CHAPTER 6

GENERAL CROSS-ASSEMBLER DIRECTIVES

A cross-assembler directive is placed in the operator field of a source line. Only one directive is allowed per source line. Each directive may have a blank operand field or one or more operands. Legal operands differ with each directive.

General cross-assembler directives are divided into the following categories:

1. Listing control
2. Function control
3. Data storage
4. Radix and numeric control
5. Location counter control
6. Terminator
7. Program sectioning and boundaries
8. Symbol control
9. Conditional assembly
10. File control

Each directive is described in its own section of this chapter (see Table 6-1 for an alphabetical listing of the directives and the associated section reference).

Directive	Function	Section Reference
.ASCII	Stores delimited strings as a sequence of the 8-bit ASCII code of their characters.	6.3.5
.ASCIZ	Same as .ASCII except the string is followed by a zero byte.	6.3.6
.ASECT	Declares absolute program section.	6.7.2
.BLKB	Allocates bytes of data storage.	6.5.3
.BLKL	Allocates long words of data storage.	6.5.3
.BLKW	Allocates words of data storage.	6.5.3
.BYTE	Stores successive bytes of data.	6.3.1
.DSABL	Disables specified cross-assembler functions.	6.2.1
.ENABL	Enables specified cross-assembler functions.	6.2.1
.END	Indicates end of source input.	6.6
.ENDC	Indicates end of conditional assembly block.	6.9.1
.EVEN	Ensures that current value of the location counter is even.	6.5.1
.GLOBL	Defines listed symbols as global.	6.8.1
.IDENT	Provides additional means of labeling an object module.	6.1.4
.IF	Assembles block if specified conditions are met.	6.9.1
.IFF	Assembles block if condition tests false.	6.9.2
.IFT	Assembles block if condition tests	6.9.2

	true.	
.IFTF	Assembles block regardless of whether condition tests true or false.	6.9.2
.IIF	Permits writing a one-line conditional assembly block.	6.9.3
.LIST	Increments listing count or lists certain types of code.	6.1.1
.LONG	Stores successive long words of data.	6.3.3
.NLIST	Decrements listing count or suppresses certain types of code.	6.1.1
.ODD	Ensures that the current value of the location counter is odd.	6.5.2
.PAGE	Starts a new listing page.	6.1.5
.PSECT	Declares names for program sections and establishes their attributes.	6.7.1
.RAD50	Generates data in Radix-50 packed format.	6.3.7
.RADIX	Changes radices throughout or in portions of the source program.	6.4.1.1
.REM	Delimits a section of comments.	6.1.6
.SBTTL	Produces a table of contents immediately preceding the assembly listing and puts subheadings on each page in the listing.	6.1.3
.TITLE	Assigns a name to the object module and puts headings on each page of the assembly listing.	6.1.2
.WORD	Generates successive words of data in the object module.	6.3.2

TABLE 6-1: General Cross-Assembler Directives

6.1 LISTING CONTROL DIRECTIVES

Listing control directives control the content, format, and pagination of all line printer assembly listing output. On the first line of each page, the cross assembler prints the following (from left to right):

1. Title of the object module, as established through the .TITLE directive (see section 6.1.2).

2. Cross-assembler version identification.

3. Day of the week.

4. Date.

5. Time of day.

6. Page number.

The second line of each assembly listing page contains the subtitle text specified in the last-encountered .SBTTL directive (see sectio 6.1.3).

In the teleprinter and line printer format, binary extensions for statements generating more than one word are listed horizontally.

6.1.1 .LIST and .NLIST Directives

Formats:

```
.LIST
.LIST  arg
.NLIST
.NLIST arg
```

where: arg represents one or more of the optional symbolic arguments defined in Table 6-2.

As indicated above, the listing control directives may be used without arguments, in which case the listing directives alter the listing level count. The listing level count is initialized to zero. At each occurrence of a .LIST directive, the listing level count is incremented; at each occurrence of an .NLIST directive, the listing level count is decremented. When the level count is negative, the listing is suppressed (unless the line contains an error)

Conversely, when the level count is greater than zero, the listing is generated regardless of the context of the line. Finally, when the count is zero, the line is either listed or suppressed, depending on the listing controls currently in effect for the program. The following macro definition employs the .LIST and .NLIST directives to selectively list portions of the macro body when the macro is expanded:

```
        .MACRO  LTEST               ;List test
; A-this line should list           ;Listing level count is 0.
        .NLIST                      ;Listing level count is -1.
; B-this line should not list
        .NLIST                      ;Listing level count is -2.
; C-this line should not list
        .LIST                       ;Listing level count is -1.
; D-this line should not list
        .LIST                       ;Listing level count is 0.
; E-this line should list           ;Listing level count is 0.
; F-this line should list           ;Listing level count is 0.
; G-this line should list           ;Listing level count is 0.
        .ENDM
          .
          .
          .
        .LIST   ME                  ;List macro expansion.
        LTEST                       ;Call the macro
; A-this line should list           ;Listing level count is 0.
; E-this line should list           ;Listing level count is 0.
; F-this line should list           ;Listing level count is 0.
; G-this line should list           ;Listing level count is 0.
```

Note that the line following line E will list because the listing level count remains 0. If a .LIST directive is placed at the beginning of a program, all macro expansions will be listed unless an .NLIST directive is encountered.

An important purpose of the level count is to allow macro expansions to be listed selectively and yet exit with the listing level count restored to the value existing prior to the macro call.

When used with arguments, the listing directives do not alter the listing level count. However, the .LIST and .NLIST directives can be used to override current listing control, as shown in the example below:

```
        .MACRO  XX
          .
          .
          .
        .LIST                    ;List next line.
X=.
        .NLIST                   ;Do not list remainder of macro
          .                      ;expansion.
          .
          .
        .ENDM

        .NLIST  ME               ;Do not list macro expansions.
        XX
X=.
```

The symbolic arguments allowed for use with the listing directives are described in Table 6-2. These arguments can be used singly or in combination with each other. If multiple arguments are specified in a listing directive, each argument must be separated by a comma, tab, or space. For any argument not specifically included in the control statement, the associated default assumption (List or No list) is applicable throughout the source program. The default assumptions for the listing control directives also appear in Table 6-2.

NOTE

> If the .NLIST arguments SEQ, LOC, BIN, and SRC are in effect at the same time (that is, if all four significant fields in the listing are to be suppressed), the printing of the resulting blank line is inhibited.

Argument	Default	Function
SEQ	List	Controls the listing of the sequential numbers assigned to the source lines. If this number field is suppressed through an .NLIST SEQ directive, the cross assembler generates a tab, effectively allocating blank space for the field. Thus, the positional relationships of the other fields in the listing remain undisturbed. During the assembly process, the cross assembler examines each source line for possible error conditions. For any line in error, the error code is printed preceding the number field. The cross assembler does not assign line numbers to files that have had such numbers assigned by other programs (an editor program, for instance).
LOC	List	Controls the listing of the current location counter field. Normally, this field is not suppressed. However, if it is suppressed through the .NLIST LOC directive, the cross assembler does not generate a tab, nor does it allocate space for the field, as is the case with the SEQ field described above. Thus, the suppression of the current location counter (LOC) field effectively left-justifies all subsequent fields (while preserving positional relationships) to the position normally occupied by the counter's field.

BIN	List	Controls the listing of generated binary code. If this field is suppressed through an .NLIST BIN directive, left-justification of the source code field occurs in the same manner described above for the LOC field.
BEX	List	Controls the listing of binary extensions (the locations and binary contents beyond those that will fit on the source statement line). This is a subset of the BIN argument.
SRC	List	Controls the listing of source lines.
COM	List	Controls the listing of comments. This is a subset of the SRC argument. The .NLIST COM directive reduces listing time and space when comments are not desired.
MD	List	Controls the listing of macro definitions and repeat range expansions.
MC	List	Controls the listing of macro calls and repeat range expansions.
ME	No list	Controls the listing of macro expansions.
MEB	No list	Controls the listing of macro expansion binary code. A .LIST MEB directive lists only those macro expansion statements that generate binary code. This is a subset of the ME argument.

CND	List	Controls the listing of unsatisfied conditional coding and associated .IF and .ENDC directives in the source program. A .NLIST CND directive lists only satisfied conditional coding.
LD	No list	Controls the listing of all listing directives having no arguments, in other words, the directives that alter the listing level count.
TOC	List	Controls the listing of the table of contents during assembly pass 1 (see section 6.1.3 describing the .SBTTL directive). This argument does not affect the printing of the full assembly listing during assembly pass 2.
SYM	List	Controls the listing of the symbol table resulting from the assembly of the source program.
TTM	No list	Sets the listing output format to teleprinter. The default is set to line printer format.

TABLE 6-2: Symbolic Arguments of Listing Control Directives

Any argument specified in a .LIST/.NLIST directive other than those listed in Table 6-2 causes the directive to be flagged with an error code (A) in the assembly listing.

The listing control options can also be specified at assembly time through switches included in the command string to the cross assembler (see the Emulogic Cross Assembler User´s Manual, section 2.3). The use of these switches overrides all corresponding listing control (.LIST or .NLIST) directives specified in the source program.

6.1.2 .TITLE Directive

Format:

```
     .TITLE string
```

where: string   represents an identifier of 1 or more Radix-50 characters which must begin with an alphabetic character. (See Appendix A.2 for a table of Radix-50 characters.)

The .TITLE directive assigns a name to the object module. The name assigned is the first six non-blank, Radix-50 characters following the .TITLE directive. All spaces and/or tabs up to the first non-space/non-tab character following the .TITLE directive are ignored by the cross assembler when evaluating the text string. Any characters beyond the first six are checked for ASCII legality, but they are not used as part of the object module name. For example, the directive

```
                         .TITLE PROGRAM TO PERFORM DAILY ACCOUNTING
```

causes the assembled object module to be named PROGRA. This 6-character name bears no relationship to the filename of the object module, as specified in the command string to the cross assembler. The name of an object module (specified in the .TITLE directive) appears in the load map produced at link time. This is also the module name which the Librarian will recognize.

If the .TITLE directive is not specified, the cross assembler assigns the default name .MAIN. to the object module. If more than one .TITLE directive is specified in the source program, the last .TITLE directive encountered during assembly pass 1 establishes the name for the entire object module.

If the .TITLE directive is specified without an object module name, or if the first non-space/non-tab character in the object module name is not a Radix-50 character, the directive is flagged with an error code (A) in the assembly listing.

6.1.3 .SBTTL Directive.

Format:

```
     .SBTTL string
```

where: string must begin with an alphanumeric character and represents an identifier of 1 or more printable ASCII characters.

The .SBTTL directive is used to produce a table of contents immediately preceding the assembly listing and to print the text following the .SBTTL directive on the second line of the header of each page in the listing. The subheading in the text will be listed until altered by a subsequent .SBTTL directive in the program. For example, the directive

```
.SBTTL Conditional assemblies
```

causes the text

```
Conditional assemblies
```

to be printed as the second line in the header of the assembly listing.

During assembly pass 1, a table of contents containing the line sequence number, the page number, and the text accompanying each .SBTTL directive is printed for the assembly listing. The listing of the table of contents is suppressed whenever an .NLIST TOC directive is encountered in the source program. An example of a table of contents listing is shown in Figure 6-1.

```
TABLE OF CONTENTS

   50-    1   .MTOUT - Single character output EMT
   51-    1   .MTRCTO - Reset CTRL/O EMT
   52-    1   .MTATCH - Attach to terminal EMT
   54-    1   .MTDTCH - Detach from a terminal EMT
   55-    1   .MTPRNT - Print message EMT
   56-    1   .MTSTAT - Return multi-terminal system status EMT
   57-    1   MTTIN - Single character input
   58-    1   MTTGET - Get a character from the ring buffer
   59-    1   TRRSET - Reset terminal status bits
   60-    1   MTTPUT - Single character output
   62-    1   MTRSET - Stop and detach all terminals attached to a job
   63-    1   ESCAPE SEQUENCE TEST SUBROUTINE
```

Figure 6-1: Assembly Listing Table of Contents

6.1.4 .IDENT Directive

Format:

```
.IDENT /string/
```

where: string represents a string of six or fewer Radix-50 characters which establish the program identification or version number. This string is included in the global symbol directory of the object module and is printed in the link map and Librarian listing.

/ / represent delimiting characters. These delimiters may be any paired printing characters, other than the equal sign (=), the left angle bracket (<), or the semicolon (;), as long as the delimiting character is not contained within the text string itself (see Note in section 6.3.4). If the delimiting characters do not match, or if an illegal delimiting character is used, the .IDENT directive is flagged with an error code (A) in the assembly listing.

In addition to the name assigned to the object module with the .TITLE directive (see section 6.1.2), the .IDENT directive allows the user to label the object module with the program version number.

An example of the .IDENT directive is shown below:

```
.IDENT /V01.00/
```

The character string is converted to Radix-50 representation and included in the global symbol directory of the object module. This character string also appears in the link map produced at link time and the Librarian directory listings.

When more than one .IDENT directive is encountered in a given program, the last such directive encountered establishes the character string which forms part of the object module identification.

The Linker allows only one .IDENT string in a program. The Linker uses the first .IDENT directive encountered during the first pass to establish the character string that will be identified with all of the object modules.

