DEC-11-LASMA-A-D

# DOS/BATCH Assembler (MACRO) Programmer's Manual

FOR THE DOS/BATCH OPERATING SYSTEM

Monitor Version VØ9

August 1973

For additional copies, order No. DEC-11-LASMA-A-D from Digital Equipment Corporation, Software Distribution Center, Maynard, Massachusetts Ø1754.

Your attention is invited to the last two pages of this document. The "How to Obtain Software Information" page tells you how to keep up-to-date with DEC's software. The "Reader's Comments" page when filled in and mailed, is beneficial to both you and DEC; any comments received are acknowledged and are considered when documenting subsequent manuals.

Associated Documents:

DOS/BATCH Monitor
Programmer's Manual, DEC-11-OMPMA-A-D

DOS/BATCH User's Guide, DEC-11-OBUGA-A-D

DOS/BATCH FORTRAN Compiler and Object Time System
Programmer's Manual, DEC-11-LFRTA-A-D

DOS/BATCH System Manager's Guide, DEC-11-OSMGA-A-D

DOS/BATCH File Utility Package (PIP)
Programmer's Manual, DEC-11-UPPAA-A-D

DOS/BATCH Debugging Program (ODT-11R)
Programmer's Manual, DEC-11-UDEBA-A-D

DOS/BATCH Linker (LINK)
Programmer's Manual, DEC-11-ULKAA-A-D

DOS/BATCH Librarian (LIBR)
Programmer's Manual, DEC-11-ULBAA-A-D

DOS/BATCH Text Editor (EDIT-11)
Programmer's Manual, DEC-11-UEDAA-A-D

DOS/BATCH File Compare Program (FILCOM)
Programmer's Manual, DEC-11-UFCAA-A-D

DOS/BATCH File Dump Program (FILDMP)
Programmer's Manual, DEC-11-UFLDA-A-D

DOS/BATCH Verification Program (VERIFY)
Programmer's Manual, DEC-11-UVERA-A-D

DOS/BATCH Disk Initializer (DSKINT)
Programmer's Manual, DEC-11-UDKIA-A-D

## PREFACE

This manual describes the PDP-11 MACRO-11 Assembler and Assembly Language. It is recommended that the reader refer to the PDP-11 Processor Handbook and, optionally, the PDP-11 Peripherals and Interfacing Handbook. References are made to these handbooks throughout this document (although this document is complete by itself, the additional material provides further details). The user is also advised to obtain a PDP-11 Pocket Instruction List card for easy reference. (These items can be obtained from the Software Distribution Center.)

MACRO-11 operates under the PDP-11 DOS/BATCH Monitor.

Some notable features of MACRO-11 are:

1. Program and command string control of assembly functions;
2. Device and filename specifications for input and output files;
3. Error listing on command output device;
4. Alphabetized, formatted symbol table listing;
5. Relocatable object modules;
6. Global symbols for linking between object modules;
7. Conditional assembly directives;
8. Program sectioning directives;
9. User-defined macros;
10. Comprehensive set of system macros; and
11. Extensive listing control.

CONTENTS

# CHAPTER 1

# EFFECTIVE USE OF ASSEMBLY LANGUAGE PROGRAMMING

This Chapter presents a brief overview of some fundamental software concepts essential to efficient assembly language programming of the PDP-11 family of computers. A description of the hardware components of the PDP-11 family can be found in the two DEC paperback handbooks:

PDP-11 Processor Handbook (11/40 or 11/45 edition)
PDP-11 Peripherals and Interfacing Handbook

No attempt is made in this document to describe the PDP-11 hardware or the function of the various PDP-11 instructions. The reader is advised to become familiar with this material before proceeding.

## 1.1 STANDARDS AND CONVENTIONS

Because assembly level programming deals directly with the host hardware, greater care must be taken in specifying programming standards and conventions if code written by different groups is to be easily interchanged. The payoff achievable from strict adherence to standards can be considerable. When a set of standards guides the entire programming process, the total programming effort becomes easier to

plan;
comprehend;
test;
modify; and
convert.

Even though standards must take into consideration local installation requirements, many components of the programming process have universal applicability. Appendix E contains a set of recommended programming standards. It is a minimal set found to be practical and useful. Users adhering to these standards in coding their own software will reap the benefits of interchangeability, and tend to develop work-sharing arrangements mutually rewarding to DIGITAL and the user.

## 1.2 POSITION-INDEPENDENT CODE (PIC)

The output of a MACRO-11 assembly is a relocatable object module. LINK can bind one or more modules together and create an executable task.

Once built, a program can generally be loaded and executed only at the address specified at LINK time. This is because LINK has had to make adjustments in some codes to reflect the memory locations in which the program is to run.

