is DSK:, but you can override this by prefixing devicecode: to any of the files. Default file extensions are .REL for relfile, .LST for listfile (.CRF if you use the /C switch), and .MAC for each sourcefile. You can override these by suffixing a file extension to any of the files.

You can specify a directory for any of these files by suffixing a project-programmer number (PPN) in square brackets.

You can set switches by suffixing /char or (char) to a file, where char is a switch code. Switch codes and their meanings are given in Table 7-1.

You can suppress the binary file by omitting its file specification (but keeping the comma):

```
,listfile=sourcefile,...,sourcefile
```

You can suppress the listing file by omitting its file specification and the comma:

```
relfile=sourcefile,...,sourcefile
```

You can suppress both output files by omitting their file specifications (but keeping the equal sign):

```
=sourcefile,...,sourcefile
```

You can access an indirect file (containing valid asterisk-level MACRO command strings) by typing a command of the form:

```
@indirectfile
```

where indirectfile is the file specification for the file.

Examples:

DATE,DATE=DATE	Assemble source file DATE.MAC from disk into binary program file DATE.REL on disk, and put the listing in file DATE.LST on disk.
DATE=DATE	No listing file.
,DATE=DATE	No binary file.
=DATE	No binary or listing file. Print all error diagnostics on the terminal.
DATE,TTY:=DATE	Send the listing to the terminal.
DATE,DATE=TTY:	Accept source code from the terminal.
DATE,DATE=TTY:,DSK:DATE	Accept source code from the terminal (usually symbol definitions), followed by more source code from the disk. Notice that DSK must be specified; otherwise, TTY would be assumed.

NOTE

Many programmers use the following commands to check assembly of short code sequences:

```
*,TTY:=TTY:
PASS2
```

This displays the assembled code line by line as you type it in.

Table 7-1
MACRO Switch Options

Switch	Meaning
/A	Advance magnetic tape reel by one file. The /A switch must immediately follow the device to which the switch refers.
/B	Backspace magnetic tape reel by one file. The /B switch must immediately follow the device to which the switch refers.
/C	Produce listing file in a format acceptable as input to CREF. Unless the filename is given, CREF.CRF is assumed; if no file extension is given, .CRF is assumed; if no listing device is specified, DSK: is assumed. The /C switch can be used only with the file specification for the program listing file; it must appear between the comma and the equal sign.
/E	List macro expansions (same as LALL pseudo-op).
/F	Output binary listing in multiformat (same as .MFRMT pseudo-op).
/G	Output binary listing in halfword format (same as .HWFRMT pseudo-op).
/H	Print HELP text (list of switches and explanations).
/L	Reinstate listing (same as LIST pseudo-op).
/M	List only the call and binary produced in a macro expansion (same as SALL pseudo-op).
/N	Suppress error printouts on the terminal.
/O	End literal with CR-LF or right square bracket (same as MLOFF pseudo-op).
/P	Increase the size of the pushdown list. This switch can appear as many times as desired. The pushdown list is initially set to a size of 80 (decimal) locations; each /P increases the size by 80 (decimal). /P must appear on the left of the =.
/Q	Suppress Q (questionable) warning errors on the listing. /Q must appear on the left of the =.

(Continued on next page)

Table 7-1 (Cont.)
MACRO Switch Options

Switch	Meaning
/S	Suppress listing (same as XLIST pseudo-op).
/T	Skip to the logical end of the magnetic tape. The /T switch must immediately follow the device to which the switch refers.
/U	Do not generate a .UNV file on DSK. The /U switch must appear immediately after the specification for the binary program file; that is, it must appear between the file specification and the comma.
/W	Rewind the magnetic tape. The /W switch must immediately follow the device to which the switch refers.
/X	Suppress listing of macro expansions (same as XALL).

CHAPTER 8

ERRORS AND MESSAGES

MACRO has three kinds of messages:

1. Informational messages

2. Single-character error codes

3. MCRxxx messages (where xxx is a 3-letter mnemonic code)

8.1 INFORMATIONAL MESSAGES

MACRO's informational messages are printed at the foot of the program listing. These messages and their explanations are given in Table 8-1.

Table 8-1
MACRO Informational Messages

Message	Explanation
ABSLUTE BREAK	The highest absolute address over 137.
CORE USED	The size of the low segment used to assemble the source program.
CPU TIME USED	The CPU time for assembly in minutes and seconds.
ERRORS DETECTED	The number of errors detected by MACRO during assembly (errors marked on the listing by single-character codes other than Q).
HI-SEG. BREAK	The length of the high segment.
PROGRAM BREAK	The length of the low segment.
PSECT n BREAK	The length of PSECT n.
UNASSIGNED DEFINED AS IF EXTERNAL	Undefined symbol; treated as EXTERNAL.
WARNINGS GIVEN	The number of Q errors found. Processing is terminated if under BATCH.

8.2 SINGLE-CHARACTER ERROR CODES

Single-character error codes are printed in the program listing near the left margin of the line where the error occurs. If more than one kind of error occurs in the same line, more than one character will be printed; if more than one error of the same kind occurs in the line, the code is printed only once.

Codes for M, P, V, and X errors are typed during Pass 1.

If you use CREF to produce a cross-referenced listing file, all the single-character error codes will appear in the cross-reference table as %....x, where x is the code character.

Table 8-2 gives the single-character error codes and their explanations.

Table 8-2
MACRO Single-Character Error Codes

Code	Explanation
A	Argument error in pseudo-op. This is a broad class of errors that can be caused by an improper argument in a pseudo-op. The A errors include: 1. Symbol used is improperly formed. 2. IFIDN comparison string is too large. 3. OPDEF of macro or SYN. 4. Invalid SIXBIT character. 5. Byte size in BYTE more than 36. 6. RADIX50 code not absolute. 7. End of line of IF reached before < character seen. 8. Assignment made in an address field; for example, MOVEI A=10. (However, MOVEI <A=10> is valid.) 9. Assignment of a label; for example, TAG: TAG=1. 10. Missing symbol in SYN. 11. Unknown symbol in SYN. 12. Missing right parenthesis in an index. 13. Missing left parenthesis in a BYTE statement. 14. No comma after repeat count. 15. IRP or IRPC not in a macro. 16. Argument for IRP or IRPC is not a dummy symbol; for example, DEFINE GO (A) IRP B. 17. IRP or IRPC argument is a created symbol. 18. STOPI not in IRP or IRPC.
D	Multiply defined symbol. The statement contains a tag that refers to a multiply defined symbol. The first definition is used for assembling the statement.

(Continued on next page)

Table 8-2 (Cont.)
MACRO Single-Character Error Codes

Code	Explanation
E	Improper use of an EXTERNAL symbol. The E errors include: 1. Symbol both EXTERNAL and internal. 2. EXTERNAL symbol used as accumulator register address. 3. EXTERNAL symbol used with IF. 4. EXTERNAL symbol used as address for LOC, RELOC, PHASE, HISEG, or TWOSEG. 5. EXTERNAL symbol used for array name or size in ARRAY. 6. EXTERNAL symbol used as REPEAT count.
L	Literal generates less than 1 or more than 99 words of data.
M	Symbol defined more than once; retains its first definition. If a symbol is first defined as a variable and later as a label, it retains the label definition. This error can be caused by multiple appearances of TITLE, or TITLE with UNIVERSAL.
N	Number error. The N errors include: 1. Number exceeds the permitted range. 2. B shift not absolute. 3. Digits exceed current radix. If radix is 8, the single character 9 is acceptable but the number 19 is not acceptable. 4. Character after up-arrow not B, O, F, L, D, !, or -. 5. Illegal expression after E.
O	Operation code undefined. It is assembled as zeros.
P	Phase error. In general, the assembler generates the same number of program locations in Pass 1 and Pass 2. Any discrepancy causes a phase error. Phase errors can be caused by incorrect literal allocation. If a symbol is used as a macro to generate code in Pass 1, and is used as a label in Pass 2, a phase error can occur. A relocatable label that is defined in a literal and then used in an arbitrary expression; MACRO generates a Polish expression instead of treating the label as EXTERNAL.

(Continued on next page)

Table 8-2 (Cont.)
MACRO Single-Character Error Codes

Code	Explanation
Q	Questionable. This is a broad class of warnings in which the assembler finds ambiguous language. Statements causing Q errors may not generate correct code. The Q errors include: 1. Too many ASCII characters in double quotes ("). Only the first five are used. 2. Too many SIXBIT characters. Only the first six are used. 3. Value too large; high-order bits are lost. 4. Illegal expression after E. 5. Illegal control character. 6. Comma detected after all required fields filled; for example, MOVE 1,2,. 7. HISEG or TWOSEG found after relocatable code assembled. 8. Instruction memory address operand does not have either all 0's or all 1's in its left half; for example, 1,,0 or -4,,-1. 9. More than 18-bit values used in XWD.
R	Relocation error. The R errors include: 1. Expression neither absolute nor relocatable. 2. LOC or RELOC used improperly. 3. Relocatable BLOCK size given. 4. Relocatable accumulator address given.
S	PSECT usage error. The S errors include: 1. More than 64 distinct PSECT names used. 2. More than 16 levels of PSECT nesting used. 3. PSECT name given with .ENDPS is not the name of the current PSECT.
U	Undefined symbol.
V	Symbol used to control the assembler is undefined. Make the definition precede the reference.
X	Error in defining or calling a macro during Pass 1.

8.3 MCRxxx MESSAGES

The MCRxxx messages are issued to the terminal during assembly. (The xxx represents a 3-letter code.)

Any MCRxxx message that is preceded by a question mark is normally fatal under batch processing. A few MCRxxx messages are informational; these are issued within square brackets.

Table 8-3 gives all the MCRxxx messages. Each 3-letter code and its message are printed in **boldface** type. For some messages, an explanation is printed in lightface type.

Table 8-3
MCRxxx Messages

Code	Message and Explanation
ATS	LINES/PAGE ARGUMENT TOO SMALL The argument given must be greater than three to allow space for the page heading.
CAP	CORE ALLOCATION PROBLEM WITH MEMORY-RESIDENT UNIVERSALS UNIVERSAL programs assembled with the /U switch must have the same output specifications as succeeding files. (See the pseudo-op UNIVERSAL in Chapter 3.) However, if none of the memory-resident UNIVERSALS are to be searched by subsequent files in the command sequence, you can clear the UNIVERSALs and force the needed memory allocation by typing CTRL/C, followed by START.
CFU	CANNOT FIND UNIVERSAL Correct the request for the UNIVERSAL file, or assemble the required UNIVERSAL file.
CME	COMMAND ERROR The last command string contains an error.
CTL	COMMAND LINE TOO LONG The last input command string contains more than 200 characters.
ECF	ERROR READING COMMAND FILE This is a file status error.
EP1	END OF PASS 1 Manual input is required to begin Pass 2 because input is from cards or terminal.
EPP	EXPRESSION PARSING PROBLEM An expression was misinterpreted because MACRO interpreted a slash as a division operator, or a hyphen as a subtraction operator.
ERU	UNEXPECTED END-OF-FILE READING UNIVERSAL FILE
EWE	ERROR WHILE EXPANDING MACRO has an internal error in expanding a macro. Rewrite the macro, and submit a Software Performance Report.