6.1.5 .PAGE Directive/Page Ejection

Format:

```
                    .PAGE
```

The .PAGE directive is used within the source program to perform a page eject at desired points in the listing. This directive takes no arguments and causes a skip to the top of the next page when encountered. It also causes the page number to be incremented and the line sequence counter to be cleared. The .PAGE directive does not appear in the listing.

When used within a macro definition, the .PAGE directive is ignored during the assembly of the macro definition. Rather, the page eject operation is performed as the macro itself is expanded. In this case, the page number is also incremented.

Page ejection is accomplished in three other ways:

1. After reaching a count of 58 lines in the listing, the cross assembler automatically performs a page eject to skip over page perforations on line printer paper and to formulate teleprinter output into pages. The page number is not changed.

2. A page eject is performed when a form-feed character is encountered. If the form-feed character appears within a macro definition, a page eject occurs during the assembly of the macro definition, but not during the expansion of the macro itself. A page eject resulting from the use of the form-feed character causes the page number to be incremented and the line sequence counter to be cleared.

3. A page eject is performed when encountering a new source file. In this case the page number is incremented and the line sequence count is reset.

6.1.6 .REM Directive/Begin Remark Lines

Format:

```
                    .REM comment-character
```

where: comment-character — represents a character that marks the end of the comment block when the character reoccurs.

The .REM directive allows a programmer to insert a block of comments into a cross-assembler source program without having to precede the comment lines with the comment character (;). The text between the specified delimiting characters is treated as comments. The comments may span any number of lines. For example:

```
.TITLE    Remark example
.REM      &
All the text that resides here is interpreted by
the cross assembler to be comment lines until
another ampersand character is found.  Any
character may be used in place of the ampersand.&
CLR       PC
.END
```

6.2 FUNCTION DIRECTIVES

The following function directives are included in a source program to invoke or inhibit certain cross-assembler functions and operations incidental to the assembly process itself.

6.2.1 .ENABL and .DSABL Directives

Formats:

```
.ENABL arg
.DSABL arg
```

where: arg — represents one or more of the optional symbolic arguments defined in Table 6-3.

Specifying any argument in an .ENABL/.DSABL directive other than those listed in Table 6-3 causes that directive to be flagged with an error code (A) in the assembly listing.

Argument	Default	Function
CRF	Enabled	Disabling this function inhibits the generation of cross-reference output. This function only has meaning if cross-reference output generation is specified in the command string.
LC	Disabled	Enabling this function causes the cross assembler to accept lowercase ASCII input instead of converting it to uppercase. If this function is not enabled, all text is converted to uppercase (see Figure 6-2).
LSB	Disabled	This argument permits the enabling or disabling of a local symbol block. Although a local symbol block is normally established by encountering a new symbolic label or a .PSECT directive in the source program, an .ENABL LSB directive establishes a new local symbol block which is not terminated until (1) another .ENABL LSB is encountered, or (2) another symbolic label or .PSECT directive is encountered following a paired .DSABL LSB directive. The basic function of this directive with regard to .PSECTs is limited to those instances where it is desirable to leave a program section temporarily to store data, followed by a return to the original program section. Attempts to define local symbols in an alternate program section are flagged with an error code (P) in the assembly listing.

GBL	Disabled	This argument, if enabled, causes the cross assembler to treat all undefined symbol references as global, allowing the Linker to resolve them. The default for this option is disabled, which causes the cross assembler to mark all undefined references in assembly pass 2 with a (U) error in the assembly listing.

TABLE 6-3: Symbolic Arguments of Function Control Directives

```
 1                                     .TITLE .ENABL/.DSABL
 2                                     .LIST TTM
 3                          ;+
 4                          ;ILLUSTRATE  .ENABL/.DSABLE LC
 5                          ;-
 6
 7                                     .ENABL LC ;STORE MACRO IN LOWER CASE
 8
 9                          .MACRO      TEXT $$$
10                                     .ASCII /this $$$ lowercase string/
11                          .ENDM
12
13                                     .LIST ME
14                                     .NLIST BEX
15
16 000000                              TEXT is ;Invoke macro in lower case
   000000   164                        .ASCII /this is a lower case string/
17
18                                     .DSABL LC ;Now disable lower case
19
20 000033                              TEXT WAS ;RE-INVOKE MACRO UPPERCASE
   000033   124                        .ASCII /THIS WAS A LOWERCASE STRING/
21
22       000001                        .END
```

Figure 6-2: Example of .ENABL and .DSABL Directives

6.3 DATA STORAGE DIRECTIVES

A wide range of data and data types can be generated with the following data-storage directives. ASCII conversion and radix-control cross-assembler directives are described in the following sections.

6.3.1 .BYTE Directive

Format:

```
          .BYTE     exp               ;Stores the binary value of the
                                      ;expression in the next byte.

          .BYTE     exp1,exp2,expn    ;Stores the binary values of the
                                      ;list of expressions in
                                      ;successive bytes.
```

```
where:      exp,    represent expressions that must be reduced to 8
            exp1,   bits of data or less.  Each expression will be
             .      read as a 16-bit word expression, the high-order
             .      byte to be truncated.  The high-order byte must
             .      be either all zeros or a truncation (T) error
            expn    results.  Multiple expressions must be separated by
                    commas.
```

The .BYTE directive is used to generate successive bytes of binary data in the object module.

Example:

```
SAM=5
.=410
           .BYTE ^D48,SAM       ;The value 30 (hex equivalent of 48
                                ;decimal) is stored in location 410.
                                ;The value 005 is stored in location
                                ;411.
```

The construction ^D in the first operand of the .BYTE directive above illustrates the use of a temporary radix-control operator. The function of such special unary operators is described in section 6.4.1.2.

At link time, it is likely that a relocatable expression will result in a value having more than eight bits, in which case the Linke issues a byte-relocation error for the object module in question. For example, the following statements create such a possibility:

```
          .BYTE     23                  ;Stores 23 in next byte.
A:
          .BYTE     A                   ;Relocatable value A will
                                        ;probably cause a
                                        ;byte-relocation error.
```

If an expression following the .BYTE directive is null, it is interpreted as a zero:

```
          .=420
          .BYTE     ,,,                 ;Zeros are stored in bytes
                                        ;420, 421, 422, and 423.
```

Note that in the above example, four bytes of storage result from the .BYTE directive. The three commas in the operand field represent an implicit declaration of four null values, each separated from the other by a comma. Hence, four bytes, each containing a value of zero, are reserved in the object module.

6.3.2 .WORD Directive

Formats:

```
          .WORD     exp                  ;Stores the binary equivalent
                                         ;of the expression in the next
                                         ;word.

          .WORD     exp1,exp2,expn       ;Stores the binary equivalents
                                         ;of the list of expressions
                                         ;in successive words.
```

```
where:     exp,    represent expressions that must reduce to 16 bits
           exp1,   of data or less. Multiple expressions must be
            .      separated by commas.
            .
            .
           expn
```

The .WORD directive is used to generate successive words of data in the object module.

Example:

```
SAL=0
.=500
          .WORD     123A,.+4,SAL        ;Stores the values 123A, 506,
                                        ;and 0 in words 500, 502, and
                                        ;504, respectively.
```

If an expression following the .WORD directive contains a null value, it is interpreted as a zero, as shown in the following example:

```
.=500
          .WORD     ,5,                 ;Stores the values 0, 5, and 0
                                        ;in location 500, 502, and
                                        ;504, respectively.
```

A statement with a blank operator field (one that contains a symbol other than a macro call, an instruction mnemonic, a cross-assembler directive, or a semicolon) is interpreted during assembly as an implicit .WORD directive, as shown in the example below:

```
.=440
LABEL:    100,LABEL               ;Stores the value 100 in location 440
                                  ;and the value 440 in location 442.
```

6.3.3 .LONG Directive

Formats:

```
          .LONG     exp                  ;Stores the binary equivalent
                                         ;of the expression in the next
                                         ;long word.

          .LONG     expl,exp2,expn       ;Stores the binary equivalents
                                         ;of the list of expressions in
                                         ;successive long words.
```

```
where:     exp,   represent expressions that must reduce to 32 bits
           expl,  of data or less.  Multiple expressions must be
            .     separated by commas.
            .
            .
           expn
```

The .LONG directive is used to generate successive long words of data in the object module.

Example:

```
SAL=0
.=500
        .LONG     123ABC,.+4,SAL    ;Stores the values 123ABC, 508,
                                    ;and 0 in long words 500, 504,
                                    ;and 508, respectively.
```

If an expression following the .LONG directive contains a null value, it is interpreted as a zero, as shown in the following example:

```
.=500
        .LONG     ,5,               ;Stores the values 0, 5, and 0
                                    ;in long word 500, 504, and 508,
                                    ;respectively.
```

6.3.4 ASCII Conversion Characters

The single quote (´) and the double quote (") characters are unary operators that can appear in any cross-assembler expression. Used in cross-assembler expressions, these characters cause a 16-bit expression value to be generated.

When the single quote is used, the cross assembler takes the next character in the expression and converts it from its 7-bit ASCII value to a 16-bit expression value. The high-order byte of the resulting expression value is always zero (0). The 16-bit value is then used as an absolute term within the expression. For example, the statement

```
LABEL:  .WORD  ´A
```

defines the following 16-bit expression value at LABEL:

```
00000000 01000001
           |
           ----Binary Value of ASCII A
```

Thus the expression ´A results in a value of 0041 (hex).

The single quote (´) character must not be followed by a carriage-return, null, RUBOUT, line-feed, or form-feed character; if it is, an error code (A) is generated in the assembly listing.

When the double quote is used, the cross assembler takes the next two characters in the expression and converts them to a 16-bit binary expression value from their 7-bit ASCII values. This 16-bit value is then used as an absolute term within the expression. For example, the statement

```
LABEL: .WORD "AB
```

defines the following 16-bit expression value at LABEL:

```
01000010 01000001
   |        |
   |        -----Binary Value of ASCII A
   |
   -----Binary Value of ASCII B
```

Thus the expression "AB results in a value of 4241 (hex).

The double quote (") character, like the single quote (´) character, must not be followed by a carriage-return, null, RUBOUT, line-feed, or form-feed character; if it is, an error code (A) is generated in the assembly listing.

The ASCII character set is listed in Appendix A.1.

6.3.5 .ASCII Directive

Format:

```
.ASCII /string 1/.../string n/
```

where: string is a string of printable ASCII characters. The vertical-tab, null, line-feed, RUBOUT, and all other non-printable ASCII characters, except carriage-return and form-feed, cause an error code (I) if used in an .ASCII string. The carriage-return and form-feed characters are flagged with an error code (A) because these characters end the scan of the line, preventing the cross-assembler from detecting the matching delimiter at the end of the character string.

/ /   represent delimiting characters. These delimiters may be any paired printing characters, other than the equal sign (=), the left angle bracket (<), or the semicolon (;), as long as the delimiting character is not contained within the text string itself. If the delimiting characters do not match, or if an illegal delimiting character is used, the .ASCII directive is flagged with an error code (A) in the assembly listing.

The .ASCII directive translates character strings into their 7-bit ASCII equivalents and stores them in the object module. A non-printing character can be expressed only by enclosing its equivalent octal value within angle brackets. Each set of angle brackets so used represents a single character. For example, in the following statement

```
.ASCII    <15>/ABC/<A+2>/DEF/<5><4>
```

the expressions <15>, <A+2>, <5>, and <4> represent the values of non-printing characters. Each bracketed expression must reduce to eight bits of absolute data or less.

Angle brackets can be embedded between delimiting characters in the character string, but angle brackets so used do not take on their usual significance as delimiters for non-printing characters. For example, the statement

```
.ASCII  /ABC<expression>DEF/
```

contains a single ASCII character string, and performs no evaluation of the embedded, bracketed expression. This use of the angle brackets is shown in the third example of the .ASCII directive below:

```
.ASCII  /HELLO/              ;Stores the binary
                             ;representation of the letters
                             ;HELLO in five consecutive
                             ;bytes.

.ASCII  /ABC/<15><12>/DEF/   ;Stores the binary
                             ;representation of the
                             ;characters A, B, C, carriage
                             ;return, line feed, D, E, F
                             ;in eight consecutive bytes.

.ASCII  /A<15>B/             ;Stores the binary
                             ;representation of the
                             ;characters A, <, 1, 5, >,
                                ;and B in six consecutive
                                ;bytes.
```

NOTE

The semicolon (;) and equal sign (=) can be used as delimiting characters in the string, but care must be exercised in so doing because of their significance as a comment indicator and assignment operator, respectively, as illustrated in the examples below:

```
.ASCII      ;ABC;/DEF/      ;Stores the binary
                            ;representation of
                            ;the characters
                            ;A, B, C, D, E, and
                            ;F in six
                            ;consecutive bytes;
                            ;not recommended
                            ;practice.

.ASCII /ABC/;DEF;           ;Stores the binary
                            ;representations of
                            ;the characters A,
                            ;B, and C in three
                            ;consecutive bytes;
                            ;the characters D,
                            ;E, F, and ; are
                            ;treated as a
                            ;comment.

.ASCII /ABC/=DEF=           ;Stores the binary
                            ;representation of
                            ;the characters A,
                            ;B, C, D, E, and
                            ;F in six
                            ;consecutive bytes;
                            ;not recommended
                            ;practice.
```

An equal sign is treated as an assignment operator when it appears as the first character in the ASCII string, as illustrated by the following example:

```
.ASCII =DEF=                 ;The direct
                             ;assignment
                             ;operation
                             ;.ASCII=DEF is
                             ;performed, and a
                             ;syntax error (Q)
                             ;is generated upon
                             ;encountering the
                             ;second = sign.
```

6.3.6 .ASCIZ Directive

Format:

```
        .ASCIZ  /string 1/.../string n/
```

where: string is a string of printable ASCII characters. The vertical-tab, null, line-feed, RUBOUT, and all other non-printable ASCII characters, except carriage-return and form-feed, cause an error code (I) if used in an .ASCIZ string. The carriage-return and form-feed characters are flagged with an error code (A) because they end the scan of the line, preventing the cross assembler from detecting the matching delimiter.