(Continued on next page)

Table 8-3 (Cont.)
MCRxxx Messages

Code	Message and Explanation
IBL	INPUT BLOCK TOO LARGE DEVICE An input block from the specified device is too large.
ICP	INPUT CHECKSUM OR PARITY ERROR DEVICE This is a hard-data error.
IDE	INPUT DATA ERROR DEVICE This is a hard-data error.
ISC	ILLEGAL SYNTAX IN CONDITIONAL OR REPEAT
ISD	ILLEGAL SYNTAX IN MACRO DEFINITION The macro is improperly defined.
ISI	ILLEGAL SYNTAX IN [IRP or IRPC] INSIDE MACRO
ISR	ILLEGAL SYNTAX IN REPEAT
LNF	LOAD THE NEXT FILE The command string specifies the next file device as card reader or terminal. Input the file through the appropriate device.
LRE	(?) LOOKUP, RENAME, OR ENTER ERROR
LRE	(0) FILE WAS NOT FOUND
LRE	(1) NO DIRECTORY FOR PROJECT-PROGRAMMER NUMBER
LRE	(2) PROTECTION FAILURE
LRE	(3) FILE WAS BEING MODIFIED
LRE	(4) RENAME FILE NAME ALREADY EXISTS
LRE	(5) ILLEGAL SEQUENCE OF UUO'S
LRE	(6) BAD UFD OR BAD RIB
LRE	(7) NOT A SAV FILE
LRE	(10) NOT ENOUGH CORE
LRE	(11) DEVICE NOT AVAILABLE
LRE	(12) NO SUCH DEVICE
LRE	(13) NO TWO RELOC REG. CAPABILITY

(Continued on next page)

Table 8-3 (Cont.)
MCRxxx Messages

Code	Message and Explanation
LRE	(14) NO ROOM OR QUOTA EXCEEDED
LRE	(15) WRITE LOCK ERROR
LRE	(16) NOT ENOUGH MONITOR TABLE SPACE
LRE	(17) PARTIAL ALLOCATION ONLY
LRE	(20) BLOCK NOT FREE ON ALLOCATION
LRE	(21) CAN'T SUPERSEDE (ENTER) AN EXISTING DIRECTORY
LRE	(22) CAN'T DELETE (RENAME) A NON-EMPTY DIRECTORY
LRE	(23) SFD NOT FOUND
LRE	(24) SEARCH LIST EMPTY
LRE	(25) SFD NESTED TOO DEEPLY
LRE	(26) NO-CREATE ON FOR SPECIFIED PATH
LTL	LITERAL TOO LONG
MDE	MONITOR DETECTED SOFTWARE INPUT ERROR DEVICE The input file is not in a valid mode.
MPA	MISSING CLOSE PAREN AROUND ARG LIST
NEC	INSUFFICIENT CORE Not enough memory is available to assemble the program.
NES	NO END STATEMENT ENCOUNTERED ON INPUT FILE
NUF	NOT A REAL UNIVERSAL FILE No such UNIVERSAL file was found.
OBL	OUTPUT BLOCK TOO LARGE DEVICE This is a file-status error.
OCP	OUTPUT CHECKSUM OR PARITY ERROR DEVICE This is a hard-data error.
ODE	OUTPUT DATA ERROR DEVICE This is a hard-data error.
OQE	OUTPUT QUOTA EXCEEDED ON DEVICE
OUF	UNIVERSAL FILE DEFAULT ARGUMENTS LOST, REASSEMBLE

(Continued on next page)

Table 8-3 (Cont.)
MCRxxx Messages

Code	Message and Explanation
PDL	PDP OVERFLOW, TRY /P See the /P switch in Table 7-1.
PET	INPUT PHYSICAL END OF TAPE DEVICE
PGE	PRGEND ERROR See the PRGEND pseudo-op for proper use of PRGEND.
PTC	POLISH TOO COMPLEX A Polish expression is too complex for MACRO to handle. Restructure or split the expression.
SOC	STATEMENT OUT OF ORDER .COMMON The .COMMON pseudo-op must precede all statements that generate code, and all references to the COMMON block.
STO	SEARCH TABLE OVERFLOW, CANNOT SEARCH UNIVERSAL
TMU	TOO MANY UNIVERSALS Too many UNIVERSAL files are being searched. The number permitted is an assembly parameter; it can be increased by reassembling MACRO.
UVS	UNIVERSAL VERSION SKEW, REASSEMBLE UNIVERSAL The UNIVERSAL file was assembled with a later version of MACRO than you are using now. Reassemble the UNIVERSAL file.
UWU	UNABLE TO WRITE UNIVERSAL FILE
WLE	OUTPUT WRITE-LOCK ERROR DEVICE

CHAPTER 9

PROGRAMMING CONSIDERATIONS

The previous chapters of this manual define the MACRO language elements. In particular, the pseudo-op definitions in Chapter 3 define many of MACRO's most important features. However, the usefulness of some pseudo-ops can be seen only in the context of a "family" of pseudo-ops.

In this chapter, we discuss three such families of pseudo-ops. The programming features concerned are:

1. Program segmentation
2. UNIVERSAL files
3. Conditional assembly

9.1 PROGRAM SEGMENTATION

MACRO's relocation counters can accommodate three types of programs:

1. A single-segment program uses only one relocation counter.
2. A two-segment program also uses one relocation counter, and is characterized by its use of the TWOSEG pseudo-op.
3. A program with PSECTS can use many relocation counters, and is characterized by its use of the .PSECT and .ENDPS pseudo-ops.

9.1.1 Single-Segment Programs

A single-segment program uses only one relocation counter. This counter can be used to assign any address from 0 to 777777. The initial setting of the counter is 0.

As MACRO assembles your program, it places code and data at the address given by the current value of the relocation counter, incrementing the counter's value for each word assembled.

For example, a statement can require assembly of one word of code, incrementing the relocation counter by 1. Another statement can require assembly of five words of code, incrementing the relocation counter by 5. Still another statement may not generate code, leaving the relocation counter unchanged.

You can reset the value of the relocation counter by using the pseudo-op RELOC with an argument. For example, using RELOC A sets the value of the relocation counter to the value of A.

In the following example, 100 words are allocated for a table, incrementing the relocation counter by 100. Then the table length is calculated as TABLEN. A RELOC TABLE returns to the top of the table, where the first three words are initialized. Finally a RELOC TABLE+TABLEN sets the relocation to the foot of the table to continue assembly.

```
000000'                   TABLE:  BLOCK 100          ;Allocate table
               000100             TABLEN=.-TABLE     ;Table length
000000'                           RELOC TABLE        ;Top of table
000000' 000000 000001             EXP 1,2,3          ;Init first 3
000001' 000000 000002
000002' 000000 000003
000100'                           RELOC TABLE+TABLEN ;Continue
```

9.1.2 Two-Segment Programs

By using the TWOSEG pseudo-op, you can divide your program into a high segment and a low segment. This pseudo-op must precede any statement in your program that generates code.

The TWOSEG pseudo-op tells MACRO that there will be two segments, and MACRO generates a REL Block Type 3, which tells LINK to expect two segments for loading.

You can use TWOSEG either with or without an address argument. There are important differences between the two:

1. TWOSEG without an argument specifies that the high segment begins at the address 400000. The initial value of the relocation counter is at the address 0 in the low segment.

2. TWOSEG with an argument specifies that the high segment begins at the given address, and further specifies that the initial value of the relocation counter is that address. (The given address is reduced to the next lower multiple of 2000 octal; if this result is 0, MACRO treats the TWOSEG as if no argument were given.)

The high-segment starting address divides all code into two segments. MACRO and LINK consider all code at addresses above the high-segment address to be in the high segment, and all other code to be in the low segment.

MACRO always remembers the value the relocation counter had before the last RELOC found. (This stored value is initially 0.)

Therefore in a two-segment program, you can begin in one segment, and then RELOC to the other. From then on, you can switch segments simply by using RELOC with no argument. MACRO will begin assigning addresses at the first unused location in the opposite segment.

For example,

```
400000'                          TWOSEG
000000'  000000   000001         EXP 1,2           ;Lo-seg
000001'  000000   000002
400000'                          RELOC 400000      ;Hi-seg
400000'  000000   000003         EXP 3,4
400001'  000000   000004
000002'                          RELOC             ;Lo-seg
000002'  000000   000005         EXP 5,6
000003'  000000   000006
400002'                          RELOC             ;Hi-seg
400002'  000000   000007         EXP 7,10
400003'  000000   000010
```

9.1.3 Programs With PSECTs

You can construct a program having up to 64 segments by using the .PSECT and .ENDPS pseudo-ops. These pseudo-ops control switching among program segments (PSECTs).

Each PSECT has its own relocation counter; each is separately relocated at load time. Therefore a program with two PSECTs is different from a two-segment program in that the PSECTed program has two relocation counters, while the two-segment program has only one.

The pseudo-op .PSECT specifies that code should be assembled for a given PSECT. For example, .PSECT A specifies that code is to be assembled in the program segment (PSECT) called A. The pseudo-op .ENDPS ends assembly in the current PSECT.

PSECTs can be nested up to 16 levels. In a nested PSECT, the .ENDPS pseudo-op begins assembly in the next outer PSECT; in an unnested PSECT, .ENDPS begins assembly in the blank PSECT. (You can think of the blank PSECT as being outside of all your explicitly declared PSECTs.)

Here is an example showing three PSECTs (A, B, and C):

```
000000'00  000000  000001    EXP 1,2          ;Blank PSECT
000001'00  000000  000002
000000'01                    .PSECT A         ;1st PSECT
000000'01  000000  000003    EXP 3,4
000001'01  000000  000004
000000'02                    .PSECT B         ;2nd PSECT (nested)
000000'02  000000  000005    EXP 5,6
000001'02  000000  000006
000002'01                    .ENDPS B         ;1st PSECT
000002'01  000000  000007    EXP 7,10
000003'01  000000  000010
000002'00                    .ENDPS A         ;Blank PSECT
000002'00  000000  000011    EXP 11,12
000003'00  000000  000012
000000'03                    .PSECT C         ;3rd PSECT
000000'03  000000  000013    EXP 13,14
000001'03  000000  000014
000004'00                    .ENDPS C         ;Blank PSECT
000002'02                    .PSECT B         ;2nd PSECT
000002'02  000000  000015    EXP 15,16
000003'02  000000  000016
000004'00                    .ENDPS B         ;Blank PSECT
```

In the example, the blank PSECT surrounds everything. Embedded in the blank PSECT are:

1. PSECT A (which also nests some of PSECT B)

2. PSECT C

3. Another segment of PSECT B

Each PSECT used in a program generates the PSECT name as a global symbol. At load time, this symbol will take the value of the origin specified for the PSECT.

When LINK loads your program, all the parts of the same PSECT are loaded together. These parts can be in more than one program, or in more than one file. For details of LINK's handling of PSECTs at load time, see the LINK Reference Manual.

9.2 UNIVERSAL FILES

A UNIVERSAL file contains direct-assignment symbol definitions. The symbols defined can have any attributes.

A UNIVERSAL file is convenient because it can contain definitions that you want for many programs. Those programs can then obtain the definitions by your use of the SEARCH pseudo-op. This searching adds to the assembly only those definitions that are needed; other definitions in the UNIVERSAL file are not used.

To build a UNIVERSAL file from a MACRO source file, insert the pseudo-op

UNIVERSAL filespec

where the filespec gives the file for output of the UNIVERSAL file. This file will contain all the symbols and definitions given in the program.

Another program can obtain these definitions if it contains the SEARCH pseudo-op:

SEARCH filespec

where filespec names the UNIVERSAL file. At the end of Pass 1 assembly, MACRO will search the UNIVERSAL file for any undefined symbols. If a definition is found in the UNIVERSAL file, MACRO moves it into the symbol tables of the current program.