/ / represent delimiting characters. These delimiters may be any paired printing characters, other than the equal sign (=), the left angle bracket (<), or the semicolon (;) (see Note in section 6.3.5), as long as the delimiting character is not contained within the text string itself. If the delimiting characters do not match or if an illegal delimiting character is used, the .ASCIZ directive is flagged with an error code (A) in the assembly listing.

The .ASCIZ directive is similar to the .ASCII directive described above, except that a zero byte is automatically inserted as the final character of the string. Thus, when a list or text string has been created with an .ASCIZ directive, a search for the null character in the last byte can effectively determine the end of the string.

6.3.7 .RAD50 Directive

Format:

```
.RAD50 /string 1/.../string n/
```

where: string represents a series of characters to be packed. The string must consist of the characters A through Z, 0 through 9, dollar sign ($), period (.) and space ( ). An illegal printing character causes an error flag (Q) to be printed in the assembly listing.

If fewer than three characters are to be packed, the string is packed left-justified within the word, and trailing spaces are assumed.

As with the .ASCII directive (described in section 5.3.4), the vertical-tab, null, line-feed, RUBOUT, and all other non-printing characters, except carriage-return and form-feed, cause an error code (I) if used in a .RAD50 string. The carriage-return and form-feed characters result in an error code (A) because these characters end the scan of the line, preventing the cross assembler from detecting the matching delimiter.

/ / represent delimiting characters. These delimiters may be any paired printing characters, other than the equal sign (=), the left angle bracket (<), or the semicolon (;) (see Note in section 6.3.5), provided that the delimiting character is not contained within the text string itself. If the delimiting characters do not match or if an illegal delimiting character is used, the .RAD50 directive is flagged with an error code (A) in the assembly listing.

The .RAD50 directive allows the user to generate data in Radix-50 packed format. Radix-50 form allows three characters to be packed into sixteen bits (one word); therefore, any 6-character symbol can be stored in two consecutive words. Examples of .RAD50 directives are shown below:

```
        .RAD50    /ABC/       ;Packs ABC into one word.
        .RAD50    /AB/        ;Packs AB (SPACE) into one word.
        .RAD50    /ABCD/      ;Packs ABC into first word and
                              ;D (SPACE) (SPACE) into second word.
        .RAD50    /ABCDEF/    ;Packs ABC into first word, DEF into
                              ;second word.
```

Each character is translated into its Radix-50 equivalent, as indicated in the following table:

Character	Radix-50 Octal Equivalent
(space)	0
A-Z	1-32
$	33
.	34
(undefined)	35
0-9	36-47

The Radix-50 equivalents for characters 1 through 3 (C1,C2,C3) are combined as follows:

Radix-50 Value = ((C1*50)+C2)*50+C3

For example:

Radix-50 Value of ABC = ((1*50)+2)*50+3 = 3223(8)

The Radix-50 character set is listed in Appendix A.2.

Angle brackets (<>) must be used in the .RAD50 directive whenever special codes are to be inserted in the text string, as shown in the example below:

```
        .RAD50    /AB/<35>    ;Stores 3255 in one word

CHR1=1
CHR2=2
CHR3=3
              .
              .
              .
        .RAD50    <CHR1><CHR2><CHR3> ;Equivalent to .RAD50 /ABC/
```

6.3.8 Temporary Radix-50 Control Operator

Format:

```
                    ^Rccc
```

where: ccc represents a maximum of three characters to be converted to a 16-bit Radix-50 value. If more than three characters are specified, any following the third character are ignored. If fewer than three are specified, it is assumed that the trailing characters are blanks.

The ^R operator specifies that an argument is to be converted to Radix-50 format. This allows up to three characters to be stored in one word. The following example shows how the ^R operator might be used to pack a 3-character file type specifier (MAC) into a single 16-bit word:

```
FILEXT    .WORD     ^RMAC        ;Defines RAD50 MAC as file
                                 ;extension
```

6.4 RADIX AND NUMERIC CONTROL FACILITIES

6.4.1 Radix Control and Unary Control Operators

Any numeric or expression value in a cross-assembler source program is read as an octal value by default. Occasionally, however, an alternate radix would be useful. By using the cross-assembler facilities described below, a programmer may declare a radix to affect a term or an entire program.

NOTE

> When two or more unary operators appear together, modifying the same term, the operators are applied to the term from right to left.

6.4.1.1 .RADIX Directive

Format:

```
     .RADIX n
```

where: n represents one of the two radices: 8 or 16. Any value other than null or one of the two acceptable radices will cause an error code (A) in the assembly listing. If the argument n is not specified, the octal default radix is assumed. The argument (n) is always read as a decimal value.

Numbers used in a cross-assembler source program are initially considered to be octal values; however, with the .RADIX directive you can declare alternate radices applicable throughout the source program or within specific portions of the program.

Any alternate radix declared in the source program through the .RADIX directive remains in effect until altered by the occurrence of another such directive, for example:

```
          .RADIX 16              ;Begins a section of code having a
                                 ;hex radix.
            .
            .
          .
        .RADIX                  ;Reverts to octal radix.
```

Please note that when .RADIX 16 is in effect, any numeric value whose first character is A-F must be preceded by a zero.

In general, macro definitions should not contain or rely on radix settings established with the .RADIX directive. Rather, temporary radix control operators should be used within a macro definition. Where a possible radix conflict exists within a macro definition or source program, it is recommended that the user specify numeric or expression values using the temporary radix control operators described below.

6.4.1.2 Temporary Radix Control Operators

Formats:

```
                  ^Dn  (decimal)
                  ^On  (octal)
                  ^Bn  (binary)
                  ^Hn  (hexadecimal)  (if the first character of n
                                       is A-F, precede n with a zero.)
```

These four unary operators allow the user to establish an alternate radix for a single term. An alternate is useful because after you have specified a radix for a section of code, usually hexadecimal, you may discover a number of cases where an alternate radix is more convenient or desirable (particularly within macro definitions). Creating a mask word (used to check bit status), for example, might best be accomplished through the use of a binary radix.

Thus an alternate radix can be declared temporarily to meet a localized requirement in the source program. The temporary radix control operator may be used any time regardless of the radix in effect or other radix declarations within the program. Because the operator affects only the term immediately following it, it may be used anywhere a numeric value is legal. The term (or expression) associated with the temporary radix control operator will be evaluated during assembly as a 16-bit entity.

The expressions below are representative of the methods of specifying temporary radix control operators:

```
                  ^D123       Decimal Radix
                  ^H0ABC      Hexadecimal Radix
```

```
^B 00001101 Binary Radix
^O<A+13>    Octal Radix
```

The up-arrow and the radix control operator may not be separated, but the radix control operator and the following term or expression can be separated by spaces or tabs for legibility or formatting purposes. A multi-element term or expression that is to be interpreted in an alternate radix should be enclosed within angle brackets, as shown in the last of the four temporary radix control expressions above.

The following example also illustrates the use of angle brackets to delimit an expression that is to be interpreted in an alternate radix. When using the temporary radix control operator, only numeric values are affected. Any symbols used with the operator will be evaluated with respect to the radix in effect at their declaration.

```
                  .RADIX 16
A=10
                  .WORD ^D<A+10>*^D10
```

When the temporary radix expression in the .WORD directive above is evaluated, it yields the following equivalent statement:

```
                  .WORD ^D260
```

The cross assembler allows a temporary radix change to decimal by specifying a number, immediately followed by a decimal point (.), as shown below:

```
100.          Equivalent to 64 (hex)
1376.         Equivalent to 560 (hex)
128.          Equivalent to 80 (hex)
```

The above expression forms are equivalent in function to:

```
^D100
^D1376
^D128
```

6.5 LOCATION COUNTER CONTROL DIRECTIVES

The directives used in controlling the value of the current location counter and in reserving storage space in the object program are described in the following sections.

Several cross-assembler statements (listed below) may cause an odd number of bytes to be allocated:

1. .BYTE directive
2. .BLKB directive
3. .ASCII or .ASCIZ directive
4. .ODD directive
5. A direct assignment statement of the form .=.+expression, which results in the assignment of an odd address value.

In cases that yield an odd address value and for chips that must start instructions on even boundaries, the next word-boundaries instruction automatically forces the location counter to an even value, but that instruction is flagged with an error code (B) in the assembly listing.

6.5.1 .EVEN Directive

Format:

```
                    .EVEN
```

The .EVEN directive ensures that the current location counter contains an even value by adding 1 if the current value is odd. If the current location counter is already even, no action is taken. Any operands following an .EVEN directive are flagged with an error code (Q) in the assembly listing.

The .EVEN directive is used as follows:

```
          .ASCIZ    /This is a test/
          .EVEN               ;Ensures that the next statement will
                              ;begin on a word boundary.
          .WORD     XYZ
```

6.5.2 .ODD Directive

Format:

```
                    .ODD
```

The .ODD directive ensures that the current location counter contains an odd value by adding 1 if the current value is even. If the current location counter is already odd, no action is taken. Any operands following an .ODD directive are flagged with an error code (Q) in the assembly listing.

6.5.3 .BLKB, .BLKW and .BLKL Directives

NOTE

> The .BLKL directive is supported only for microprocessors that handle 32-bit data.

Formats:

```
                    .BLKB exp
                    .BLKW exp
                    .BLKL exp
```

where: exp represents the specified number of bytes, words or long words (32 bits) to be reserved in the object program. Any expression defined at assembly time that reduces to an absolute value is legal. If the expression specified in either of these directives is not an absolute value, the statement is flagged with an error code (A) in the assembly listing. Furthermore, if the expression contains a forward reference (a reference to a symbol that is not previously defined), the cross assembler generates incorrect object file code and may cause statements following the .BLKB/.BLKW/.BLKL directive to be flagged with phase (P) errors. These directives should not be used without arguments. However, if no argument is present, a default value of 1 is assumed.

The .BLKB directive reserves byte blocks in the object module; the

.BLKW directive reserves word blocks, and the .BLKL directive reserves long word blocks. The following example illustrates the use of the .BLKB, .BLKW, and .BLKL directives.

```
                    ;+
                    ; Illustrate use of .BLKB, .BLKW and .BLKL
                    ;-
                              .PSECT  IMPURE,D,GBL,RW

0000                COUNT:    .BLKW   1          ;Character counter

0002                MESSAG:   .BLKB   50.        ;Message text buffer

0052                CHRSAV:   .BLKB              ;Saved character

0053                LBUF:     .BLKL              ;Long word buffer

0057                MSGPTR:   .BLKW              ;Message buffer ptr.
```

The .BLKB directive in a source program has the same effect as the following statement:

```
                    .=.+expression
```

which causes the value of the expression to be added to the current value of the location counter. The .BLKB directive, however, is easier to interpret in the context of the source code in which it appears and is therefore recommended.

6.6 TERMINATING DIRECTIVE: .END DIRECTIVE

Format:

```
.END [exp]
```

where: exp represents an optional expression value which, if present, indicates the program-entry point, which is the transfer address where the program begins.

When the cross-assembler encounters a valid occurrence of the .END directive, it terminates the current assembly pass. Any text beyond this point in the current source file, or in additional source files identified in the command line, will be ignored.

When creating an image consisting of several object modules, only one object module may be terminated with an .END [exp] statement (where exp is the starting address). All other object modules must be terminated with an .END statement (where .END has no argument); otherwise, an error message will be issued at link time. If no starting address is specified in any of the object modules, image execution will begin at location 1 of the image and immediately fault because of an odd addressing error.

The .END statement must not be used within a macro expansion or a conditional assembly block; if it is so used, it is flagged with an error code (O) in the assembly listing. The .END statement may be used, however, in an immediate conditional statement (see section 6.9.3).

If the source program input is not terminated with an .END directive, an error code (E) results in the assembly listing.

6.7 PROGRAM SECTIONING DIRECTIVES

The cross-assembler program sectioning directives are used to establish program section attributes essential to linking and to declare names for program sections (p-sects).

6.7.1 .PSECT Directive

Format:

```
        .PSECT name,arg1,arg2,...argn
```

where: name represents the symbolic name of the program section, as described in Table 6-4.

comma represents any legal separator (comma, tab and/or space).

arg1,
arg2,
.
.
.
argn represent one or more of the legal symbolic arguments defined for use with the .PSECT directive, as described in Table 6-4. The slash separating each pair of symbolic arguments listed in the table indicates that one or the other, but not both, may be specified. Multiple arguments must be separated by a legal separating character. Any symbolic argument specified in the .PSECT directive other than those listed in Table 6-4 will cause that statement to be flagged with an error code (A) in the assembly listing.