For example, a UNIVERSAL file can contain definitions for register mnemonics:

```
                        UNIVERSAL REGS

        000000          R0=0
        000001          R1=1
        000002          R2=2
        000003          R3=3
        000004          T1=4
        000005          T2=5
        000016          SP=16
        000017          P=17
                        END
```

Then another assembly can obtain these by using the SEARCH REGS pseudo-op:

```
                                  SEARCH REGS

000000'   000 00 0 00   000000          Z R0,
000001'   000 01 0 00   000000          Z R1,
000002'   000 02 0 00   000000          Z R2,
000003'   000 03 0 00   000000          Z R3,
000004'   000 04 0 00   000000          Z T1,
000005'   000 05 0 00   000000          Z T2,
000006'   000 16 0 00   000000          Z SP,
000007'   000 17 0 00   000000          Z P,
```

A UNIVERSAL file can contain definitions for any user-defined symbols. You may find it convenient to build UNIVERSAL files containing macros, OPDEFs, and direct-assignment symbols that you use often in your programs.

An example of a UNIVERSAL program appears in the program examples in Appendix D.

9.3 CONDITIONAL ASSEMBLY

Using conditional assembly in your programs can make programming easier, and can make your assembled programs shorter. The pseudo-ops used for conditional assembly are IRP, IRPC, STOPI, .IF, .IFN, and the IFx group. IRP, IRPC, and STOPI are discussed fully in Chapter 3 and Section 5.6.

We will confine the discussion here to a few classic uses of the remaining conditional assembly pseudo-ops.

The first of these is the use of IFNDEF to establish default switch settings for a program. The example here is from the MACRO program itself, and concerns assembly of F40-switch-dependent symbols.

Near the beginning of the code, MACRO has the statement:

```
IFNDEF F40 <F40==0>
```

This statement has effect only if the symbol F40 is not defined, in which case the statement F40==0 is assembled. This sets the F40 switch to "off."

But if a file defining F40 is assembled with (and before) the MACRO source file, then the statement F40==0 is not assembled, leaving the "outside" definition in force.

Therefore the statement IFNDEF F40 <F40==0> serves as a default definition for F40, and this default is used only if no other definition overrides it.

Another application of conditional assembly is connected with the symbol F40. In MACRO's program segments on symbol searching, some symbols will be defined (and therefore found in the search) only if the F40 switch is "on."

Here is how MACRO's code handles these symbols. There is a code sequence as follows:

```
;MACRO TO HANDLE F40 UUOS
   IFE F40,<
DEFINE XF (SB,CD) <>>          ;NULL MACRO
   IFN F40,<SYN X,XF>          ;USUAL X MACRO
```

The "usual X macro" is merely a macro to set up symbols to be defined and the code to assemble on finding them. The macro XF will be used to handle definitions for F40 UUOs.

Now if the F40 switch is on, the macro XF is made synonymous with the macro X, and the F40 UUOs are defined in the same way as other operators. But if the F40 switch is off, XF is made a null macro so that all the F40 UUOs are ignored during assembly and are not defined to MACRO.

The assembly of the F40 UUOs depends on the value of the F40 switch, and the value of the switch depends on its definition. If MACRO had no IFNDEF F40 statement, an "outside" file would have to define the switch at every assembly of MACRO. But the default definition allows assembly of MACRO alone, and the outside file is needed only to turn the switch on.

Examples of conditional assembly are shown in the program examples in Appendix D.

APPENDIX A

MACRO CHARACTER SETS

Table A-1 gives the 101 ASCII characters allowed in MACRO and their octal ASCII codes; the 64 SIXBIT characters and their octal SIXBIT codes; and the 40 RADIX50 characters and their octal RADIX50 codes.

Table A-1
MACRO Character Sets

Character	ASCII Code	SIXBIT Code	RADIX50 Code
(horizontal tab)	011		
(linefeed)	012		
(vertical tab)	013		
(formfeed)	014		
(carriage-return)	015		
(CTRL/Z)	032		
(CTRL/_)	037		
(blank)	040	00	00
!	041	01	
"	042	02	
#	043	03	
$	044	04	46
%	045	05	47
&	046	06	
'	047	07	
(	050	10	
)	051	11	
*	052	12	
+	053	13	
,	054	14	
-	055	15	
.	056	16	45
/	057	17	

(Continued on next page)

Table A-1 (Cont.)
MACRO Character Sets

Character	ASCII Code	SIXBIT Code	RADIX50 Code
0	060	20	01
1	061	21	02
2	062	22	03
3	063	23	04
4	064	24	05
5	065	25	06
6	066	26	07
7	067	27	10
8	070	30	11
9	071	31	12
:	072	32	
;	073	33	
<	074	34	
=	075	35	
>	076	36	
?	077	37	
@	100	40	
A	101	41	13
B	102	42	14
C	103	43	15
D	104	44	16
E	105	45	17
F	106	46	20
G	107	47	21
H	110	50	22
I	111	51	23
J	112	52	24
K	113	53	25
L	114	54	26
M	115	55	27
N	116	56	30
O	117	57	31
P	120	60	32
Q	121	61	33
R	122	62	34
S	123	63	35
T	124	64	36
U	125	65	37
V	126	66	40
W	127	67	41

(Continued on next page)

Table A-1 (Cont.)
MACRO Character Sets

Character	ASCII Code	SIXBIT Code	RADIX50 Code
X	130	70	42
Y	131	71	43
Z	132	72	44
[	133	73	
\	134	74	
]	135	75	
^	136	76	
_	137	77	
a	141		
b	142		
c	143		
d	144		
e	145		
f	146		
g	147		
h	150		
i	151		
j	152		
k	153		
l	154		
m	155		
n	156		
o	157		
p	160		
q	161		
r	162		
s	163		
t	164		
u	165		
v	166		
w	167		
x	170		
y	171		
z	172		

APPENDIX B

MACRO SPECIAL CHARACTERS

Characters and combinations having special interpretations in MACRO are given in Table B-1. These interpretations apply only in the contexts described. In particular, they do not apply within text strings or comment fields.

For each usage of special characters, a cross-reference to a text discussion is given in the rightmost column of the table. For references to pseudo-ops, only the pseudo-op name is given; all pseudo-ops are discussed in alphabetical order in Chapter 3.

Table B-1
Interpretations of Special Characters

Characters	Context	Form	Interpretation	Discussed in Section
B	between two integer expressions	mBn	causes the binary representation of m to be placed with rightmost bit at bit n (decimal).	2.2.6
^B	before integer expression	^Bn	shows that n is a binary number.	2.2.2
^D	before integer expression	^Dn	shows that n is a decimal number.	2.2.2
E	between floating-point decimal number and signed decimal integer	fE+n	multiplies f by the +nth power of 10.	2.2.5
^F	before integer expression	^Fn	shows that n is a fixed-point decimal number.	2.2.4
G	after integer	nG	suffixes nine zeros to n.	2.2.3
K	after integer	nK	suffixes three zeros to n.	2.2.3
^L	before decimal integer expression	^Ln	generates the number of leading zeros in the binary representation of n.	2.2.8
M	after integer	nM	suffixes six zeros to n.	2.2.3

(Continued on next page)

Table B-1 (Cont.)
Interpretations of Special Characters

Characters	Context	Form	Interpretation	Discussed in Section
^O	before integer expression	^On	shows that n is an octal number.	2.2.2
:	after symbol	sym:	shows that sym is a label.	2.4.2.1, 4.1, 4.5
::	after symbol	sym::	shows that sym is a global INTERNAL label.	2.4.2.1, 4.1, 4.5
:!	after symbol	sym:!	shows that sym is a label, but not to be output by debugger.	2.4.2.1, 4.1, 4.5
::!	after symbol	sym::!	shows that sym is a global INTERNAL label, but not to be output by debugger.	2.4.2.1, 4.1, 4.5
;	before end of line	;text	shows that text is a comment.	4.4, 4.5
;;	before end of line (usually in a macro)	;;text	shows that text is a comment to be printed in the macro definition but not at call.	4.4, 4.5
.	as expression		generates current value of the location counter.	2.3, 4.6
.	embedded in numerals	int.fr	shows that int.fr is a floating-point decimal number.	2.2.5

(Continued on next page)

Table B-1 (Cont.)
Interpretations of Special Characters

Characters	Context	Form	Interpretation	Discussed in Section
,	among numbers and symbols	,	delimits operands, accumulator, arguments.	4.3, 4.5 5.1, 5.2
,,	among numbers and symbols	,,	delimits a null macro argument.	5.2, 5.5
,,	between two expressions	lhw,,rhw	delimits left halfword (lhw) from right halfword (rhw).	2.5.4.1
!	between two expressions	A!B	generates the logical inclusive OR of A and B.	2.5.2
^!	between two expressions	A^!B	generates the logical exclusive OR of A and B.	2.5.2
&	between two expressions	A&B	generates the logical AND of A and B.	2.5.2
^-	before expression	^-A	generates one's complement of value of A (logical NOT).	2.5.2
*	between two expressions	A*B	generates product of A and B.	2.5.1
/	between two expressions	A/B	generates quotient of A by B.	2.5.1
+	between two expressions	A+B	generates sum of A and B.	2.5.1

(Continued on next page)

Table B-1 (Cont.)
Interpretations of Special Characters

Characters	Context	Form	Interpretation	Discussed in Section
-	between two expressions	A-B	generates difference of A and B.	2.5.1
-	before an expression	-A	generates the two's complement of the value of A.	2.2.1, 2.2.4, 2.2.5
"..."	around text	"text"	shows that text is a 7-bit ASCII string, to be right justified in field of five characters.	ASCII, ASCIZ
'...'	around text	'text'	shows that text is a SIXBIT string, to be right justified in field of six characters.	SIXBIT
'	adjoining dummy argument in macro body	text'darg or darg'text	concatenates passed argument to text at call to macro.	5.4
#	after symbol	sym#	shows that sym is a variable symbol, whose address is usually at the end of the binary program.	2.4.3
##	after symbol	sym##	shows that sym is a global EXTERNAL symbol.	2.4.5.2
\	prefixed to expression in macro call	\expr	directs that the argument passed be the string for the ASCII value of expr in the current radix.	5.7.1

(Continued on next page)

Table B-1 (Cont.)
Interpretations of Special Characters

Characters	Context	Form	Interpretation	Discussed in Section
\'	prefixed to expression in macro call	\'expr	directs that the argument passed be the string whose SIXBIT code is the value of expr.	5.7.3
\"	prefixed to expression in macro call	\"expr	directs that the argument passed be the string whose ASCII code is the value of expr.	5.7.3
CTRL/_ (CONTROL-underscore)	before CR-LF	CTRL/_	continues argument to next line; does not operate across end-of-macro.	5.2.1
_	between two expressions	A_B	shifts the binary representation of A to the left B positions. (If B is negative, shift is to right.)	2.2.6
@	prefixed to address	@address	sets bit 13 of the instruction word, indicating indirect addressing.	4.7.1
%	1st character of dummy argument in macro definition	%darg	directs that %darg be replaced by a created symbol at macro call; MACRO will substitute a different symbol for it on each use of the macro.	5.5.2
( )		(...)	encloses index field; encloses dummy arguments in macro definition; quotes characters for macro argument handling; swaps the two halves of enclosed value.	4.7.1, 5.1 5.2.2

(Continued on next page)

MACRO SPECIAL CHARACTERS

Table B-1 (Cont.)
Interpretations of Special Characters

Characters	Context	Form	Interpretation	Discussed in Section
< >		<...>	nests expressions; encloses conditional assembly code; encloses code in REPEAT, IRP, and IRPC pseudo-ops; encloses macrobody in DEFINE pseudo-op; quotes characters for macro argument handling; forces evaluation of symbol.	2.5.4 IFx, .IF, .IFN, REPEAT, IRP, IRPC, DEFINE, 5.1, 5.2.2
[ ]		[...]	delimits literals; delimits argument in ARRAY, .COMMON, and OPDEF pseudo-ops; quotes characters for macro argument handling.	2.3, ARRAY, .COMMON, OPDEF, 5.2.2
=	between symbol and expression	sym=exp	assigns value of exp to sym.	2.4.2.2, 4.5
==	between symbol and expression	sym==exp	assigns value of exp to sym but sym is not output by debugger.	2.4.2.2, 4.5
=:	between symbol and expression	sym=:exp	assigns value of exp to sym and declares sym as global INTERNAL.	2.4.2.2, 4.5
==:	between symbol and expression	sym==:exp	assigns value of exp to sym and declares sym as global INTERNAL, but sym is not output by debugger.	2.4.2.2, 4.5

MACRO SPECIAL CHARACTERS

APPENDIX C

MACRO-DEFINED MNEMONICS

This appendix contains tables showing all of MACRO's defined mnemonics and the code they generate. These mnemonics, together with the pseudo-ops and the special characters given in Appendix B, make up the entire MACRO language.