Argument	Default	Meaning
NAME	Blank	Must begin with an alphabetic character. Establishes the program section name, which is specified as one-to-six Radix-50 characters. If this argument is omitted, a comma must appear in place of the name parameter if other arguments are to follow. The Radix-50 character set is listed in Appendix A.
RO/RW	RW	Defines which type of access is permitted to the program section: RO=Read-Only Access RW=Read/Write Access
I/D	I	Defines the program section as containing either instructions (I) or data (D). These attributes allow the Linker to differentiate global symbols that are entry point instructions (I) from those that are data values (D).
ABS/REL	REL	Defines the relocatability attribute of the program section: ABS=Absolute (non-relocatable). The ABS argument causes the Linker to treat the program section as an absolute module; therefore, no relocation is required. The program section is assembled and loaded, starting at absolute virtual address 0. REL=Relocatable. The REL argument causes the Linker to treat the program section as a relocatable module and a relocation bias is added to all location references within the program section making the references absolute.

TABLE 6-4: Symbolic Arguments of .PSECT Directive

The only argument in the .PSECT directive that is position dependent is NAME. If it is omitted, a comma must be used in its place. For example, the directive:

```
        .PSECT ,ABS
```

shows a .PSECT directive with a blank name argument and the ABS argument. Default values (see Table 6-4) are assumed for all other unspecified arguments.

The .PSECT directive allows a user to create program sections. All references to one program section are concatenated to determine the total memory space available for the program section. In declaring the program sections (p-sects), you may declare the attributes of the program sections. This allows you to control memory allocation and at the same time increases program modularity.

The cross assembler provides for 256(10) program sections, as listed below:

1. One default absolute program section (. ABS.)

2. One unnamed relocatable program section.

3. Two-hundred-fifty-four named program sections.

For each program section specified or implied, the cross assembler maintains the following information:

1. Program section name,

2. Contents of the current location counter,

3. Maximum location counter value encountered,

4. Program section attributes (described in Table 6-4 above).

The first statement of a source program is always an implied .PSECT directive; this causes the cross assembler to begin assembling source statements at relocatable zero of the unnamed program section.

The first occurrence of a .PSECT directive with a given name assumes that the current location counter is set at relocatable zero. The scope of this directive then extends until a directive declaring a different program section is specified. Subsequent .PSECT directives cause assembly to resume where the named section previously ended. For example:

```
        .PSECT              ;Declares unnamed relocatable program
```

```
A:          .WORD   0             ;section assembled at relocatable
B:          .WORD   0             ;addresses 0 through 5.
C:          .WORD   0
            .PSECT ALPHA          ;Declares relocatable program section
X:          .WORD   0             ;named ALPHA assembled at relocatable
Y:          .WORD   0             ;addresses 0 through 3.
            .PSECT                ;Returns to unnamed relocatable
D:          .WORD   0             ;program section and continues assem-
                                  ;bly at relocatable address 6.
```

A given program section may be defined completely upon encountering its first .PSECT directive. Thereafter, the section can be referenced by specifying its name only, or by completely respecifying its attributes. For example, a program section can be declared through the directive:

```
            .PSECT ALPHA,ABS
```

and later referenced through the equivalent directive:

```
            .PSECT ALPHA
```

which requires no arguments. If arguments are specified, they must be identical to the ones previously declared for the program section. If the arguments differ, the arguments of the first .PSECT will remain in effect, and an error code (A) will be generated as a warning.

By maintaining separate location counters for each program section, the cross assembler allows you to write statements that are not physically sequential but that can be loaded sequentially following assembly, as shown in the following example.

```
         .PSECT     SEC1,REL     ;Start relocatable program section
A:       .WORD      0            ;named SEC1 assembled starting at
B:       .WORD      0            ;relocatable address 0.
C:       .WORD      0
ST:      operator   A            ;Assemble relocatable code
         operator   B
         operator   C
         .PSECT     SECA,ABS     ;Start an absolute program section
                                 ;named SECA. Assemble code at
         .WORD      .+2,A        ;absolute addresses 0 through 3.
         .PSECT     SEC1         ;Resume relocatable program section
         operator   A            ;SEC1.
         operator   ST
```

All labels in an absolute program section are absolute; likewise, all labels in a relocatable program section are relocatable. The current location counter symbol (.) is relocatable or absolute when referenced in a relocatable or absolute program section, respectively.

Any labels appearing on a line containing a .PSECT or .ASECT directive are assigned the value of the current location counter before the .PSECT (or other) directive takes effect. Thus, if the first statement of a program is

```
A:         .PSECT     SECB,REL
```

the label A is assigned to the address of the current program section rather than relocatable address zero of the new program section SECB.

Since it is not known during assembly where relocatable program sections will be loaded, all references to relocatable program sections are assembled as references relative to the base of the referenced section.

In the following example, references to the symbols X and Y are translated into references relative to the base of the relocatabl program section named SEN.

```
           .PSECT     ENT,ABS
.=.+1000
A:         .WORD      X            ;Assembled as base of
                                   ;relocatable section + 2
           .WORD      Y            ;Assembled as base of
                                   ;relocatable section + 4
           .PSECT     SEN,REL
           .WORD      A            ;Assembled as 1000.
Y:         .WORD      0
X:         .WORD      0
```

NOTE

> In the preceding example, using a constant in conjunction with the current location counter symbol (.) in the form .=1000 would result in an error, because constants are always absolute and are always associated with the program's .ASECT (. ABS.). If the form .=1000 were used, a program section incompatibility would be detected. See section 3.5 for a discussion of the current location counter.

6.7.2 .ASECT Directive

Format:

```
                  .ASECT
```

The cross assembler will accept .ASECT directives, but assembles them as though they were .PSECT directives with the default attributes listed in Table 6-5.

Attribute	.ASECT Default Value
Name	. ABS.
ACCESS	RW
Type	I
Relocation	ABS

TABLE 6-5: Program Section Default Values

6.8 SYMBOL CONTROL DIRECTIVES

The symbol control directives are used to set the type of a given symbol.

6.8.1 .GLOBL Directive

Format:

```
                .GLOBL sym1,sym2,...symn
```

where: sym1, sym2,... symn — represent legal symbolic names. When multiple symbols are specified, they are separated by any legal separator (comma, space, and/or tab).

A .GLOBL directive may also embody a label field and/or a comment field.

The .GLOBL directive is provided to define (and thus provide linkage to) symbols not otherwise defined as global symbols within a module In defining global symbols, the directive .GLOBL A,B,C is similar to:

```
A==expression        A::
B==expression   or   B::
C==expression        C::
```

Because object modules may be linked by global symbols, these symbols are vital to a program. The role of global symbols, describing the processing of a program from assembly to linking, is subsequently discussed.

In assembling a source module, the cross assembler produces a relocatable object module and a listing file containing the assembly listing and symbol table. The Linker joins separately assembled object modules into a single executable image. During linking, object modules are relocated relative to the base of the module and linked by global symbols. Because these symbols will be referenced by other program modules, they must be singled out as global symbols in the defining modules.

All internal symbols appearing within a given program must be defined at the end of assembly pass 1 or they will be assumed to be default global references. Refer to section 6.2.1 for a description of enabling/disabling of global references.

In the following example (in which chip-specific mnemonics have bee used for the LSI-11 microprocessor), A and B are entry-point symbols. The symbol A has been explicitly defined as a global symbol by means of the .GLOBL directive, and the symbol B has been explicitly defined as a global symbol by means of the double colon (::). Since the symbol C is not defined within the current assembly, it is an external (global) reference if .ENABL GBL is in effect.

```
;
; Define a subroutine with 2 entry points which calls an
; external subroutine
;
          .PSECT                        ;Declare the unnamed program
                                        ;section.
          .ENABL GBL
          .GLOBL         A              ;Define A as a global symbol.
A:        MOV            @(R5)=,R0      ;Define entry point A.
          MOV            #X,R1
X:        JSR            PC,C           ;Reference external subroutine
                                        ;C.
          RTS            R5             ;Exit.
B::       MOV            (R5)=,R1       ;Define entry point B.
          CLR            R2
          BR             X
```

External symbols can appear in the operand field of an instruction or cross-assembler directive as a direct reference, as shown in the examples below:

```
          CLR            EXT
          .WORD          EXT
```

External symbols may also appear as a term within an expression, as shown below:

```
          CLR            EXT+A
          .WORD          EXT-2
```

An undefined external symbol cannot be used in the evaluation of a direct assignment statement or as an argument in a conditional assembly directive (see sections 3.3, 6.9.1 and 6.9.3).

6.9 CONDITIONAL ASSEMBLY DIRECTIVES

Conditional assembly directives allow you to include or exclude blocks of source code during the assembly process, based on the evaluation of stated condition tests within the body of the program.

6.9.1 Conditional Assembly Block Directives

Format:

```
        .IF    cond,argument(s)        ;Start conditional assembly
        .                              ;block.
        .
        .
        .
        range                          ;Range of conditional assembly
        .                              ;block.
        .
        .
        .
        .ENDC                          ;End of conditional assembly
        .                              ;block.
```

where: cond — represents a specified condition that must be met if the block is to be included in the assembly. The conditions that may be tested by the conditional assembly directives are defined in Table 6-6.

comma — represents any legal separator (comma, space, and/or tab).

argument(s) — represent(s) the symbolic argument(s) or expression(s) of the specified conditional test. These arguments are thus a function of the condition to be tested (see Table 6-6).

range — represents the body of code that is either included in the assembly, or excluded, depending upon whether the condition is met.

.ENDC — terminates the conditional assembly block. This directive must be present to end the conditional assembly block.

A condition test other than those listed in Table 6-6, an illega argument, or a null argument specified in an .IF directive will cause that line to be flagged with an error code (A) in the assembly listing.

CONDITIONS			
Positive	Complement	Arguments	Assemble Block If:
EQ Z	NE NZ	Expression	Expression is equal to 0 (or not equal to 0).
GT G	LE	Expression	Expression is greater than 0 (or less than or equal to 0).
LT L	GE	Expression	Expression is less than 0 (or greater than or equal to 0).
DF	NDF	Symbolic argument	Symbol is defined (or not defined).
B	NB	Macro-type argument	Argument is blank (or non-blank).
IDN	DIF	Two macro-type arguments	Arguments are identical (or different). The .IF IDN/.IF DIF conditional directives are not alphabetically case sensitive by default. The user may enable these directives to be case sensitive by using the .ENABL option (.ENABL LCM).

TABLE 6-6: Legal Condition Tests for Conditional Assembly Directives

NOTE

A macro-type argument (which is a form of symbolic argument), as shown below, is enclosed within angle brackets or denoted with an up-arrow construction (as described in section 7.3).

```
<A,B,C>
 ^/124/
```

An example of a conditional assembly directive follows:

```
          .IF EQ  ALPHA+1      ;Assemble block if ALPHA+1=0
            .
            .

          .ENDC
```

The two operators & and ! have special meaning within DF and NDF conditions, in that they are allowed in grouping symbolic arguments.

&	Logical AND operator
!	Logical inclusive OR operator

For example, the conditional assembly statement:

```
          .IF DF SYM1 & SYM2
            .
            .
            .
          .ENDC
```

results in the assembly of the conditional block if the symbols SYM1 and SYM2 are both defined.

Nested conditional directives take the form:

```
          Conditional Assembly Directive
          Conditional Assembly Directive
             .
             .
             .
          .ENDC
          .ENDC
```

For example, the following conditional directives

```
          .IF DF SYM1
          .IF DF SYM2
            .
            .
            .
          .ENDC
          .ENDC
```

can govern whether assembly is to occur. In the example above, if the outermost condition is unsatisfied, no deeper level of evaluation of nested conditional statements within the program occurs.

Each conditional assembly block must be terminated with an .ENDC directive. An .ENDC directive encountered outside a conditional assembly block is flagged with an error code (O) in the assembly listing.

The cross assembler permits a nesting depth of 16(10) conditional assembly levels. Any statement that attempts to exceed this nesting level depth is flagged with an error code (O) in the assembly listing.

6.9.2 Subconditional Assembly Block Directives

Formats:

```
.IFF
.IFT
.IFTF
```

Subconditional directives may be placed within conditional assembly blocks to indicate:

1. The assembly of an alternate body of code when the condition of the block tests false.
2. The assembly of a non-contiguous body of code within the conditional assembly block, depending upon the result of the conditional test in entering the block.
3. The unconditional assembly of a body of code within a conditional assembly block.

The subconditional directives are described in detail in Table 6-7, below. If a subconditional directive appears outside a conditional assembly block, an error code (O) is generated in the assembly listing.

Subconditional Directive	Function
.IFF	If the condition tested upon entering the conditional assembly block is false, the code following this directive, and continuing up to the next occurrence of a subconditional directive or to the end of the conditional assembly block, is to be included in the program.
.IFT	If the condition tested upon entering the conditional assembly block is true, the code following this directive, and continuing up to the next occurrence of a subconditional directive or to the end of the conditional assembly block, is to be included in the program.
.IFTF	The code following this directive, and continuing up to the next occurrence of a subconditional directive or to the end of the conditional assembly block, is to be included in the program, regardless of the result of the condition tested upon entering the conditional assembly block.

TABLE 6-7: Subconditional Assembly Block Directives

The implied argument of a subconditional directive is the condition test specified upon entering the conditional assembly block, as reflected by the initial directive in the conditional coding examples below. Conditional or subconditional directives in nested conditional assembly blocks are not evaluated if the previous (or outer) condition in the block is not satisfied. Examples 3 and 4 below illustrate nested directives that are not evaluated because of previous unsatisfied conditional coding.

EXAMPLE 1: Assume that symbol SYM is defined.

```
.IF DF   SYM            ;Tests TRUE, SYM is defined. Assemble
  .                     ;the following code.
  .
  .
.IFF                    ;Tests FALSE. SYM is defined. Do not
  .                     ;assemble the following code.
  .
  .
.IFT                    ;Tests TRUE. SYM is defined. Assem-
  .                     ;ble the following code.
  .
  .
.IFTF                   ;Assemble following code uncondition-
  .                     ;ally.
  .
  .
  .
.IFT                    ;Tests TRUE. SYM is defined. Assem-
  .                     ;ble remainder of conditional assem-
  .                     ;bly block.
  .
.ENDC
```

EXAMPLE 2: Assume that symbol X is defined and that symbol Y is not defined.

```
.IF DF   X              ;Tests TRUE, symbol X is defined.
.IF DF   Y              ;Tests FALSE, symbol Y is not defined.
.IFF                    ;Tests TRUE, symbol Y is not defined,
  .                     ;assemble the following code.
  .
  .
.IFT                    ;Tests FALSE, symbol Y is not defined.
  .                     ;Do not assemble the following code.
  .
  .
.ENDC
```

```
          .ENDC
```

EXAMPLE 3: Assume that symbol X is not defined and that symbol Y is defined.

```
          .IF DF  X             ;Tests FALSE. Symbol X is not defined.
                                ;Do not assemble the following code.
          .IF DF  Y             ;Nested conditional directive is not
            .                   ;evaluated.
            .
            .
          .IFF                  ;Nested subconditional directive is
            .                   ;not evaluated.
            .
            .
          .IFT                  ;Nested subconditional directive is
            .                   ;not evaluated.
            .
            .
          .ENDC
          .ENDC
```

6.9.3 Immediate Conditional Assembly Directive

Format:

```
                    .IIF cond,arg,statement
```

where:

cond — represents one of the legal condition tests defined for conditional assembly blocks in Table 6-6.

comma — represents any legal separator (comma, space, and/or tab).

arg — represents the argument associated with the immediate conditional directive; an expression, symbolic argument, or macro-type argument, as described in Table 6-6.

comma — represents the separator between the conditional argument and the statement field. If the preceding argument is an expression, then a comma must be used; otherwise, a comma, space and/or tab may be used.

statement represents the specified statement to be assembled if the condition is satisfied.

An immediate conditional assembly directive provides a means for writing a 1-line conditional assembly block. The use of this directive requires no terminating .ENDC statement and the condition to be tested is completely expressed within the line containing the directive.

For example, the immediate conditional statement:

```
                    .IIF   DF FOO,BEQ ALPHA
```

generates the code

```
                    BEQ    ALPHA
```

if the symbol FOO is defined within the source program.

As with the .IF directive, a condition test other than one of those listed in Table 6-6, an illegal argument, or a null argument specified in an .IIF directive results in an error code (A) in the assembly listing.

CHAPTER 7

MACRO DIRECTIVES

7.1 DEFINING MACROS

Macro directives provide the means to manipulate the macro expansions. Only one directive is allowed per source line. Each directive may have a blank operand field or one or more operands. Legal operands differ with each directive. The macros and their associated directives are detailed in this chapter.

By using macros, a programmer can use a single line to insert a sequence of lines into a source program.

A macro definition is headed by a .MACRO directive (see section 7.1.1) followed by the source lines. The source lines may optionally contain dummy arguments. If such arguments are used, each one is listed in the .MACRO directive.

A macro call (see section 7.3) is the statement used by the programmer to call the macro into the source program. The macro call consists of the macro name followed by the real arguments needed to replace any dummy arguments used in the macro.

Macro expansion is the insertion of the macro source lines into the main program. Included in this insertion is the replacement of the dummy arguments by the real arguments.

7.1.1 .MACRO Directive

Format:

```
[label:]  .MACRO name, dummy argument list
```

where: label — represents an optional statement label.

name — represents the user-assigned symbolic name of the macro. This name may be any legal symbol and may be used as a label elsewhere.

comma — represents any legal separator (comma, space, and/or tab).

dummy argument list — represents a number of legal symbols (see section 3.2.2) that may appear anywhere in the body of the macro definition, even as a label. These dummy symbols can be used elsewhere in the program with no conflict of definition. Multiple dummy arguments specified in this directive may be separated by any legal separator. The detection of a duplicate or an illegal symbol in a dummy argument list terminates the scan and causes an error code (A) to be generated.

Example:

```
        .MACRO ABS  A,B       ;Defines macro ABS with two
                              ;arguments, A and B.
```

The first statement of a macro definition must be a .MACRO directive.

NOTE

> Although it is legal for a label to appear on a .MACRO directive, this practice is discouraged, especially in the case of nested macro definitions, because invalid labels or labels constructed with the concatenation character will cause the macro directive to be ignored. This may result in improper termination of the macro definition.
>
> This NOTE also applies to .IRP, .IRPC, and .REPT (to be discussed).

7.1.2 .ENDM Directive

Format:

```
            .ENDM [name]
```

where: name — represents an optional argument specifying the name of the macro being terminated by the directive.

Example:

```
          .ENDM                     ;Terminates the current
                                    ;macro definition.

          .ENDM  ABS                ;Terminates the current
                                    ;macro definition named ABS.
```

If specified, the macro name in the .ENDM statement must match the name specified in the corresponding .MACRO directive. Otherwise, the statement is flagged with an error code (A) in the assembly listing. In either case, the current macro definition is terminated. Specifying the macro name in the .ENDM statement thus permits the Cross Assembler to detect missing .ENDM statements or improperly nested macro definitions.

The .ENDM directive must not have a label. If a legal label is attached, it will be ignored. If an illegal label is attached, the directive will be ignored.

The .ENDM directive may be followed by a comment field, as shown below:

```
          .MACRO TYPMSG      MESSGE ;Type a message.
          JSR    R5,TYPMSG
          .WORD  MESSGE
          .ENDM                     ;End of TYPMSG macro.
```

The final statement of every macro definition must be an .ENDM directive. The .ENDM directive is also used to terminate indefinite repeat blocks (see section 7.6) and may be used to terminate repeat blocks (see section 7.7).

7.1.3 .MEXIT Directive

Format:

```
          .MEXIT
```

The .MEXIT directive may be used to terminate a macro expansion before the end of the macro is encountered. This directive is also legal within repeat blocks (see sections 7.6 and 7.7). It is most useful in nested macros. The .MEXIT directive terminates the current macro as though an .ENDM directive had been encountered. Using the .MEXIT directive bypasses the complexities of nested conditional directives and alternate assembly paths, as shown in the following example:

```
.MACRO ALTR          N,A,B
   .
   .
   .
.IF    EQ  N         ;Start conditional assembly block.
   .
   .
   .
.MEXIT               ;Terminate macro expansion.
.ENDC                ;End conditional assembly block.
.
.
.
.ENDM                ;Normal end of macro.
```

In an assembly where the dummy symbol N is replaced by zero (see Table 6-6), the .MEXIT directive would assemble the conditional block and terminate the macro expansion. When macros are nested, a .MEXIT directive causes an exit to the next higher level of macro expansion.

A .MEXIT directive encountered outside a macro definition is flagged with an error code (O) in the assembly listing.

7.1.4 MACRO Definition Formatting

A form-feed character used within a macro definition causes a page eject during the assembly of the macro definition. A page eject, however, is not performed when the macro is expanded.

Conversely, when the .PAGE directive is used in a macro definition, it is ignored during the assembly of the macro definition, but a page eject is performed when that macro is expanded.

7.2 CALLING MACROS

Format:

```
            [label:] name          real arguments
```

where:
- label — represents an optional statement label.
- name — represents the name of the macro, as specified in the .MACRO directive (see section 7.1.1).
- real arguments — represent symbolic arguments which replace the dummy arguments listed in the .MACRO directive. When multiple arguments occur, they are separated by any legal separator. Arguments to the macro call are treated as character strings; their usage is determined by the macro definition.

A macro definition must be established by means of the .MACRO directive (see section 7.1.1) before the macro can be called and expanded within the source program.

When a macro name is the same as a user label, the appearance of the symbol in the operator field designates the symbol as a macro call; the appearance of the symbol in the operand field designates it as a label, as shown below:

```
ABS:        MOV   (R0),R1             ;ABS is defined as a label.
             .
             .
             .
            BR            ABS          ;ABS is considered a label.
             .
             .
             .
            ABS        #4,ENT,LAR     ;ABS is a macro call.
```

7.3 ARGUMENTS IN MACRO DEFINITIONS AND MACRO CALLS

Multiple arguments within a macro definition or macro call must be separated by one of the legal separating characters described in Section 3.1.1.

Macro definition arguments (dummy) and macro call arguments (real) normally maintain a strict positional relationship. That is, the first real argument in a macro call corresponds with the first dummy argument in a macro definition. Only the use of keyword arguments in a macro call can override this correspondence (see section 7.3.5).

For example, the following macro definition and its associated macro call contain multiple arguments:

```
.MACRO    REN A,B,C
 .
 .
 .
REN       ALPHA,BETA,<C1,C2>
```

Arguments which themselves contain separating characters must b enclosed in paired angle brackets. For example, the macro call:

```
REN       <MOV          X,Y>,#44,WEV
```

causes the entire expression

```
MOV     X,Y
```

to replace all occurrences of the symbol A in the macro definition. Real arguments within a macro call are considered to be character strings and are treated as a single entity during the macro expansion.

The up-arrow (^) construction allows angle brackets to be passed as part of the argument. This construction, for example, could have been used in the above macro call, as follows:

```
REN       ^/<MOV  X,Y>/,#44,WEV
```

causing the entire character string <MOV X,Y> to be passed as an argument.

Because of the use of the up-arrow (^) shown above, care must be taken when passing an argument beginning with a unary operator (^O, ^D, ^B, ^R, ^F ...). These arguments must be enclosed in angle brackets (as shown below) or the Cross Assembler will read th character following the up-arrow as a delimiter.

```
          REN <^O 411>,X,Y
```

The following macro call:

```
          REN #44,WEV^/MOV  X,Y/
```

contains only two arguments (#44 and WEV^/MOV X,Y/), because the up-arrow is a unary operator (see section 3.1.3) and it is not preceded by an argument separator.

As shown in the examples above, spaces can be used within bracketed argument constructions to increase the legibility of such expressions.

NOTE

> If an argument does not contain spaces, tabs, semicolons, or commas, it may include special characters without enclosing them in a bracketed construction.

7.3.1 Macro Nesting

Macro nesting occurs where the expansion of one macro includes a call to another. The depth of nesting allowed depends upon the amount of dynamic memory used by the source program being assembled.

To pass an argument containing legal argument delimiters to nested macros, enclose the argument in the macro definition within angle brackets, as shown in the coding sequence below. This extra set of angle brackets for each level of nesting is required in the macro definition, not in the macro call. The following example, using chip-specific mnemonics for the LSI-11 microprocessor, represents a sample macro nesting.

```
          .MACRO LEVEL1        DUM1,DUM2
          LEVEL2 <DUM1>
          LEVEL2 <DUM2>
          .ENDM

          .MACRO LEVEL2        DUM3
          DUM3
          ADD    #10,R0
          MOV     R0,(R1)+
          .ENDM
```

A call to the LEVEL1 macro, as shown below,

```
          LEVEL1 <MOV          X,R0>,<MOV R2,R0>
```

causes the following macro expansion to occur:

```
          MOV     X,R0
          ADD     #10,R0
          MOV     R0,(R1)+
          MOV     R2,R0
          ADD     #10,R0
          MOV     R0,(R1)+
```

When macro definitions are nested, the inner definition cannot be called until the outer macro has been called and expanded. For example, in the following coding:

```
          .MACRO LV1 A,B
             .
             .
             .
          .MACRO LV2 C
             .
             .
             .
          .ENDM
          .ENDM
```

the LV2 macro cannot be called and expanded until the LV1 macro has been expanded. Likewise, any macro defined within the LV2 macro definition cannot be called and expanded until LV2 has also been expanded.

7.3.2 Passing Numeric Arguments as Symbols

If the unary operator backslash (\) precedes an argument, the macro treats that argument as a numeric value in the current program radix. The ASCII characters representing this value are inserted in the macro expansion, and their function is defined in the context of the resulting code, as shown in the following example:

```
          .MACRO INC A,B
          CON    A,\B              ;B is treated as a number in current
B=B+1                              ;program radix.
          .ENDM
          .MACRO     CON            A,B
A´B:      .WORD 4                  ;A´B is described in Section 7.3.6
          .ENDM
            .
            .
            .
C=0       INC    X,C
```

The above macro call (INC) would thus expand to:

```
          X0:    .WORD         4
```

In this expanded code, the label X0: results from the concatenation of two real arguments. The single quote (´) character in the label A´B: concatenates the real arguments X and 0 as they are passed during the expansion of the macro. This type of argument construction is described in more detail in Section 7.3.6.

A subsequent call to the same macro would generate the following code

```
          X1:    .WORD         4
```

and so on, for later calls. The two macro definitions are necessary because the symbol associated with dummy argument B (that is, C) cannot be updated in the CON macro definition because the character 0 has replaced C in the argument string (INC X, C). In the CON macro definition, the number passed is treated as a string argument. (Where the value of the real argument is 0, only a single 0 character is passed to the macro expansion.)