NOTE

Throughout this appendix, the following notes apply to the tables:

* Indicates mnemonic defined only if MACRO is assembled with the KL10 switch on.

** Indicates mnemonic defined only if MACRO is assembled with the KI10 switch on.

*** Indicates mnemonic defined only if MACRO is assembled with the F40 switch on.

C.1 MACHINE INSTRUCTION MNEMONICS

Table C-1 shows MACRO's machine instruction mnemonics and the code assembled by each mnemonic. See Section 4.7 for a discussion of machine instructions used in programs.

Table C-1
Machine Instruction Mnemonics

270	00	0	00	000000	ADD	303	00	0	00	000000	CAILE
273	00	0	00	000000	ADDB	306	00	0	00	000000	CAIN
271	00	0	00	000000	ADDI	310	00	0	00	000000	CAM
272	00	0	00	000000	ADDM	314	00	0	00	000000	CAMA
133	00	0	00	000000	*ADJBP	312	00	0	00	000000	CAME
105	00	0	00	000000	*ADJSP	317	00	0	00	000000	CAMG
404	00	0	00	000000	AND	315	00	0	00	000000	CAMGE
407	00	0	00	000000	ANDB	311	00	0	00	000000	CAML
410	00	0	00	000000	ANDCA	313	00	0	00	000000	CAMLE
413	00	0	00	000000	ANDCAB	316	00	0	00	000000	CAMN
411	00	0	00	000000	ANDCAI	400	00	0	00	000000	CLEAR
412	00	0	00	000000	ANDCAM	403	00	0	00	000000	CLEARB
440	00	0	00	000000	ANDCB	401	00	0	00	000000	CLEARI
443	00	0	00	000000	ANDCBB	402	00	0	00	000000	CLEARM
441	00	0	00	000000	ANDCBI	114	00	0	00	000000	*DADD
442	00	0	00	000000	ANDCBM	117	00	0	00	000000	*DDIV
420	00	0	00	000000	ANDCM	110	00	0	00	000000	**DFAD
423	00	0	00	000000	ANDCMB	113	00	0	00	000000	**DFDV
421	00	0	00	000000	ANDCMI	112	00	0	00	000000	**DFMP
422	00	0	00	000000	ANDCMM	131	00	0	00	000000	DFN
405	00	0	00	000000	ANDI	111	00	0	00	000000	**DFSB
406	00	0	00	000000	ANDM	234	00	0	00	000000	DIV
253	00	0	00	000000	AOBJN	237	00	0	00	000000	DIVB
252	00	0	00	000000	AOBJP	235	00	0	00	000000	DIVI
340	00	0	00	000000	AOJ	236	00	0	00	000000	DIVM
344	00	0	00	000000	AOJA	120	00	0	00	000000	**DMOVE
342	00	0	00	000000	AOJE	124	00	0	00	000000	**DMOVEM
347	00	0	00	000000	AOJG	121	00	0	00	000000	**DMOVN
345	00	0	00	000000	AOJGE	125	00	0	00	000000	**DMOVNM
341	00	0	00	000000	AOJL	116	00	0	00	000000	*DMUL
343	00	0	00	000000	AOJLE	137	00	0	00	000000	DPB
346	00	0	00	000000	AOJN	115	00	0	00	000000	*DSUB
350	00	0	00	000000	AOS	444	00	0	00	000000	EQV
354	00	0	00	000000	AOSA	447	00	0	00	000000	EQVB
352	00	0	00	000000	AOSE	445	00	0	00	000000	EQVI
357	00	0	00	000000	AOSG	446	00	0	00	000000	EQVM
355	00	0	00	000000	AOSGE	250	00	0	00	000000	EXCH
351	00	0	00	000000	AOSL	123	00	0	00	000000	*EXTEND
353	00	0	00	000000	AOSLE	140	00	0	00	000000	FAD
356	00	0	00	000000	AOSN	143	00	0	00	000000	FADB
320	00	0	00	000000	ARG	141	00	0	00	000000	FADL
240	00	0	00	000000	ASH	142	00	0	00	000000	FADM
244	00	0	00	000000	ASHC	144	00	0	00	000000	FADR
251	00	0	00	000000	BLT	147	00	0	00	000000	FADRB
300	00	0	00	000000	CAI	145	00	0	00	000000	FADRI
304	00	0	00	000000	CAIA	146	00	0	00	000000	FADRM
302	00	0	00	000000	CAIE	170	00	0	00	000000	FDV
307	00	0	00	000000	CAIG	173	00	0	00	000000	FDVB
305	00	0	00	000000	CAIGE	171	00	0	00	000000	FDVL
301	00	0	00	000000	CAIL	172	00	0	00	000000	FDVM

(Continued on Next Page)

Table C-1 (Cont.)
Machine Instruction Mnemonics

174	00	0	00	000000	FDVR
177	00	0	00	000000	FDVRB
175	00	0	00	000000	FDVRI
176	00	0	00	000000	FDVRM
126	00	0	00	000000	**FIXR
127	00	0	00	000000	**FLTR
160	00	0	00	000000	FMP
163	00	0	00	000000	FMPB
161	00	0	00	000000	FMPL
162	00	0	00	000000	FMPM
164	00	0	00	000000	FMPR
167	00	0	00	000000	FMPRB
165	00	0	00	000000	FMPRI
166	00	0	00	000000	FMPRM
150	00	0	00	000000	FSB
153	00	0	00	000000	FSBB
151	00	0	00	000000	FSBL
152	00	0	00	000000	FSBM
154	00	0	00	000000	FSBR
157	00	0	00	000000	FSBRB
155	00	0	00	000000	FSBRI
156	00	0	00	000000	FSBRM
132	00	0	00	000000	FSC
500	00	0	00	000000	HLL
530	00	0	00	000000	HLLE
531	00	0	00	000000	HLLEI
532	00	0	00	000000	HLLEM
533	00	0	00	000000	HLLES
501	00	0	00	000000	HLLI
502	00	0	00	000000	HLLM
520	00	0	00	000000	HLLO
521	00	0	00	000000	HLLOI
522	00	0	00	000000	HLLOM
523	00	0	00	000000	HLLOS
503	00	0	00	000000	HLLS
510	00	0	00	000000	HLLZ
511	00	0	00	000000	HLLZI
512	00	0	00	000000	HLLZM
513	00	0	00	000000	HLLZS
544	00	0	00	000000	HLR
574	00	0	00	000000	HLRE
575	00	0	00	000000	HLREI
576	00	0	00	000000	HLREM
577	00	0	00	000000	HLRES
545	00	0	00	000000	HLRI
546	00	0	00	000000	HLRM
564	00	0	00	000000	HLRO
565	00	0	00	000000	HLROI
566	00	0	00	000000	HLROM
567	00	0	00	000000	HLROS
547	00	0	00	000000	HLRS
554	00	0	00	000000	HLRZ
555	00	0	00	000000	HLRZI
556	00	0	00	000000	HLRZM
557	00	0	00	000000	HLRZS
504	00	0	00	000000	HRL
534	00	0	00	000000	HRLE
535	00	0	00	000000	HRLEI
536	00	0	00	000000	HRLEM
537	00	0	00	000000	HRLES
505	00	0	00	000000	HRLI
506	00	0	00	000000	HRLM
524	00	0	00	000000	HRLO
525	00	0	00	000000	HRLOI
526	00	0	00	000000	HRLOM
527	00	0	00	000000	HRLOS
507	00	0	00	000000	HRLS
514	00	0	00	000000	HRLZ
515	00	0	00	000000	HRLZI
516	00	0	00	000000	HRLZM
517	00	0	00	000000	HRLZS
540	00	0	00	000000	HRR
570	00	0	00	000000	HRRE
571	00	0	00	000000	HRREI
572	00	0	00	000000	HRREM
573	00	0	00	000000	HRRES
541	00	0	00	000000	HRRI
542	00	0	00	000000	HRRM
560	00	0	00	000000	HRRO
561	00	0	00	000000	HRROI
562	00	0	00	000000	HRROM
563	00	0	00	000000	HRROS
543	00	0	00	000000	HRRS
550	00	0	00	000000	HRRZ
551	00	0	00	000000	HRRZI
552	00	0	00	000000	HRRZM
553	00	0	00	000000	HRRZS
133	00	0	00	000000	IBP
230	00	0	00	000000	IDIV
233	00	0	00	000000	IDIVB
231	00	0	00	000000	IDIVI
232	00	0	00	000000	IDIVM
136	00	0	00	000000	IDPB
134	00	0	00	000000	ILDB
220	00	0	00	000000	IMUL
223	00	0	00	000000	IMULB
221	00	0	00	000000	IMULI
222	00	0	00	000000	IMULM
434	00	0	00	000000	IOR
437	00	0	00	000000	IORB

(Continued on Next Page)