Passing numeric values in this manner is useful in identifying source listings. For example, versions of programs created through conditional assemblies of a single source program can be identified through such coding as that shown below. Assume, for example, that the symbol ID in the macro call (IDT) has been equated elsewhere in the source program to the value 6.

```
        .MACRO  IDT       SYM     ;Assume that the symbol ID takes
        .IDENT  /V01.´SYM/        ;on a unique 2-digit value.
        .ENDM                     ;Where V01 is the update
          .                       ;version of the program.
          .
          .
        IDT   \ID
```

The above macro call would then expand to

```
        .IDENT /V01.6/
```

where 6 is the numeric value of the symbol ID.

7.3.3 Number of Arguments in Macro Calls

A macro can be defined with or without arguments. If more arguments appear in the macro call than in the macro definition, an error code (Q) is generated in the assembly listing. If fewer arguments appear in the macro call than in the macro definition, missing arguments are assumed to be null values. The conditional directives .IF B and .IF NB (see Table 6-6) can be used within the macro to detect missing arguments. The number of arguments can also be determined using the .NARG directive (Section 7.4.1).

7.3.4 Creating Local Symbols Automatically

A label is often required in an expanded macro. In the conventional macro facilities thus far described, a label must be explicitly specified as an argument with each macro call. The user must be careful in issuing subsequent calls to the same macro in order to avoid duplicating labels. This concern can be eliminated through a feature of the Cross Assembler that creates a unique symbol where a label is required in an expanded macro.

As noted in Section 3.4, the Cross Assembler can automatically create local symbols of the form n$, where n is a decimal integer within the range 30000 through 65535, inclusive. Such local symbols are created by the Cross Assembler in numerical order, as shown below:

```
        30000$
        30001$
          .
          .
          .
        65534$
        65535$
```

This automatic generation is invoked on each call of a macro whose definition contains a dummy argument preceded by the question mark (?) character, as in the following macro definition using the chip-specific mnemonics for the LSI-11 microprocessor:

```
          .MACRO     ALPHA,     A,?B ;Contains dummy argument B
                                     ;preceded by question mark.
          TST    A
          BEQ    B
          ADD    #5,A
B:
          .ENDM
```

A local symbol is created automatically by the Cross Assembler only when a real argument of the macro call is either null or missing, as shown in Example 1 below. If the real argument is specified in the macro call, however, the Cross Assembler inhibits the generation of a local symbol and normal argument replacement occurs, as shown in Example 2 below. (Examples 1 and 2 are both expansions of the ALPHA macro defined above.)

EXAMPLE 1: Create a Local Symbol for the Missing Argument:

```
          ALPHA  R1               ;Second argument is missing.
          TST    R1
          BEQ    30000$           ;Local symbol is created.
          ADD    #5,R1
30000$:
```

EXAMPLE 2: Do Not Create a Local Symbol:

```
          ALPHA  R2,XYZ           ;Second argument XYZ is specified.
          TST    R2
          BEQ    XYZ              ;Normal argument replacement occurs.
          ADD    #5,R2
XYZ:
```

Automatically created local symbols are restricted to the first 16(10) arguments of a macro definition.

Automatically created local symbols resulting from the expansion of a macro, as described above, do not establish a local symbol block in their own right.

When a macro has several arguments earmarked for automatic local symbol generation, substituting a specific label for one such argument risks assembly errors because the Cross Assembler constructs its argument substitution list at the point of macro invocation. Therefore, the appearance of a label, the .ENABL LSB directive, or the .PSECT directive, in the macro expansion will create a new local symbol block. The new local symbol block could leave local symbol references in the previous block and their symbol definitions in the new one, causing error codes in the assembly listing. Furthermore, a later macro expansion that creates local symbols in the new block may duplicate one of the symbols in question, causing an additional error code (P) in the assembly listing.

7.3.5 Keyword Arguments

Format:

name=string

where: name represents the dummy argument,

string represents the real symbolic argument.

The keyword argument may not contain embedded argument separators unless delimited as described in Section 7.3.

Macros may be defined with, and/or called with, keyword arguments. When a keyword argument appears in the dummy argument list of a macro definition, the specified string becomes the default real argument at macro call. When a keyword argument appears in the real argument list of a macro call, however, the specified string becomes the real argument for the dummy argument that matches the specified name, whether or not the dummy argument was defined with a keyword. If a match fails, the entire argument specification is treated as the next positional real argument.

A keyword argument may be specified anywhere in the dummy argument list of a macro definition and is part of the positional ordering of the argument. A keyword argument may also be specified anywhere in the real argument list of a macro call but, in this case, does not affect the positional ordering of the arguments.

```
                .LIST   ME
        ;
        ; Define a macro having keywords in dummy
        ; argument list
        ;
                .MACRO  TEST CONTRL=1,BLOCK,ADDRES=TEMP
                .WORD CONTRL
                .WORD BLOCK
                .WORD ADDRES
                .ENDM


        ;
        ; Now invoke several times
```

```
15                      ;
16
17     000000                       TEST     A,B,C
       000000 000000G               .WORD    A
       000002 000000G               .WORD    B
       000004 000000G               .WORD    C
18
19     000006                       TEST     ADDRES=20,BLOCK=30,CONTRL=40
       000006 000040                .WORD    40
       000010 000030                .WORD    30
       000012 000020                .WORD    20
20
21     000014                       TEST     BLOCK=5
       000014 000001                .WORD    1
       000016 000005                .WORD    5
       000020 000000G               .WORD    TEMP
22
23     000022                       TEST     CONTRL=5,ADDRES=VARIAB
       000022 000005                .WORD    5
       000024 000000                .WORD
       000026 000000G               .WORD    VARIAB
24
25     000030                       TEST
       000030 000001                .WORD    1
       000032 000000                .WORD
       000034 000000G               .WORD    TEMP
26
27     000036                       TEST     ADDRES=JACK!JILL
       000036 000001                .WORD    1
       000040 000000                .WORD
       000042 000000C               .WORD    JACK!JILL
28
29
30            000001                .END
```

7.3.6 Concatenation of Macro Arguments

The apostrophe or single quote character (´) operates as a legal delimiting character in macro definitions. A single quote that precedes and/or follows a dummy argument in a macro definition is removed, and the substitution of the real argument occurs at that point. For example, in the following statements:

```
          .MACRO DEF A,B,C,
A´B:      .ASCIZ /C/
          .BYTE  ´´A,´´B
          .ENDM
```

when the macro DEF is called through the statement:

```
           DEF    X,Y,<START>
```

it is expanded, as follows:

```
XY:        .ASCIZ /START/
           .BYTE  ´X,´Y
```

In expanding the first line, the scan for the first argument terminates upon finding the first apostrophe (´) character. Since A is a dummy argument, the apostrophe (´) is removed. The scan then resumes with B; B is also noted as another dummy argument. The two real arguments X and Y are then concatenated to form the label XY:. The third dummy argument is noted in the operand field of the .ASCIZ directive, causing the real argument START to be substituted in this field.

When evaluating the arguments of the .BYTE directive during expansion of the second line, the scan begins with the first apostrophe (´) character. Since it is neither preceded nor followed by a dummy argument, this apostrophe remains in the macro expansion. The scan then encounters the second apostrophe, which is followed by a dummy argument and is therefore discarded. The scan of argument A is terminated upon encountering the comma (,). The third apostrophe is neither preceded nor followed by a dummy argument and again remains in the macro expansion. The fourth (and last) apostrophe is followed by another dummy argument and is likewise discarded. (Four apostrophe (´) characters were necessary in the macro definition to generate two apostrophe (´) characters in the macro expansion.)

7.4 MACRO ATTRIBUTE DIRECTIVES: .NARG, .NCHR, AND .NTYPE

The assembler has three directives that allow the user to determine certain attributes of macro arguments: .NARG, .NCHR, and .NTYPE. The use of these directives permits selective modifications of a macro expansion, depending on the nature of the arguments being passed. These directives are described below.

7.4.1 .NARG Directive

Format:

```
            [label:]  .NARG symbol
```

where:

label — represents an optional statement label.

symbol — represents any legal symbol. This symbol is equated to the number of non-keyword arguments in the macro call currently being expanded. If a symbol is not specified, the .NARG directive is flagged with an error code (A) in the assembly listing.

The .NARG directive is used to determine the number of non-keyword arguments in the macro call currently being expanded. Hence, the .NARG directive can appear only within a macro definition; if it appears elsewhere, an error code (O) is generated in the assembly listing. An example of the .NARG directive is shown in Figure 7-1.

```
 1                         .TITLE   NARG
 2
 3                         .ENABLE  .LC
 4                         .LIST     ME
 5                 ;+
 6                 ; Example of the .NARG directive
 7                 ;-
 8
 9                 .MACRO  NULL     NUM
10                         .NARG    SYM
11                         .IF EQ   SYM
12                         .MEXIT
13                         .IFF
14                         .REPT    NUM
15                         NOP
16                         .ENDM
17                         .ENDC
18                 .ENDM
19
20 000000                  NULL
          000000           .NARG    SYM
                           .IF EQ   SYM
                           .MEXIT
                           .IFF
                           .REPT
                           NOP
                           .ENDM
                           .ENDC
21
22 000000                  NULL     6
          000001           .NARG    SYM
                           .IF EQ   SYM
                           .MEXIT
                           .IFF
          000006           .REPT    6
                           NOP
                           .ENDM
   000000 000240           NOP
   000002 000240           NOP
   000004 000240           NOP
   000006 000240           NOP
   000010 000240           NOP
   000012 000240           NOP
                           .ENDC
23
24        000001           .END
```

Figure 7-1: Example of .NARG Directive

7.4.2 .NCHR Directive

Format:

```
        [label:]   .NCHR symbol,<string>
```

where:

label — represents an optional statement label.

symbol — represents any legal symbol. This symbol is equated to the number of characters in the specified character string. If a symbol is not specified, the .NCHR directive is flagged with an error code (A) in the assembly listing.

comma — represents any legal separator (comma, space, and/or tab).

<string> — represents a string of printable characters. If the character string contains a legal separator (comma, space, and/or tab) the whole string must be enclosed within angle brackets (< >) or up-arrows (^). If the delimiting characters do not match or if the ending delimiter cannot be detected because of a syntactical error in the character string (thus prematurely terminating its evaluation), the .NCHR directive is flagged with an error code (A) in the assembly listing.

The .NCHR directive, which can appear anywhere in an assembler program, is used to determine the number of characters in a specified character string. This directive is useful in calculating the length of macro arguments. See the example below in Figure 7-2.

```
 1                              .TITLE    NCHR
 2
 3                              .ENABL    LC
 4                              .LIST     ME
 5                      ;+
 6                      ; Illustrate the .NCHR directive
 7                      ;-
 8
 9                      .MACRO  STRING    MESSAG
10                              .NCHR     $$$,MESSAG
11                              .WORD     $$$
12                              .ASCII    /MESSAG/
13                              .EVEN
14                      .ENDM
15
16 000000               MSG1:   STRING    <Hello>
           000005               .NCHR     $$$,Hello
           000005               .WORD     $$$
  000002      110               .ASCII    /Hello/
  000003      145
  000004      154
  000005      154
  000006      157
                                .EVEN
17
18         000001               .END
```

Figure 7-2: Example of .NCHR Directive

7.4.3 .NTYPE Directive

Format:

```
        [label:]   .NTYPE symbol,aexp
```

where: label represents an optional statement label.

symbol represents any legal symbol. This symbol is equated to the 6-bit addressing mode of the following expression (aexp). If a symbol is not specified, the .NTYPE directive is flagged with an error code (A) in the assembly listing.

comma represents any legal separator (comma, space, or tab).

aexp represents any legal address expression, as used with an opcode. If no argument is specified, an error code (A) will appear in the assembly listing.

The .NTYPE directive is used to determine the addressing mode of a specified macro argument. Hence, the .NTYPE directive can appear only within a macro definition; if it appears elsewhere, it is flagged with an error code (O) in the assembly listing.

For additional information concerning addressing modes, refer to Chapter 5 and Appendix B.

7.5 .ERROR AND .PRINT DIRECTIVES

Format:

```
        [label:]   .ERROR [expr]    ;text
```

where:

label — represents an optional statement label.

expr — represents an optional expression whose value is output when the .ERROR directive is encountered during assembly.

; — denotes the beginning of the text string.

text — represents the message associated with the .ERROR directive.

The .ERROR directive is used to output messages to the listing file during assembly pass 2. A common use of this directive is to alert the user to a rejected or erroneous macro call or to the existence of an illegal set of conditions in a conditional assembly. If the listing file is not specified, the .ERROR messages are output to the cross-assembler output device.

Upon encountering an .ERROR directive anywhere in a source program, the Cross Assembler outputs a single line containing:

1. An error code (P);
2. The sequence number of the .ERROR directive statement;
3. The value of the current location counter;
4. The value of the expression, if one is specified;
5. The source line containing the .ERROR directive.

For example, the following directive

```
        .ERROR  A  ;Invalid macro argument
```

causes a line in the following form to be output to the listing file:

```
      Seq. Loc.     Exp.
      No.  No.      Value                        Text

P     512  005642   000076       .ERROR  A      ;Invalid macro
                                                  argument
```

The PRINT directive is identical in function to the .ERROR directive, except that it is not flagged with the error code (P).

7.6 INDEFINITE REPEAT BLOCK DIRECTIVES: .IRP AND .IRPC

An indefinite repeat block is similar to a macro definition with only one dummy argument. At each expansion of the indefinite repeat range, this dummy argument is replaced with successive elements of a real argument list. Since the repeat directive and its associated range are coded in-line within the source program, this type of macro definition and expansion does not require calling the macro by name, as required in the expansion of the conventional macros previously described in this chapter.

An indefinite repeat block can appear either within or outside another macro definition, indefinite repeat block, or repeat block. The rules for specifying indefinite repeat block arguments are the same as for specifying macro arguments (see section 7.3).

7.6.1 .IRP Directive

Format:

```
        [label:]       .IRP sym,<argument list>
               .
               .
               .
             (range of indefinite repeat block)
               .
               .
               .
             .ENDM
```

where: label   represents an optional statement label.

NOTE

> Although it is legal for a label to appear on a .MACRO directive, this practice is discouraged, especially in the case of nested macro definitions, because invalid labels or labels constructed with the concatenation character will cause the macro directive to be ignored. This may result in improper termination of the macro definition.
>
> This warning also applies to .IRPC and .REPT.

sym represents a dummy argument that is replaced with successive real arguments from within the angle brackets. If no dummy argument is specified, the .IRP directive is flagged with an error code (A) in the assembly listing.

comma represents any legal separator (comma, space, and/or tab).

<argument list> represents a list of real arguments enclosed within angle brackets that is to be used in the expansion of the indefinite repeat range. A real argument may consist of one or more characters; multiple arguments must be separated by any legal separator (comma, space, and/or tab). If no real arguments are specified, no action is taken.

range represents the block of code to be repeated once for each occurrence of a real argument in the list. The range may contain other macro definitions, repeat ranges and/or the .MEXIT directive (see section 7.1.3).

.ENDM indicates the end of the indefinite repeat block range.

The .IRP directive is used to replace a dummy argument with successive real arguments specified in an argument string. This replacement process occurs during the expansion of an indefinite repeat block range. (See Figure 7-3 below.)

```
 1                              .TITLE    IRPTEST
 2
 3                              .LIST     ME
 4                      ;+
 5                      ; Illustrate the .IRP and .IRPC directives
 6                      ; by creating a pair of RAD50 tables
 7                      ;-
 8
 9 000000               REGS:   .IRP      REG,<PC,SP,R5,R4,R3,R2,R1,R0>
10                              .RAD50    /REG/
11                              .ENDR
   000000  062170               .RAD50    /PC/
   000002  074500               .RAD50    /SP/
   000004  072770               .RAD50    /R5/
   000006  072720               .RAD50    /R4/
   000010  072650               .RAD50    /R3/
   000012  072600               .RAD50    /R2/
   000014  072530               .RAD50    /R1/
   000016  072460               .RAD50    /R0/
12
13 000020               REGS2:  .IRPC     NUM,<76543210>
14                              .RAD50    /R'NUM/
15                              .ENDR
   000020  073110               .RAD50    /R7/
   000022  073040               .RAD50    /R6/
   000024  072770               .RAD50    /R5/
   000026  072720               .RAD50    /R4/
   000030  072650               .RAD50    /R3/
   000032  072600               .RAD50    /R2/
   000034  072530               .RAD50    /R1/
   000036  072460               .RAD50    /R0/
16         000001               .END
```

Figure 7-3: Example of .IRP and .IRPC Directives

7.6.2 .IRPC Directive

Format:

```
        [label:]          .IRPC sym,<string>
                    .
                    .
                    .
                 (range of indefinite repeat block)
                    .
                    .
                    .
                 .ENDM
```

where:

label — represents an optional statement label (see Note in Section 7.6.1).

sym — represents a dummy argument that is replaced with successive real arguments from within the angle brackets. If no dummy argument is specified, the .IRPC directive is flagged with an error code (A) in the assembly listing.

comma — represents any legal separator (comma, space, and/or tab).

<string> — represents a list of characters, enclosed within angle brackets, to be used in the expansion of the indefinite repeat range. Although the angle brackets are required only when the string contains separating characters, their use is recommended for legibility.

range — represents the block of code to be repeated once for each occurrence of a character in the list. The range may contain macro definitions, repeat ranges and/or the .MEXIT directive (see section 7.1.3).

.ENDM — indicates the end of the indefinite repeat block range.

The .IRPC directive is available to permit single character substitution, rather than argument substitution. On each iteration of the indefinite repeat range, the dummy argument is replaced with successive characters in the specified string.

An example of the use of the .IRPC directive is shown in Figure 7-3.

7.7 REPEAT BLOCK DIRECTIVE: .REPT, .ENDR

Format:

```
        [label:]                .REPT exp
             .
             .
             .
          (range of repeat block)
             .
             .
             .
          .ENDR
```

where:

label represents an optional statement label (see Note in Section 7.6.1).

exp represents any legal expression. This value controls the number of times the block of code is to be assembled within the program. When the expression value is less than or equal to zero (0), the repeat block is not assembled. If this expression is not an absolute value, the .REPT statement is flagged with an error code (A) in the assembly listing.

range represents the block of code to be repeated. The repeat block may contain macro definitions, indefinite repeat blocks, other repeat blocks and/or the .MEXIT directive (see section 7.1.3).

.ENDM or .ENDR indicates the end of the repeat block range.

The .REPT directive is used to duplicate a block of code, a certain number of times, in line with other source code.

7.8 MACRO LIBRARY DIRECTIVE: MCALL

Format:

```
        .MCALL  argl,arg2,...argn
```

where:

argl, arg2,... argn	represent the symbolic names of the macro definitions required in the assembly of the source program. The names must be separated by any legal separator (comma, space, and/or tab).

The .MCALL directive allows you to indicate in advance those system and/or user-defined macro definitions that are not defined within the source program but which are required to assemble the program. The .MCALL directive must appear before the first occurrence of a call to any externally defined macro.

The /M switch, used with an input file specification, indicates to the Cross Assembler that the file is a macro library. When a macro call is encountered in the source program, the Cross Assembler first searches the user macro library for the named macro definitions, and, if necessary, continues the search with the system macro library.

Any number of such user-supplied macro files may be designated. For multiple library files, the search for the named macros begins with the last such file specified. The files are searched in reverse order until the required macro definitions are found, finishing, if necessary, with a search of the system macro library.

If any named macro is not found upon completion of the search, the .MCALL statement is flagged with an error code (U) in the assembly listing. Furthermore, a statement elsewhere in the source program that attempts to expand such an undefined macro is flagged with an error code (O) in the assembly listing.

APPENDIX A

CROSS ASSEMBLER CHARACTER SETS

A.1 ASCII CHARACTER SET

Even Parity Bit	7-Bit Octal Code	Character	Remarks
0	000	NUL	Null, tape feed, CONTROL/SHIFT/P.
1	001	SOH	Start of heading; also SOM, start of message, CONTROL/A.
1	002	STX	Start of text; also EOA, end of address, CONTROL/B.
0	003	ETX	End of text; also EOM, end of message, CONTROL/C.
1	004	EOT	End of transmission (END); shuts off TWX machines, CONTROL/D.
0	005	ENQ	Enquiry (ENQRY); also WRU, CONTROL/E.
0	006	ACK	Acknowledge; also RU, CONTROL/F.
1	007	BEL	Rings the bell. CONTROL/G.
1	010	BS	Backspace; also FEO, format effector. backspaces some machines, CONTROL/H.
0	011	HT	Horizontal tab. CONTROL/I.
0	012	LF	Line feed or Line space (new line); advances paper to next line. CONTROL/J.
1	013	VT	Vertical tab (VTAB). CONTROL/K.
0	014	FF	Form Feed to top of next page (PAGE). CONTROL/L.
1	015	CR	Carriage return to beginning of line. CONTROL/M.
1	016	SO	Shift out; changes ribbon color to red. CONTROL/N.
0	017	SI	Shift in; changes ribbon color to black. CONTROL/O.
1	020	DLE	Data link escape. CONTROL/P (DCO).

0	021	DC1	Device control 1; turns transmitter (READER) on, CONTROL/Q (X ON). 0 022 DC2 Device control 2; turns punch or auxiliary on. CONTROL/R (TAPB, AUX ON).
1	023	DC3	Device control 3; turns transmitter (READER) off, CONTROL/S (X OFF).
0	024	DC4	Device control 4; turns punch or auxiliary off. CONTROL/T (AUX OFF).
1	025	NAK	Negative acknowledge; also ERR, ERROR. CONTROL/U.
1	026	SYN	Synchronous file (SYNC). CONTROL/V.
0	027	ETB	End of transmission block; also LEM, logical end of medium. CONTROL/W.
0	030	CAN	Cancel (CANCL). CONTROL/X.
1	031	EM	End of medium. CONTROL/Y.
1	032	SUB	Substitute. CONTROL/Z.
0	033	ESC	Escape. CONTROL/SHIFT/K.
1	034	FS	File separator. CONTROL/SHIFT/L.
0	035	GS	Group separator. CONTROL/SHIFT/M.
0	036	RS	Record separator. CONTROL/SHIFT/N.
1	037	US	Unit separator. CONTROL/SHIFT/O.
1	040	SP	Space.
0	041	!	
0	042	"	
1	043	#	
0	004	$	
1	045	%	
1	046		
0	047	´	Accent acute or apostrophe.
0	050	(	
1	051	)	
1	052		
0	053	+	
1	054	´	
0	055	–	
0	056	.	
1	057	/	
0	060	0	
1	061	1	

1	052	2	
0	063	3	
1	064	4	
0	065	5	
0	066	6	
1	057	7	
1	070	8	
0	071	9	
0	072	:	
1	073	;	
0	074	<	
1	075	=	
1	076	>	
0	077	?	
1	100	@	
0	101	A	
0	102	B	
1	103	C	
0	104	D	
1	105	E	
1	106	F	
0	107	G	
0	110	H	
0	111	I	
1	112	J	
0	113	K	
1	114	L	
0	115	M	
0	116	N	
1	117	O	
0	120	P	
1	121	Q	
1	122	R	
0	123	S	
1	124	T	
0	125	U	
0	126	V	
1	127	W	
1	130	X	
0	131	Y	
0	132	Z	
1	133	[	shift/k.
0	134		shift/l.
1	135	]	shift/m.
1	136	^	*
0	137	_	**
0	140	`	Accent grave.
1	141	a	

1	142	b	
0	143	c	
1	144	d	
0	145	e	
0	146	f	
1	147	g	
1	150	h	
0	151	i	
0	152	j	
1	153	k	
0	154	l	
1	155	m	
1	156	n	
0	157	o	
1	160	p	
0	161	q	
0	162	r	
1	163	s	
0	164	t	
1	165	u	
1	166	v	
0	167	w	
0	170	x	
1	171	y	
1	172	z	
0	173		
1	174		
0	175		This code generated by ALTMODE.
0	176		This code generated by prefix key (if present).
1	177		DEL Delete, Rubout.

*^ Appears as # or^ on some machines.
** Appears as < on some machines.

A.2 RADIX-50 CHARACTER SET

Character	ASCII Octal Equivalent	Radix-50 Equivalent
Space	40	0
A-Z	101-132	1-32
$	44	33
.	56	34
Unused		35
0-9	60-71	36-47

The maximum Radix-50 value is, therefore:

```
47*50**2+47*50+47=174777
```

The following table provides a convenient means of translating between the ASCII character set and its Radix-50 equivalents. For example, given the ASCII string X2B, the Radix-50 equivalent (arithmetic performed in octal) is:

```
X=113000
2=002400
B=000002
X2B=115402
```

Single Char. or First Char.		Second Character		Third Character	
Space	000000	Space	000000	Space	000000
A	003100	A	000050	A	000001
B	006200	B	000120	B	000002
C	011300	C	000170	C	000003
D	014400	D	000240	D	000004
E	017500	E	000310	E	000005
F	022600	F	000360	F	000006
G	025700	G	000430	G	000007
H	031000	H	000500	H	000010
I	034100	I	000550	I	000011
J	037200	J	000620	J	000012
K	042300	K	000670	K	000013
L	045400	L	000740	L	000014
M	050500	M	001010	M	000015
N	053600	N	001060	N	000016
O	056700	O	001130	O	000017
P	062000	P	001200	P	000020
Q	065100	Q	001250	Q	000021
R	070200	R	001320	R	000022
S	073300	S	001370	S	000023
T	076400	T	001440	T	000029
U	101500	U	001510	U	000025
V	104600	V	001560	V	000026
W	107700	W	001630	W	000027
X	113000	X	001700	X	000030
Y	116100	Y	001750	Y	000031
Z	121200	Z	002020	Z	000032
$	124300	$	002070	$	000033
.	127400	.	002140	.	000034
Unused	132500	Unused	002210	Unused	000035
0	135600	0	002260	0	000036
1	140700	1	002330	1	000037
2	144000	2	002400	2	000040
3	147100	3	002450	3	000041
4	152200	4	002520	4	000042
5	155300	5	002570	5	000043
6	160400	6	002640	6	000044
7	163500	7	002710	7	000045
8	166600	8	002760	8	000046
9	171700	9	003030	9	000047

APPENDIX B

CROSS ASSEMBLER LANGUAGE AND DIRECTIVES

B.1 SPECIAL CHARACTERS

Character	Designation	Function
:	Colon	Label terminator.
::	Double colon	Label terminator; defines the label as a global label.
=	Equal sign	Direct assignment operator and macro keyword indicator.
==	Double equal sign	Direct assignment operator; defines the symbol as a global symbol.
	Tab	Item or field terminator.
	Space	Item or field terminator/ separator.
#	Number sign	**
@	At sign	**
(	Left parenthesis	**
)	Right parenthesis	**
.	Period	Current location counter.
,	Comma	Operand field separator.
;	Semicolon	Comment field indicator.
<	Left angle bracket	Initial argument or expression indicator.