Table C-1 (Cont.)
Machine Instruction Mnemonics

435	00	0	00	000000	IORI
436	00	0	00	000000	IORM
255	00	0	00	000000	JFCL
243	00	0	00	000000	JFFO
267	00	0	00	000000	JRA
254	00	0	00	000000	JRST
266	00	0	00	000000	JSA
265	00	0	00	000000	JSP
264	00	0	00	000000	JSR
104	00	0	00	000000	JSYS
320	00	0	00	000000	JUMP
324	00	0	00	000000	JUMPA
322	00	0	00	000000	JUMPE
327	00	0	00	000000	JUMPG
325	00	0	00	000000	JUMPGE
321	00	0	00	000000	JUMPL
323	00	0	00	000000	JUMPLE
326	00	0	00	000000	JUMPN
135	00	0	00	000000	LDB
242	00	0	00	000000	LSH
246	00	0	00	000000	LSHC
257	00	0	00	000000	**MAP
200	00	0	00	000000	MOVE
201	00	0	00	000000	MOVEI
202	00	0	00	000000	MOVEM
203	00	0	00	000000	MOVES
214	00	0	00	000000	MOVM
215	00	0	00	000000	MOVMI
216	00	0	00	000000	MOVMM
217	00	0	00	000000	MOVMS
210	00	0	00	000000	MOVN
211	00	0	00	000000	MOVNI
212	00	0	00	000000	MOVNM
213	00	0	00	000000	MOVNS
204	00	0	00	000000	MOVS
205	00	0	00	000000	MOVSI
206	00	0	00	000000	MOVSM
207	00	0	00	000000	MOVSS
224	00	0	00	000000	MUL
227	00	0	00	000000	MULB
225	00	0	00	000000	MULI
226	00	0	00	000000	MULM
434	00	0	00	000000	OR
437	00	0	00	000000	ORB
454	00	0	00	000000	ORCA
457	00	0	00	000000	ORCAB
455	00	0	00	000000	ORCAI
456	00	0	00	000000	ORCAM
470	00	0	00	000000	ORCB
473	00	0	00	000000	ORCBB
471	00	0	00	000000	ORCBI
472	00	0	00	000000	ORCBM
464	00	0	00	000000	ORCM
467	00	0	00	000000	ORCMB
465	00	0	00	000000	ORCMI
466	00	0	00	000000	ORCMM
435	00	0	00	000000	ORI
436	00	0	00	000000	ORM
262	00	0	00	000000	POP
263	00	0	00	000000	POPJ
261	00	0	00	000000	PUSH
260	00	0	00	000000	PUSHJ
241	00	0	00	000000	ROT
245	00	0	00	000000	ROTC
424	00	0	00	000000	SETA
427	00	0	00	000000	SETAB
425	00	0	00	000000	SETAI
426	00	0	00	000000	SETAM
450	00	0	00	000000	SETCA
453	00	0	00	000000	SETCAB
451	00	0	00	000000	SETCAI
452	00	0	00	000000	SETCAM
460	00	0	00	000000	SETCM
463	00	0	00	000000	SETCMB
461	00	0	00	000000	SETCMI
462	00	0	00	000000	SETCMM
414	00	0	00	000000	SETM
417	00	0	00	000000	SETMB
415	00	0	00	000000	SETMI
416	00	0	00	000000	SETMM
474	00	0	00	000000	SETO
477	00	0	00	000000	SETOB
475	00	0	00	000000	SETOI
476	00	0	00	000000	SETOM
400	00	0	00	000000	SETZ
403	00	0	00	000000	SETZB
401	00	0	00	000000	SETZI
402	00	0	00	000000	SETZM
330	00	0	00	000000	SKIP
334	00	0	00	000000	SKIPA
332	00	0	00	000000	SKIPE
337	00	0	00	000000	SKIPG
335	00	0	00	000000	SKIPGE
331	00	0	00	000000	SKIPL
333	00	0	00	000000	SKIPLE
336	00	0	00	000000	SKIPN
360	00	0	00	000000	SOJ
364	00	0	00	000000	SOJA
362	00	0	00	000000	SOJE
367	00	0	00	000000	SOJG

(Continued on Next Page)

Table C-1 (Cont.)
Machine Instruction Mnemonics

365	00	0	00	000000	SOJGE	667	00	0	00	000000	TLON
361	00	0	00	000000	SOJL	621	00	0	00	000000	TLZ
363	00	0	00	000000	SOJLE	625	00	0	00	000000	TLZA
366	00	0	00	000000	SOJN	623	00	0	00	000000	TLZE
370	00	0	00	000000	SOS	627	00	0	00	000000	TLZN
374	00	0	00	000000	SOSA	640	00	0	00	000000	TRC
372	00	0	00	000000	SOSE	644	00	0	00	000000	TRCA
377	00	0	00	000000	SOSG	642	00	0	00	000000	TRCE
375	00	0	00	000000	SOSGE	646	00	0	00	000000	TRCN
371	00	0	00	000000	SOSL	600	00	0	00	000000	TRN
373	00	0	00	000000	SOSLE	604	00	0	00	000000	TRNA
376	00	0	00	000000	SOSN	602	00	0	00	000000	TRNE
274	00	0	00	000000	SUB	606	00	0	00	000000	TRNN
277	00	0	00	000000	SUBB	660	00	0	00	000000	TRO
275	00	0	00	000000	SUBI	664	00	0	00	000000	TROA
276	00	0	00	000000	SUBM	662	00	0	00	000000	TROE
650	00	0	00	000000	TDC	666	00	0	00	000000	TRON
654	00	0	00	000000	TDCA	620	00	0	00	000000	TRZ
652	00	0	00	000000	TDCE	624	00	0	00	000000	TRZA
656	00	0	00	000000	TDCN	622	00	0	00	000000	TRZE
610	00	0	00	000000	TDN	626	00	0	00	000000	TRZN
614	00	0	00	000000	TDNA	651	00	0	00	000000	TSC
612	00	0	00	000000	TDNE	655	00	0	00	000000	TSCA
616	00	0	00	000000	TDNN	653	00	0	00	000000	TSCE
670	00	0	00	000000	TDO	657	00	0	00	000000	TSCN
674	00	0	00	000000	TDOA	611	00	0	00	000000	TSN
672	00	0	00	000000	TDOE	615	00	0	00	000000	TSNA
676	00	0	00	000000	TDON	613	00	0	00	000000	TSNE
630	00	0	00	000000	TDZ	617	00	0	00	000000	TSNN
634	00	0	00	000000	TDZA	671	00	0	00	000000	TSO
632	00	0	00	000000	TDZE	675	00	0	00	000000	TSOA
636	00	0	00	000000	TDZN	673	00	0	00	000000	TSOE
641	00	0	00	000000	TLC	677	00	0	00	000000	TSON
645	00	0	00	000000	TLCA	631	00	0	00	000000	TSZ
643	00	0	00	000000	TLCE	635	00	0	00	000000	TSZA
647	00	0	00	000000	TLCN	633	00	0	00	000000	TSZE
601	00	0	00	000000	TLN	637	00	0	00	000000	TSZN
605	00	0	00	000000	TLNA	130	00	0	00	000000	UFA
603	00	0	00	000000	TLNE	256	00	0	00	000000	XCT
607	00	0	00	000000	TLNN	430	00	0	00	000000	XOR
661	00	0	00	000000	TLO	433	00	0	00	000000	XORB
665	00	0	00	000000	TLOA	431	00	0	00	000000	XORI
663	00	0	00	000000	TLOE	432	00	0	00	000000	XORM

C.2 I/O INSTRUCTION AND DEVICE CODE MNEMONICS

Table C-2 shows MACRO's I/O instruction mnemonics and the code each assembles. Note that I/O machine instructions are executable only in executive mode.

Table C-2
I/O Instruction Mnemonics

7 000 00 0 00 000000	BLKI	7 000 30 0 00 000000	CONSZ
7 000 10 0 00 000000	BLKO	7 000 04 0 00 000000	DATAI
7 000 24 0 00 000000	CONI	7 000 14 0 00 000000	DATAO
7 000 20 0 00 000000	CONO	7 000 04 0 00 000000	RSW
7 000 34 0 00 000000	CONSO		

Table C-3 shows MACRO's I/O device code mnemonics. Each is assembled with the I/O instruction mnemonic DATAI so that the value of the device code will be in its proper field. In the first table entry, for example, the assembled code is:

    7 024 04 0 00 000000

where the 7 and 04 are generated by the DATAI instruction, and the 024 by the ADC device code mnemonic.

NOTE

> MACRO leaves these device code mnemonics as undefined symbols during Pass 1. At the end of Pass 1, the mnemonics are found in MACRO's tables only if one or more I/O instructions have been found.
>
> Therefore, if a device code mnemonic is not assembled in Pass 1, or if no I/O instruction mnemonics were found, MACRO will not have defined the device code mnemonic.

Table C-3
I/O Device Code Mnemonics

7	024	04	0	00	000000	DATAI ADC,
7	030	04	0	00	000000	DATAI ADC2,
7	000	04	0	00	000000	DATAI APR,
7	014	04	0	00	000000	DATAI CCI,
7	110	04	0	00	000000	DATAI CDP,
7	114	04	0	00	000000	DATAI CDR,
7	070	04	0	00	000000	DATAI CLK,
7	074	04	0	00	000000	DATAI CLK2,
7	000	04	0	00	000000	DATAI CPA,
7	150	04	0	00	000000	DATAI CR,
7	154	04	0	00	000000	DATAI CR2,
7	200	04	0	00	000000	DATAI DC,
7	204	04	0	00	000000	DATAI DC2,
7	300	04	0	00	000000	DATAI DCSA,
7	304	04	0	00	000000	DATAI DCSB,
7	270	04	0	00	000000	DATAI DDC,
7	274	04	0	00	000000	DATAI DDC2,
7	270	04	0	00	000000	DATAI DF,
7	130	04	0	00	000000	DATAI DIS,
7	134	04	0	00	000000	DATAI DIS2,
7	060	04	0	00	000000	DATAI DLB,
7	160	04	0	00	000000	DATAI DLB2,
7	064	04	0	00	000000	DATAI DLC,
7	164	04	0	00	000000	DATAI DLC2,
7	240	04	0	00	000000	DATAI DLS,
7	244	04	0	00	000000	DATAI DLS2,
7	250	04	0	00	000000	DATAI DPC,
7	254	04	0	00	000000	DATAI DPC2,
7	260	04	0	00	000000	DATAI DPC3,
7	264	04	0	00	000000	DATAI DPC4,
7	464	04	0	00	000000	DATAI DSI,
7	474	04	0	00	000000	DATAI DSI2,
7	170	04	0	00	000000	DATAI DSK,
7	174	04	0	00	000000	DATAI DSK2,
7	460	04	0	00	000000	DATAI DSS,
7	470	04	0	00	000000	DATAI DSS2,
7	320	04	0	00	000000	DATAI DTC,
7	330	04	0	00	000000	DATAI DTC2,
7	324	04	0	00	000000	DATAI DTS,
7	334	04	0	00	000000	DATAI DTS2,
7	124	04	0	00	000000	DATAI LPT,
7	234	04	0	00	000000	DATAI LPT2,
7	260	04	0	00	000000	DATAI MDF,
7	264	04	0	00	000000	DATAI MDF2,
7	220	04	0	00	000000	DATAI MTC,
7	230	04	0	00	000000	DATAI MTM,
7	224	04	0	00	000000	DATAI MTS,
7	010	04	0	00	000000	DATAI PAG,

(Continued on Next Page)

Table C-3 (Cont.)
I/O Device Code Mnemonics

7	004	04	0	00	000000	DATAI PI,
7	140	04	0	00	000000	DATAI PLT,
7	144	04	0	00	000000	DATAI PLT2,
7	100	04	0	00	000000	DATAI PTP,
7	104	04	0	00	000000	DATAI PTR,
7	340	04	0	00	000000	DATAI TMC,
7	350	04	0	00	000000	DATAI TMC2,
7	344	04	0	00	000000	DATAI TMS,
7	354	04	0	00	000000	DATAI TMS2,
7	120	04	0	00	000000	DATAI TTY,
7	210	04	0	00	000000	DATAI UTC,
7	214	04	0	00	000000	DATAI UTS,

C.3 KL10 EXTEND INSTRUCTION MNEMONICS

Table C-4 shows the KL10 EXTEND instruction mnemonics and the code assembled by each. All of these mnemonics are defined only if MACRO is assembled with the KL10 switch on.

See the Supplement to the Hardware Reference Manual for a discussion of these EXTEND instructions.

Table C-4
KL10 EXTEND Instruction Mnemonics

002 00 0 00 000000	*CMPSE	010 00 0 00 000000	*CVTDBO
007 00 0 00 000000	*CMPSG	011 00 0 00 000000	*CVTDBT
005 00 0 00 000000	*CMPSGE	004 00 0 00 000000	*EDIT
001 00 0 00 000000	*CMPSL	016 00 0 00 000000	*MOVSLJ
003 00 0 00 000000	*CMPSLE	014 00 0 00 000000	*MOVSO
006 00 0 00 000000	*CMPSN	017 00 0 00 000000	*MOVSRJ
012 00 0 00 000000	*CVTBDO	015 00 0 00 000000	*MOVST
013 00 0 00 000000	*CVTBDT	020 00 0 00 000000	*XBLT

C.4 JRST AND JFCL MNEMONICS

Table C-5 shows mnemonics that assemble both operator and accumulator fields in the machine instruction. The left side of the table shows the mnemonics and the code they generate; the right side shows JRST and JFCL mnemonics with accumulators generating the equivalent code.