>	Right angle bracket	Terminal argument or expression indicator.
+	Plus sign	Arithmetic addition operator or **.
-	Minus sign	Arithmetic subtraction operator or **.
*	Asterisk	Arithmetic multiplication operator.
/	Slash	Arithmetic division operator.
&	Ampersand	Logical AND operator.
!	Exclamation point	Logical inclusive OR operator.
"	Double quote	Double ASCII character indicator.
'	Single quote	Single ASCII character indicator; or concatenation indicator.
^	Up arrow or circumflex	Universal unary operator or argument indicator.
\	Backslash	Macro call numeric argument indicator.
[	Left square bracket	**
]	Right square bracket	**

** Refer to chapter 5 of this manual for chip-specific syntax.

B.2 CROSS-ASSEMBLER OPERATORS AND DIRECTIVES

The following table summarizes the Emulogic Cross Assembler operators and directives. "Section Reference" refers to the section or sections where you will find a detailed description of a particular directive.

Form	Section Reference	Operation
'	6.3.4 7.3	Followed by one ASCII character a single quote (apostrophe) generates a word which contains the 7-bit ASCII representation of the character in the low-order byte and zero in the high-order byte. This character is also used as a concatenation indicator in the expansion of macro arguments.
"	6.3.3	Followed by two ASCII characters a double quote generates a word which contains the 7-bit ASCII representation of the two characters. The first character is stored in the low-order byte; the second character is stored in the high-order byte.
^Bn	6.4.1.2	A temporary radix control, causes the value n to be treated as a binary number.
^Dn	6.4.1.2	A temporary radix control, causes the value n to be treated as a decimal number.

^Hn	6.4.1.2	A temporary radix control, causes the value n to be treated as a hexadecimal number.
^On	6.4.1.2	A temporary radix control, causes the value n to be treated as an octal number.
^Rccc	6.3.8	Converts ccc to Radix-50 form.
.ASCII /string/	6.3.4	Generates a block of data containing the ASCII equivalent of the character string (enclosed in delimiting characters), one character per byte.
.ASCIZ /string/	6.3.6	Generates a block of data containing the ASCII equivalent of the character string (enclosed in delimiting characters), one character per byte, with a zero byte terminating the specified string.
.ASECT	6.7.2	Begins or resumes the absolute program section.

.BLKB exp	6.5.3	Reserves a block of storage space whose length in bytes is determined by the specified expression.
.BLKW exp	6.5.3	Reserves a block of storage space whose length in words is determined by the specified expression.
.BLKL exp	6.5.3	Reserves a block of storage space whose length in words is determined by the specified expression (used for 32-bit processors only).
.BYTE exp1,exp2,..	6.3.1	Generates successive bytes of data; each byte contains the value of the corresponding specified expression.
.DSABL arg	6.2.1	Disables the function specified by the argument.
.ENABL arg	6.2.1	Enables (invokes) the function specified by the argument.
.END [exp]	6.6	Indicates the logical end of the source program. The optional argument specifies the transfer address where program execution is to begin.
.ENDC	6.9.1	Indicates the end of a conditional assembly block.
.ENDM [name]	7.1.2	Indicates the end of the current repeat block, indefinite repeat block, or macro definition. The optional name, if used, must be identical to the name specified in the macro definition.

.ENDR	7.7	Indicates the end of the current repeat block. This directive is provided for compatibility with other cross assemblers.
.ERROR exp;text	7.5	A user-invoked error directive, causes output to the listing file or the command output device containing the optional expression and the statement containing the directive.
.EVEN	6.5.1	Ensures that the current location counter contains an even address by adding 1 if it is odd.
.GLOBL sym1,sym2,...	6.8.1	Defines the symbol(s) specified as global symbol(s).
.IDENT /string/	6.1.4	Provides a means of labeling the object module with the program version number. The version number is the Radix-50 string appearing between the paired delimiting characters.
.IF cond,arg1	6.9.1	Begins a conditional assembly block of source code which is included in the assembly only if the stated condition is met with respect to the argument(s) specified.

.IFF	5.9.2	Appears only within a conditional assembly block, indicating the beginning of a section of code to be assembled if the condition upon entering the block tests false.
.IFT	6.9.2	Appears only within a conditional assembly block, indicating the beginning of a section of code to be assembled if the condition upon entering the block tests true.
.IFTF	6.9.2	Appears only within a conditional assembly block, indicating the beginning of a section of code to be assembled unconditionally.
.IIF cond,arg,	6.9.3	Acts as a 1-line conditional statement assembly block where the condition is tested for the argument specified. The statement is assembled only if the condition tests true.
.IRP sym, <arg1,arg2,...>	7.6.1	Indicates the beginning of an indefinite repeat block in which the symbol specified is replaced with successive elements of the real argument list enclosed within angle brackets.

.IRPC sym,<string>	7.6.2	Indicates the beginning of an indefinite repeat block in which the specified symbol takes on the value of successive characters, optionally enclosed within angle brackets.
.LIST [arg]	6.1.1	Without an argument, the .LIST directive increments the listing level count by 1. With an argument, this directive does not alter the listing level count, but formats the assembly listing according to the argument specified.
.LONG exp1,exp2,...	6.3.3	Generates successive long words of data; each long word contains the value of the corresponding specified expression.
.MACRO name,arg1,...	7.1.1	Indicates the start of a macro definition having the specified name and the following dummy arguments.
.MCALL arg1,arg2,...	7.8	Specifies the symbolic names of the user or system macro definitions required in the assembly of the current user program, but which are not defined within the program.
.MEXIT	7.1.3	Causes an exit from the current macro expansion or indefinite repeat block.

.NARG symbol	7.4.1	Appearing only within a macro definition, equates the specified symbol to the number of arguments in the macro call currently being expanded.
.NCHR symbol,<string>	7.4.2	Appearing anywhere in a source program, equates the symbol specified to the number of characters in the specified string.
.NLIST [arg]	6.1.1	Without an argument, decrements the listing level count by 1. With an argument, this directive suppresses that portion of the listing specified by the argument.
.NTYPE symbol,aexp	7.4.3	Appearing only within a macro definition, equates the symbol to the 6-bit addressing mode of the specified address expression.
.ODD	6.5.2	Ensures that the current location counter contains an odd address by adding 1 if it is even.
.PAGE	6.1.5	Causes the assembly listing to skip to the top of the next page and to increment the page count.

.PRINT exp;text	7.5	User-invoked message directive; causes output to the listing file or the command output device containing the optional expression and the statement containing the directive.
.PSECT name,att1,...	6.7.1	Begins or resumes a named or unnamed program section having the specified attributes.
.RADIX n	6.4.1.1	Alters the current program radix to n, where n is 8 or 16.
.RAD50 /string/	6.3.7	Generates a block of data containing the Radix-50 equivalent of the character string enclosed within delimiting characters.
.REM comment-character	6.1.6	Allows a programmer to insert a block of comments into an assembler source program without having to precede the comment lines with the comment character (;).

.REPT exp	7.7	Begins a repeat block; causes the section of code up to the next .ENDM or .ENDR directive to be repeated number of times specified as exp.
.SBTTL string	6.1.3	Causes the specified string to be printed as part of the assembly listing page header. The string component of each .SBTTL directive is collected into a table of contents at the beginning of the assembly listing.
.TITLE string	6.1.2	Assigns the first six Radix-50 characters in the string as an object module name and causes the string to appear on each page of the assembly listing.
.WORD exp1,exp2,...	6.3.2	Generates successive words of data; each word contains the value of the corresponding specified expression.

APPENDIX C

ERROR MESSAGES

An error code is printed as the first character in a source line containing an error. This error code identifies the error condition detected during the processing of the line. Example:

```
Q    26 000236  010102          MOV R1,R2,A
```

The extraneous argument A in the MOV instruction causes the line to be flagged with a Q (syntax) error.

```
-----------------------------------------------------------------
Error Code                Meaning
-----------------------------------------------------------------
```

A — Assembly error. Because many different conditions produce this error message, the directives which may yield a general assembly error have been categorized below to reflect these error conditions:

CATEGORY 1: ILLEGAL ARGUMENT SPECIFIED.

.RADIX -- A value other than 8 or 16 is specified as a new radix.

.LIST/.NLIST -- An illegal argument is specified with the directive (see Table 6-2).

.ENABL/.DSABL -- An illegal argument is specified with the directive, or the attribute arguments of a previously declared program section. (See Table 6-3).

.PSECT -- An illegal argument is specified with the directive, or the attribute arguments of a previously declared program section change (see Table 6-4 and Section 6.7.1.1).

A (cont.)

.IF/.IIF -- An illegal conditional test or an illegal argument expression value is specified with the directive (see Table 6-6)

.MACRO -- An illegal or duplicate symbol found in dummy argument list.

.TITLE -- Program name is not specified in the directive, or first non-blank character following the directive is a non-Radix-50 character.

.IRP/.IRPC -- No dummy argument is specified in the directive.

.NARG/.NCHAR/.NTYPE -- No symbol is specified in the directive.

.IF/.IIF -- No conditional argument is specified in the directive.

CATEGORY 2: UNMATCHED DELIMITER/ILLEGAL ARGUMENT CONSTRUCTION.

.ASCII/.ASCIZ/.RAD50/.IDENT -- Character string or argument string delimiters do not match, or an illegal character is used as a delimiter, or an illegal argument construction is used in the directive.

.NCHAR -- Character string delimiters do not match, or an illegal character is used as a delimiter in the directive.

A (cont.)

CATEGORY 3: GENERAL ADDRESSING ERRORS.

This type of error results from one of several possible conditions:

1. Permissible range of a branch instruction has been exceeded.

2. A statement makes invalid use of the current location counter. For example, a ".=expression" statement attempts to force the current location counter to cross program section (.PSECT) boundaries.

3. A statement contains an invalid address expression:

   In cases where an absolute address expression is required, specifying a global symbol, a relocatable value, or a complex relocatable value (see Section 3.9) results in an invalid address expression.
   If an undefined symbol is made a default global reference by the .ENABL GBL directive (see Section 6.2.1) during pass 1, any attempt to redefine the symbol during pass 2 will result in an invalid address expression.

   In cases where a relocatable address expression is required, either a relocatable or absolute value is permissible, but a global symbol or a complex relocatable value in the statement results in an invalid address expression.

A (cont.)

For example:

.BLKB/.BLKW/.REPT -- User has not specified an absolute value or an expression which reduces to an absolute value has been specified with the directive.

4. Multiple expressions are not separated by a comma. This condition causes the next symbol to be evaluated as part of the current expression.

CATEGORY 4: ILLEGAL FORWARD REFERENCE.

This type of error results from either of two possible conditions:

1. A global assignment statement (symbol==expression) contains a forward reference to another symbol.

2. An expression defining the value of the current location counter contains a forward reference.

B Bounding error. Instructions or word data are being assembled at an odd address. The location counter is incremented by 1. (Only used for microprocessors that must start instructions on word boundaries.)

D Doubly defined symbol referenced. Reference was made to a symbol which is defined more than once.

E End directive not found. When the end-of-file is reached during source input and the .END directive has not yet been encountered, the assembler generates this error code, ends assembly pass 1, and proceeds with assembly pass 2. Also caused by assembler-stack overflow. In this case the assembler will place a question mark (?) into the line at the point where the overflow occurred.

I Illegal character detected. Illegal characters which are also non-printable are replaced by a question mark (?) on the listing. The character is then ignored.

L Input line is greater than 132(10) characters in length. Currently, this error condition is caused only during macro expansion when longer real arguments, replacing the dummy arguments, cause a line to exceed 132(10) characters.

M Multiple definition of a label. A label was encountered which was equivalent (in the first six characters) to a label previously encountered.

N A number contains a digit that is not in the current program radix. The number is evaluated as a decimal value.

O Opcode error. Directive out of context. Permissible nesting level depth for conditional assemblies has been exceeded. Attempt to expand a macro which was unidentified after .MCALL search.

P Phase error. A label´s definition of value varies from one assembly pass to another or a multiple definition of a local symbol has occurred within a local symbol block. Also, when in a local symbol block defined by the .ENABL LSB directive, an attempt has occurred to define a local symbol in a program section other than that which was in effect when the block was entered. An error code P also appears if an .ERROR directive is assembled.

R Questionable syntax. Arguments are missing, too many arguments are specified, or the instruction scan was not completed.

T Truncation error. A number generated more bits than allowed.

U Undefined symbol. An undefined symbol was encountered during the evaluation of an expression; such an undefined symbol is assigned a value of zero. Other possible conditions which result in this error code include unsatisfied macro names in the list of .MCALL arguments and a direct assigment (symbol=expression) statement which contains a forward reference to a symbol whose definition also contains a forward reference; also, a local symbol may have been referenced that does not exist in the current local symbol block.

EMULOGIC, INCORPORATED
3 Technology Way
Norwood, Massachusetts 02062
Telephone: (617) 329-1031
Telex: 710-336-5908