Table C-5
JRST and JFCL Mnemonics

Code and Mnemonic		Equivalent Code and Mnemonic	
254 04 0 00 000000	HALT	254 04 0 00 000000	JRST 4,
255 06 0 00 000000	JCRY	255 06 0 00 000000	JFCL 6,
255 04 0 00 000000	JCRY0	255 04 0 00 000000	JFCL 4,
255 02 0 00 000000	JCRY1	255 02 0 00 000000	JFCL 2,
254 12 0 00 000000	JEN	254 12 0 00 000000	JRST 12,
255 01 0 00 000000	JFOV	255 01 0 00 000000	JFCL 1,
255 10 0 00 000000	JOV	255 10 0 00 000000	JFCL 10,
254 02 0 00 000000	JRSTF	254 02 0 00 000000	JRST 2,
254 01 0 00 000000	PORTAL	254 01 0 00 000000	JRST 1,
254 06 0 00 000000	*XJEN	254 06 0 00 000000	JRST 6,
254 05 0 00 000000	*XJRSTF	254 05 0 00 000000	JRST 5,
254 07 0 00 000000	*XPCW	254 07 0 00 000000	JRST 7,
254 14 0 00 000000	*XSFM	254 14 0 00 000000	JRST 14,

C.5 DECsystem-10 MONITOR CALL MNEMONICS

Tables C-6 through C-9 show MACRO's DECsystem-10 monitor call mnemonics and related mnemonics.

Table C-6 shows DECsystem-10 monitor calls and their assembled code.

Tables C-7 through C-9 show DECsystem-10 CALLI, TTCALL, and MTAPE mnemonics and their assembled code. These mnemonics are defined only if MACRO's UUOSYM switch is on when MACRO is assembled.

Table C-6
DECsystem-10 Monitor Calls

040	00	0	00	000000	CALL	067	00	0	00	000000	OUTPUT
047	00	0	00	000000	CALLI	052	00	0	00	000000	*RDCLK
070	00	0	00	000000	CLOSE	071	00	0	00	000000	RELEAS
077	00	0	00	000000	ENTER	055	00	0	00	000000	RENAME
062	00	0	00	000000	GETSTS	060	00	0	00	000000	SETSTS
056	00	0	00	000000	IN	061	00	0	00	000000	STATO
064	00	0	00	000000	INBUF	062	00	0	00	000000	STATUS
041	00	0	00	000000	INIT	063	00	0	00	000000	STATZ
066	00	0	00	000000	INPUT	051	00	0	00	000000	TTCALL
076	00	0	00	000000	LOOKUP	073	00	0	00	000000	UGETF
072	00	0	00	000000	MTAPE	100	00	0	00	000000	UJEN
050	00	0	00	000000	OPEN	074	00	0	00	000000	USETI
057	00	0	00	000000	OUT	075	00	0	00	000000	USETO
065	00	0	00	000000	OUTBUF						

Table C-7
DECsystem-10 CALLI Mnemonics

047	00	0	00	000161	ALLOC.	047	00	0	00	000144	IPCFQ.
047	00	0	00	000016	APRENB	047	00	0	00	000142	IPCFR.
047	00	0	00	000104	ATTACH	047	00	0	00	000143	IPCFS.
047	00	0	00	000125	CAL11.	047	00	0	00	000113	JBSET.
047	00	0	00	000156	CAL78.	047	00	0	00	000103	JOBPEK
047	00	0	00	000074	CHGPPN	047	00	0	00	000047	JOBSTR
047	00	0	00	000100	CHKACC	047	00	0	00	000061	JOBSTS
047	00	0	00	000134	CLRST.	047	00	0	00	777777	LIGHTS
047	00	0	00	000130	CNECT.	047	00	0	00	000062	LOCATE
047	00	0	00	000147	COMPT.	047	00	0	00	000060	LOCK
047	00	0	00	000011	CORE	047	00	0	00	000015	LOGIN
047	00	0	00	000065	CTLJOB	047	00	0	00	000017	LOGOUT
047	00	0	00	000105	DAEFIN	047	00	0	00	000111	METER.
047	00	0	00	000102	DAEMON	047	00	0	00	000023	MSTIME
047	00	0	00	000014	DATE	047	00	0	00	000126	MTAID.
047	00	0	00	000005	DDTGT	047	00	0	00	000112	MTCHR.
047	00	0	00	000001	DDTIN	047	00	0	00	000131	MVHDR.
047	00	0	00	000003	DDTOUT	047	00	0	00	000157	NODE.
047	00	0	00	000007	DDTRL	047	00	0	00	000077	OTHUSR
047	00	0	00	000137	DEBRK.	047	00	0	00	000145	PAGE.
047	00	0	00	000152	DEQ.	047	00	0	00	000110	PATH.
047	00	0	00	000004	DEVCHR	047	00	0	00	000033	PEEK
047	00	0	00	000076	DEVGEN	047	00	0	00	000162	PERF.
047	00	0	00	000107	DEVLNM	047	00	0	00	000135	PIINI.
047	00	0	00	000064	DEVNAM	047	00	0	00	000141	PIRST.
047	00	0	00	000055	DEVPPN	047	00	0	00	000140	PISAV.
047	00	0	00	000101	DEVSIZ	047	00	0	00	000136	PISYS.
047	00	0	00	000054	DEVSTS	047	00	0	00	000030	PJOB
047	00	0	00	000053	DEVTYP	047	00	0	00	000114	POKE.
047	00	0	00	000121	DISK.	047	00	0	00	000021	REASSI
047	00	0	00	000045	DSKCHR	047	00	0	00	000037	REMAP
047	00	0	00	000122	DVRST.	047	00	0	00	000117	RESDV.
047	00	0	00	000123	DVURS.	047	00	0	00	000000	RESET
047	00	0	00	000151	ENQ.	047	00	0	00	000057	RTTRP
047	00	0	00	000153	ENQC.	047	00	0	00	000035	RUN
047	00	0	00	000132	ERLST.	047	00	0	00	000027	RUNTIM
047	00	0	00	000160	ERRPT.	047	00	0	00	000150	SCHED.
047	00	0	00	000012	EXIT	047	00	0	00	000056	SEEK
047	00	0	00	000155	FILOP.	047	00	0	00	000133	SENSE.
047	00	0	00	000106	FRCUUO	047	00	0	00	000002	SETDDT
047	00	0	00	000052	FRECHN	047	00	0	00	000043	SETNAM
047	00	0	00	000006	GETCHR	047	00	0	00	000032	SETPOV
047	00	0	00	000034	GETLIN	047	00	0	00	000075	SETUUO
047	00	0	00	000024	GETPPN	047	00	0	00	000036	SETUWP
047	00	0	00	000040	GETSEG	047	00	0	00	000031	SLEEP
047	00	0	00	000041	GETTAB	047	00	0	00	000042	SPY
047	00	0	00	000066	GOBSTR	047	00	0	00	000050	STRUUO
047	00	0	00	000072	HIBER	047	00	0	00	000146	SUSET.
047	00	0	00	000071	HPQ	047	00	0	00	000020	SWITCH
047	00	0	00	000127	IONDX.	047	00	0	00	000051	SYSPHY

(Continued on Next Page)

Table C-7 (Cont.)
DECsystem-10 CALLI Mnemonics

047 00 0 00 000046	SYSSTR	047 00 0 00 000025	TRPSET
047 00 0 00 000154	TAPOP.	047 00 0 00 000120	UNLOK.
047 00 0 00 000022	TIMER	047 00 0 00 000013	UTPCLR
047 00 0 00 000044	TMPCOR	047 00 0 00 000010	WAIT
047 00 0 00 000115	TRMNO.	047 00 0 00 000073	WAKE
047 00 0 00 000116	TRMOP.	047 00 0 00 000063	WHERE
047 00 0 00 000026	TRPJEN	047 00 0 00 000124	XTTSK.

Table C-8
DECsystem-10 TTCALL Mnemonics

051 11 0 00 000000	CLRBFI	051 15 0 00 000000	IONEOU
051 12 0 00 000000	CLRBFO	051 01 0 00 000000	OUTCHR
051 06 0 00 000000	GETLCH	051 03 0 00 000000	OUTSTR
051 02 0 00 000000	INCHRS	051 10 0 00 000000	RESCAN
051 00 0 00 000000	INCHRW	051 07 0 00 000000	SETLCH
051 05 0 00 000000	INCHSL	051 13 0 00 000000	SKPINC
051 04 0 00 000000	INCHWL	051 14 0 00 000000	SKPINL

Table C-9
DECsystem-10 MTAPE Mnemonics

072 00 0 00 000013	MTBLK.	072 00 0 00 000101	MTIND.
072 00 0 00 000017	MTBSF.	072 00 0 00 000001	MTREW.
072 00 0 00 000007	MTBSR.	072 00 0 00 000016	MTSKF.
072 00 0 00 000100	MTDEC.	072 00 0 00 000006	MTSKR.
072 00 0 00 000003	MTEOF.	072 00 0 00 000011	MTUNL.
072 00 0 00 000010	MTEOT.	072 00 0 00 000000	MTWAT.

C.6 F40 UUO MNEMONICS

Table C-10 shows mnemonics that are defined only if MACRO is assembled with the F40 switch on. These mnemonics generate UUOs, which are handled properly if the program is running under control of the FORSE object-time system.

Table C-10
F40 UUO Mnemonics

020 00 0 00 000000 ***DATA.	017 00 0 00 000000 ***OUT.
021 00 0 00 000000 ***FIN.	027 00 0 00 000000 ***OUTF.
016 00 0 00 000000 ***IN.	015 00 0 00 000000 ***RESET.
026 00 0 00 000000 ***INF.	022 00 0 00 000000 ***RTB.
024 00 0 00 000000 ***MTOP.	025 00 0 00 000000 ***SLIST.
031 00 0 00 000000 ***NLI.	023 00 0 00 000000 ***WTB.
032 00 0 00 000000 ***NLO.	

APPENDIX D

PROGRAM EXAMPLES

The following pages contain examples of MACRO programs. Each program has been assembled with the /C (CREF) switch on; this produces a .CRF file for the program listing (instead of the usual .LST file). The /O switch has been used with the CREF program to produce a .LST file that includes all operators in an operator symbol table.

```
MACROS  MACRO %53(1017) 16:12  2-Mar-78 Page 1
EXAMPL  MAC      2-Mar-78 15:03         Example One

                                        SUBTTL Example One
                                        UNIVERSAL MACROS

                                        ;This UNIVERSAL program contains the macro QUIT, which uses
                                        ;  conditional assembly to generate a program exit monitor
                                        ;  call.  If the TOPS10 switch is on when QUIT is called (or if
                                        ;  it is undefined), QUIT generates 'EXIT'; if the switch
                                        ;  is off, QUIT generates 'HALTF'.

                                        DEFINE QUIT <
                                                IFNDEF TOPS10,<
                                                TOPS10==-1              ;;Default is TOPS10
                                        >
                                                IFE TOPS10,<
                                                HALTF
                                        >
                                                IFN TOPS10,<
                                                EXIT
                                        >
                                        >
                                        PRGEND

NO ERRORS DETECTED

PROGRAM BREAK IS 000000
CPU TIME USED 00:00.411

10P CORE USED
```

```
Second Example of MACRO Program MACRO %53(1017) 16:12  2-Mar-78 Page 2
EXAMPL  MAC      2-Mar-78 15:03         Example Two

                                        SUBTTL Example Two
                                        TITLE Second Example of MACRO Program

                                        ;This program contains the macros CLEAR, CONCAT, and EXPAND.
                                        ;  These can be used to append arbitrary text into a buffer,
                                        ;  and to recall the text later.  Two sequences of calls
                                        ;  to the macros show possible uses.
                                        ;
                                        ;The following points are of interest:
                                        ;
                                        ;  1.  The buffer is cleared by calling CLEAR.  Text is added
                                        ;      (on the right side of the buffer) by calling CONCAT.
                                        ;      EXPAND, when used in a context allowed for macro calls,
                                        ;      expands the contents of the buffer into source code.
                                        ;
                                        ;  2.  A call to CLEAR defines the text buffer, EXPAND, to
                                        ;      contain no text.  It also defines the macro CONCAT in
                                        ;      such a way that the first call to CONCAT redefines
                                        ;      EXPAND to contain the first piece of text, and CONCAT
                                        ;      redefines itself so that further calls to CONCAT will
                                        ;      call the internal macro CON1.  Following the second
                                        ;      call to CONCAT, each further call merely appends new
                                        ;      text to the old.
                                        ;
                                        ;  3.  A key feature of EXPAND is that it contains no carriage
                                        ;      returns.  If it did, then each concatenation of new
                                        ;      text would also insert a carriage return into the text.
                                        ;
                                        ;  4.  The first use of these macros shows that EXPAND can be
                                        ;      placed in contexts where more than one argument will
                                        ;      result (as in the BYTE pseudo-op).  Note that because
                                        ;      angle brackets are used internally (inside the macros)
                                        ;      to delimit text, all concatenated text must contain
                                        ;      matched angle brackets.
                                        ;
                                        ;  5.  Note that carriage returns, if desired, can be easily
                                        ;      concatenated to the buffer; this is done in the second
                                        ;      use of the macros.

                                        ;  . . .
```

```
Second Example of MACRO Program MACRO %53(1017) 16:12  2-Mar-78 Page 3
EXAMPL  MAC      2-Mar-78 15:03          Example Two

                                         ;   . . .

                                         DEFINE CLEAR <
                                            DEFINE CONCAT (FTXT) <
                                               DEFINE CONCAT (TEXT) <
                                                  CON1 <TEXT>,<FTXT>
                                               >
                                               DEFINE EXPAND <FTXT>
                                            >
                                            DEFINE EXPAND <>
                                         >

                                         DEFINE CON1 (NTXT,OTXT) <
                                            DEFINE CONCAT (TEXT) <
                                               CON1 <TEXT>,<OTXT'NTXT>
                                            >
                                            DEFINE EXPAND <OTXT'NTXT>
                                         >

                                         SALL

                                         CLEAR

                                         CONCAT <10>
                                         CONCAT <,>
                                         CONCAT <"A">
                                         CONCAT <,<<-1,,6>&177>>

                                         LALL
         000000' 010 101 006 00000       BYTE (7)EXPAND^10,"A",<<-1,,6>&177>^

                                         SALL
                                         CLEAR

                                         CONCAT <DEF>
                                         CONCAT <INE FOO (>
                                         CONCAT <N)>
                                         CONCAT <<2*N>
                                         DEFINE>
                                         CONCAT < BAR (N) <3*N>
                                         >

                                         ;   . . .
```

```
Second Example of MACRO Program MACRO %53(1017) 16:12  2-Mar-78 Page 4
EXAMPL  MAC      2-Mar-78 15:03          Example Two

                                          ;  . . .

                                          LALL
                                          EXPAND^DEFINE FOO (N)<2*N>
                                          DEFINE BAR (N) <3*N>
                                          ^

     000001' 000000  000004               FOO 2^2*2^
     000002' 000000  000006               FOO 3^2*3^
     000003' 000000  000006               BAR 2^3*2^
     000004' 000000  000011               BAR 3^3*3^

                                          PRGEND

NO ERRORS DETECTED

PROGRAM BREAK IS 000005
CPU TIME USED 00:00.180

10P CORE USED
```

```
Third Example of MACRO Program  MACRO %53(1017) 16:12  2-Mar-78 Page 5
EXAMPL  MAC      2-Mar-78 15:03           Example Three

                                SUBTTL Example Three
                                TITLE Third Example of MACRO Program

                                ;This program uses the macros NUMLST and X to generate parallel
                                ;  tables.
                                ;
                                ;This example generates a table that contains keywords suitable
                                ;  for comparison to user input; the second table generated
                                ;  contains addresses of routines that handle those keywords;
                                ;  the third table contains useful values.
                                ;
                                ;The keyword table is arranged alphabetically to speed searching;
                                ;  the other two tables correspond entry-for-entry to the
                                ;  keyword table.
                                ;
                                ;Key features of this program include:
                                ;
                                ;  1.  Changing the size of the tables is easy.  For example,
                                ;      if a new entry, FIFTH, is needed, adding the word and
                                ;      a dummy label to the definition of NUMLST will update
                                ;      both tables; no separate update is required.
                                ;
                                ;  2.  The macro NUMLST calls the macro X.  Before each call
                                ;      to NUMLST, X is redefined so that the proper kind of
                                ;      table is built.  Note that a definition of X need not
                                ;      use both arguments in the macrobody.  (However, X should
                                ;      define both arguments.)
                                ;
                                ;  3.  The second definition of X uses concatenation to build
                                ;      mnemonic labels for the table LBLTBL.
                                ;
                                ;  4.  The program uses the macro QUIT so that it can be used
                                ;      for either TOPS-10 or TOPS-20.  The SEARCH MACROS statement
                                ;      makes the definition of QUIT available; since the default
                                ;      for QUIT is TOPS-10, the program will run on TOPS-10 if
                                ;      either it defines TOPS10=-1 or does not define TOPS10;
                                ;      the program will run on TOPS-20 only if it defines
                                ;      TOPS10=0.

                                ;  . . .
```

```
Third Example of MACRO Program  MACRO %53(1017) 16:12  2-Mar-78 Page 6
EXAMPL  MAC  .   2-Mar-78 15:03         Example Three

                                        ; . . .

                                        SEARCH MACROS
                                        .DIRECTIVE SFCOND

                                        DEFINE NUMLST <
                                           X (FIRST,1)
                                           X (FOURTH,4)
                                           X (SECOND,2)
                                           X (THIRD,3)
                                        >

                                        DEFINE X (TEXT,JUNK) <EXP SIXBIT /TEXT/>

     000000'                            NAMTBL: NUMLST^
     000000' 465162  636400                X (FIRST,1)^EXP SIXBIT /FIRST/^
     000001' 465765  626450                X (FOURTH,4)^EXP SIXBIT /FOURTH/^
     000002' 634543  575644                X (SECOND,2)^EXP SIXBIT /SECOND/^
     000003' 645051  624400                X (THIRD,3)^EXP SIXBIT /THIRD/^
                     000004             TBLLEN==.-NAMTBL

                                        DEFINE X (JUNK,LABL) <$'LABL>

     000004'                            LBLTBL: NUMLST^
     000004' 000000  000014'               X (FIRST,1)^$1^
     000005' 000000  000017'               X (FOURTH,4)^$4^
     000006' 000000  000015'               X (SECOND,2)^$2^
     000007' 000000  000016'               X (THIRD,3)^$3^

                                        DEFINE X (JUNK,VALU) <DEC VALU>

     000010'                            VALTBL: NUMLST^
     000010' 000000  000001                X (FIRST,1)^DEC 1^
     000011' 000000  000004                X (FOURTH,4)^DEC 4^
     000012' 000000  000002                X (SECOND,2)^DEC 2^
     000013' 000000  000003                X (THIRD,3)^DEC 3^

                                        ; . . .
```

```
Third Example of MACRO Program  MACRO %53(1017) 16:12  2-Mar-78 Page 7
EXAMPL  MAC      2-Mar-78 15:03          Example Three

                                         ;  . . .

                                         XALL
000014'                                  $1:     QUIT^
000014' 047 00 0 00 000012                       EXIT
000015'                                  $2:     QUIT^
000015' 047 00 0 00 000012                       EXIT
000016'                                  $3:     QUIT^
000016' 047 00 0 00 000012                       EXIT
000017'                                  $4:     QUIT^
000017' 047 00 0 00 000012                       EXIT

                                         PRGEND

NO ERRORS DETECTED

PROGRAM BREAK IS 000020
CPU TIME USED 00:00.142

10P CORE USED
```

```
Third Example of MACRO Program  MACRO %53(1017) 16:12  2-Mar-78 Page S-1
EXAMPL  MAC      2-Mar-78 15:03         SYMBOL TABLE

EXIT     047000  000012
LBLTBL           000004'
NAMTBL           000000'
TBLLEN           000004  spd
TOPS10   777777  777777  spd
VALTBL           000010'
$1               000014'
$2               000015'
$3               000016'
$4               000017'
```

```
SUBTTL Example Four
TITLE Fourth Example of MACRO Program

;This program contains a complex and useful macro, COMMON.
;  The macro allows declaration of variable names for a
;  FORTRAN-compatible COMMON block.  Note that the pseudo-op
;  .COMMON allows declaration of a COMMON block, but not of
;  variable names within the block.
;
;The COMMON macro uses two arguments:
;
;  1.  The name of the COMMON block.
;
;  2.  An IRP-style list of the variable names for the block.
;      The list can contain either variable names only (with
;      an assumed length of one word for each variable), or
;      can contain an angle-bracketed pair giving the name and
;      the length in decimal.
;
;Key features of the program include:
;
;  1.  Lengths for variables are given in decimal numbers,
;      so that the definitions look much like those in the
;      FORTRAN language.  This is accomplished by storing
;      the current radix in a created symbol, and restoring
;      it at the end of the macro.
;
;  2.  The macro uses the technique of IRPing more than once
;      on the IRP list.  The first IRP counts the length of
;      the entire COMMON block, so that the .COMMON pseudo-op
;      can be used; the second IRP declares variable names
;      for each entry in the block.
;
;  3.  The pseudo-ops .XCREF and PURGE are used often
;      in the macro; this is to remove references to created
;      symbols from the CREF listing and the symbol table.
;
;  4.  Created symbols are used in the macro for symbols that
;      are used only within the macro itself.  This minimizes
;      the chance that other definitions will conflict with
;      these symbols.
;
;  5.  Once the COMMON macro has been called, symbols in the
;      COMMON block may be used much as any other symbols;
;      this is shown in the IFIX and ZERO routines.

;  . . .
```

```
;  . . .

DEFINE COMMON (COM,VARS,%RAD,%LEN,%VAL,%COM,%PAS) <
   .XCREF %RAD,%LEN,%VAL,%COM,%PAS

   ;;Temp macro to strip one pair of angle brackets from
   ;; a macro argument and pass it to another macro

   DEFINE %PAS (A,B) <A B>

   ;;Temp macro to compute length of COMMON

   DEFINE %COM (VAR,LEN<1>) <%LEN==%LEN+LEN>

   %RAD==10                          ;;Save current radix, use 10
   RADIX 10                          ;; so defs read like FORTRAN
   %LEN==0                           ;;Set to count length of COMMON
   IRP VARS<%PAS %COM,VARS>          ;;Get length of this COMMON
   .COMMON COM[%LEN]                 ;;Allocate the whole COMMON

   DEFINE %COM (VAR,LEN<1>) <        ;;Set up another temp macro
      VAR=%VAL                       ;;Define COMMON block entry
      %VAL==%VAL+LEN                 ;;Increment to next entry
   >

   %LEN==0                           ;;Reinitialize length
   %VAL==COM                         ;;Start to define entries in block
   IRP VARS<%PAS %COM,VARS>          ;;Define next COMMON entry
   RADIX %RAD                        ;;Restore current radix

   IF2,<PURGE %LEN,%RAD,%VAL,%COM,%PAS> ;;Keep symbol table clean
>

;  . . .
```

```
Fourth Example of MACRO Program MACRO %53(1017) 16:12  2-Mar-78 Page 10
EXAMPL  MAC      2-Mar-78 15:03          Example Four

                                         ;  . . .

                                         ;       INTEGER SNGLE,ARRAY,MULTI
                                         ;       REAL REAL
                                         ;       DOUBLE PRECISION DOUBLE
                                         ;       COMMON /AREA/SNGLE,REAL,DOUBLE,ARRAY(10),MULTI(5,10)

                                         COMMON AREA,<SNGLE,REAL,<DOUBLE,2>,<ARRAY,10>,<MULTI,5*10>>^

                                         ;Sample routine to do SNGLE=IFIX(REAL)

     000000' 122 01 0 00 000000#         IFIX:   FIX 1,REAL
     000001' 202 01 0 00 000000*                 MOVEM 1,SNGLE
     000002' 263 17 0 00 000000                  POPJ 17,

                                         ;Sample routine to set all elements in ARRAY to 0

     000003' 200 01 0 00 000007'         ZERO:   MOVE 1,[XWD ARRAY,ARRAY+1]
     000004' 402 00 0 00 000000#                 SETZM ARRAY
     000005' 251 01 0 00 000000#                 BLT 1,ARRAY+^D9
     000006' 263 17 0 00 000000                  POPJ 17,

     000007'                             LIT
     000007' 000000# 000000#

                                                 END

NO ERRORS DETECTED

PROGRAM BREAK IS 000010
CPU TIME USED 00:00.228

10P CORE USED
```

```
AREA              000001' ext
ARRAY   000000000000#   pol
DOUBLE  000000000000#   pol
IFIX              000000'
MULTI   000000000000#   pol
REAL    000000000000#   pol
SNGLE             000000*
ZERO              000003'
```

AREA	298#	298		
ARRAY	298#	307	308	309
DOUBLE	298#			
IFIX	301#			
LBLTBL	181#			
MULTI	298#			
NAMTBL	172#	177		
REAL	298#	301		
SNGLE	298#	302		
TBLLEN	177#			
TOPS10	200	202	204	206
VALTBL	189#			
ZERO	307#			
$1	182	199#		
$2	184	201#		
$3	185	203#		
$4	183	205#		

BAR	109#	114	115											
CLEAR	64#	83	94											
COMMON	258#	297												
CON1	74#	86	87	88	97	98	100	102						
CONCAT	83#	85	85#	86	86#	87	87#	88	88#	94#	96	96#	97	97#
	98	98#	99	100#	101	102#								
EXIT	200	202	204	206										
EXPAND	83#	85#	86#	87#	88#	91	94#	96#	97#	98#	100#	102#	108	
FOO	108#	112	113											
NUMLST	163#	172	181	189										
QUIT	10#	199	201	203	205									
X	170#	173	174	175	176	179#	182	183	184	185	187#	190	191	192
	193													
..0004	298	298#												
..0005	298													

BLT	309													
BYTE	91													
DEC	190	191	192	193										
DEFINE	10	64	74	83	85	86	87	88	94	96	97	98	100	102
	108	109	163	170	179	187	258	298						
END	315													
EXP	173	174	175	176										
FIX	301													
IF2	298													
IFE	200	202	204	206										
IFN	200	202	204	206										
IFNDEF	200	202	204	206										
IRP	298													
LALL	90	107												
LIT	312													
MOVE	307													
MOVEM	302													
POPJ	303	310												
PRGEND	21	117	208											
PURGE	298													
RADIX	298													
SALL	81	93												
SEARCH	160													
SETZM	308													
SIXBIT	173	174	175	176										
SUBTTL	1	22	118	209										
TITLE	23	119	210											
UNIVER	2													
XALL	198													
XWD	307													
.COMMO	298													
.DIREC	161													

APPENDIX E

PSEUDO-OPS FOR PROGRAM COMPATIBILITY

The pseudo-ops in this appendix are included only for compatibility of old programs to be assembled using MACRO Version 53.

HISEG

FORMAT

HISEG address

address = program high-segment origin address. Must be equal to or greater than 400000 and must be a multiple of 1000.

FUNCTION

Directs the loader to load the current program into the high segment if the program has reentrant (two-segment) capability. HISEG should appear at the beginning of the source program.

HISEG does not affect assembler operation. The code produced by HISEG will execute at either relocatable 0 or relocatable 400000, depending on the loading instructions given.

The code following HISEG looks as if it was assembled to start at relocatable 0.

This pseudo-op has been replaced by TWOSEG.

RIM

FORMAT RIM

FUNCTION Specifies a format for absolute binary programs (useful only for PDP-6 systems), and consists of a series of paired words.

The first word of each pair is a paper-tape read instruction giving the memory address of the second word. The last pair of words is a transfer block; the first is an instruction obtained from the END statement and executed when the transfer block is read, and the second is a dummy word to stop the reader.

RIM10

FORMAT RIM10

FUNCTION Causes a program format in which programs are absolute, unblocked, and not checksummed. When the RIM10 statement follows a LOC statement in a program, the assembler punches out each storage word in the object program, starting at the absolute address specified in the LOC statement. RIM10 writes an arbitary "paper tape"; if it is in the format given below, it can be read by the DECsystem-10 Read-In Mode hardware.

```
IOWD n,first
```

where n is the length of the program including the ending word transfer, and first is the first memory location to be occupied. The last location must contain a transfer instruction to begin the program, such as

```
JRST 4,GO
```

For example, if a program with RIM10 output has its first location at START and its last location at FINISH,you can write

```
IOWD FINISH-START+1,START
```

NOTE

If the location counter is increased but no binary output occurs (for example, BLOCK, LOC, and VAR pseudo-ops), MACRO inserts a zero word into the binary output file for each location skipped by the location counter.

RIM10B

FORMAT RIM10B

FUNCTION

If a program is assembled into absolute locations (not relocatable), a RIM10B statement following the LOC statement at the beginning of the source program causes the assembler to write out the object program in RIM10B format. This format is designed for use with the DECsystem-10 Read-In Mode hardware.

The program is punched during Pass 2, starting at the location specified in the LOC statement. If the first two statements in the program are

```
LOC 1000
RIM10B
```

MACRO assembles the program with absolute addresses starting at 1000 and punches the program in RIM10B format, also starting at location 1000. You can reset the location counter during assembly, but only one RIM10B statement is needed to punch the entire program.

In RIM10B format, the assembler punches the RIM10B Loader, followed by the program in 17-word (or less) data blocks, each block separated by blank tape. The assembler inserts an I/O transfer word (IOWD) preceding each data block, and also inserts a 36-bit checksum following each data block. The word count in the IOWD counts only the data words in the block, and the checksum is the 36-bit added checksum of the IOWD and the data words.

Data blocks can contain less than 17 words. If the assembler assigns a nonconsecutive location, the current data block is terminated, and an IOWD containing the next location is inserted, starting a new data block.

The transfer block consists of two words. The first word of the transfer block is an instruction obtained from the END statement. This first word is executed when the transfer block is read. The second word is a dummy word to stop the reader.

APPENDIX F

STORAGE ALLOCATION

MACRO allocates storage in two directions:

1. User symbols and macronames are entered in the symbol tables.

2. Macros and literals are entered in free space.

A symbol table entry is two words long. The first word is the symbol name in SIXBIT. The second word has flags in the left half, and either the value or a pointer in the right half. The flags indicate symbol type and attributes.

The following list shows how symbols and values are stored.

Type	How Stored
18-bit symbol	Value in right half of second word.
36-bit symbol (includes OPDEFs and negative numbers)	Value in free storage with a pointer in symbol table.
EXTERNAL symbol	Pointer in symbol table to a 2-word block in free storage. The first word is the value that is the last reference in a chain of references to the symbol; the second word is the symbol name in SIXBIT.
Polish symbol	The symbol table entry points to a 2-word block: word 1: 0 word 2: negative number,,address Word 1 is the relocation word and is always zero. Word 2 gives the address of a Polish stack in free storage. The Polish stack is of the form: word 1: 0 word 2: opcode word 3: relocation constant word 4: value word 5: relocation constant word 6: value

Words 3 and 4 designate an operand. If the operator is binary, words 5 and 6 designate the second operand; if the operator is unary, the stack contains only four words.

If an operand is EXTERNAL, its two words (3 and 4, or 5 and 6) are:

word i: pointer to EXTERNAL symbol
word i+1: 0

If an operand is itself a Polish symbol, its two words are:

word i: Polish pointer
word i+1: 0

Inter-PSECT reference	Polish stack containing: word 1: 0 word 2: 15 word 3: -2 word 4: referenced PSECT index word 5: relocation constant word 6: address
Synonym operator (SYN argument)	SIXBIT operator name in free storage with a pointer in the symbol table.
Macroname	Value in free storage with a pointer to the text string in symbol table.

The text string is stored in a 4-word block of the form:

word 1: link to next block (0 if last),,two characters
word 2: five characters
word 3: five characters
word 4: five characters

However, the first such block is special:

word 1: link to next block,,link to last block
word 2: pointer to default arg.,,number of args expected + reference count
word 3: five characters
word 4: five characters

The number of args expected is the number of dummy-arguments in the macro definition.

The reference count is incremented when the macro is called and decremented when the macro is exited. When this count goes to zero, the macro is removed from free space.

Macro arguments

Stored in the same linked block, but not in the symbol table. Repeats (two or more times) are also stored in the same way. The text blocks are removed when the macro exits or the repeat exits, since the reference count has gone to zero.

The addresses of the actual argument blocks are stored in a pushdown stack in order of generation.

Default arguments are stored in the same way, except that the list is in free core. The pointer to the default arg list is stored in the left half of the second word of the first block of the macro definition.

Macros

The macrobody is stored as is, except that dummy-arguments are replaced by special symbols.

ASCII 177 (RUBOUT) signals that the next character is a special character, as follows:

```
001  ;end of macro
002  ;end of dummy symbol
003  ;end of REPEAT
004  ;end of IRP or IRPC
005  ;RUBOUT
```

If the character is more than 5 and less than 100, it is illegal.

If the character is greater than or equal to 100, it is a dummy symbol; the value of the character is ANDed with 37 to get the dummy symbol number, and the corresponding pointer retrieved from the stack of actual arguments.

If the symbol was not specified (that is, has no pointer), and if the 40 bit is on, this symbol requires a created symbol, and one is created; otherwise the argument is ignored.

NOTE

> Verbose macros can use too much storage space.

Literals

Four-word block for each word generated

word 1: form word
word 2: relocation bits
word 3: code
word 4: pointer to next block

Form word is the word used for listing. This word is not checked when comparing literals, so that different forms producing the same code are classed as equal.

Relocation bits are 0, 1, or EXTERNAL pointers.

Pointer is the address of the zero word of the next block.

NOTE

Long literals slow assembly and use storage; they should be written as subroutines or inline code.

INDEX

MACRO ASSEMBLER
Reference Manual
AA-C780C-TB

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. Problems with software should be reported on a Software Performance Report (SPR) form. If you require a written reply and are eligible to receive one under SPR service, submit your comments on an 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 ______________

Please cut along this line.

Do Not Tear - Fold Here and Tape

digital

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

SOFTWARE PUBLICATIONS
200 FOREST STREET MR1-2/E37
MARLBOROUGH, MASSACHUSETTS 01752

Do Not Tear - Fold Here and Tape

Cut Along Dotted Line