# CHAPTER 2

# SOURCE PROGRAM FORMAT

A source program is composed of a sequence of source lines, where each line contains a single assembly language statement.

An assembly language line can contain up to 132(decimal) characters. Beyond this limit an I/O error is generated.

## 2.1 STATEMENT FORMAT

A statement can contain up to four fields which are identified by order of appearance and by specified terminating characters. The general format of a MACRO-11 assembly language statement is:

```
label: operator operand        ;comments
```

The label and comment fields are optional. The operator and operand fields are interdependent; either may be omitted depending upon the contents of the other.

The Assembler interprets and processes these statements one by one, generating one or more binary instructions or data words, or performing an assembly process. A statement must contain one of these fields and may contain all four types. (Blank lines are legal.)

Some statements have one operand, for example:

```
        CLR     R0
```

while others have two, for example:

```
        MOV     #344,R2
```

An assembly language statement must be complete on one source line. No continuation lines are allowed.

MACRO-11 source statements may be formatted such that use of the TAB character causes the statement fields to be aligned. The standards used are:

Label - column 1;

Operator - column 9;

Operand(s) - column 17;

Comments - column 33.

For example:

```
REGTST: BIT     #MASK,VALUE     ;3 BITS?
```

2.1.1 Label Field

A label is a user-defined symbol which is assigned the value of the current location counter and entered into the user-defined symbol table. The value of the label may be either absolute or relocatable, depending on whether the location counter value is currently absolute or relocatable. In the latter case, the absolute value of the symbol is assigned by LINK; i.e., the stated relocatable value plus a the relocation bias, calculated by LINK.

A label is a symbolic means of referring to a specific location within a program. If present, a label always occurs first in a statement and must be terminated by a colon. For example, if the current location is absolute 100(octal), the statement:

```
ABCD:   MOV     A,B
```

assigns the value 100(octal) to the label ABCD. Subsequent references to ABCD reference location 100(octal). In this example if the location counter were relocatable, the final value of ABCD would be 100(octal)+K, where K is the location of the beginning of the relocatable section in which the label ABCD appears.

A double colon defines the label as global and is accessible to independently assembled modules; thus:

```
ABCD::  MOV     A,B
```

establishes ABCD as a global symbol.

More than one label may appear within a single label field; each label within the field has the same value. For example, if the current location counter is 100(octal), the multiple labels in the statement:

```
ABC:    $DD:    A7.7:   MOV     A,B
```

cause each of the three labels ABC, $DD, and A7.7 to be equated to the value 100(octal). The legal label characters are:

```
A - Z
0 - 9
  .
  $
```

(By convention, $ and . characters are reserved for use in system software symbols.)

The first six characters of a label are significant. An error message is generated if two or more labels share the same first six characters.

A symbol used as a label may not be redefined within the user program. An attempt to redefine a label results in an error flag (M) in the assembly listing.

## 2.1.2 Operator Field

An operator field follows the label field in a statement, and may contain a macro call, an instruction mnemonic, or an assembler directive. The operator may be preceded by none, one or more labels and may be followed by one or more operands and/or a comment. Leading and trailing spaces and tabs are ignored.

When the operator is a macro call, the Assembler inserts the appropriate code to expand the macro. When the operator is an instruction mnemonic, it specifies the intruction to be generated and the action to be performed on any operand(s) which follow. When the operator is an assembler directive, it specifies a certain function or action to be performed during assembly.

An operator is legally terminated by a space, tab, or any non-alphanumeric character (symbol component).

Consider the following examples

```
        MOV     A,B ;space terminates the operator MOV
        MOV     @A,B ;@ terminates the operator MOV
```

A blank operator field is interpreted as a .WORD assembler directive (See section 6.3.2).

## 2.1.3 Operand Field

An operand is that part of a statement which is manipulated by the operator. Operands may be expressions, numbers, or symbolic or macro arguments (within the context of the operation). When multiple operands appear within a statement, each is separated from the next by one of the following characters: comma, tab, space, or paired angle brackets around one or more operands (see section 3.1.1). An operand may be preceded by an operator, label, or other operand and followed by another operand or a comment.

The operand field is terminated by a semicolon when followed by a comment, or by a statement terminator when the operand completes the statement. For example:

```
LABEL:  MOV     A,B             ;COMMENT
```

The tab between MOV and A terminates the operator field and begins the operand field; a comma separates the operands A and B; a semicolon terminates the operand field and begins the comment field.

### 2.1.4 Comment Field

The comment field is optional and may contain any ASCII characters except null, rubout, carriage return, line feed, vertical tab or form feed. All other characters, even special characters with a defined use, are ignored by the Assembler when appearing in the comment field.

The comment field may be preceded by one, any, none or all of the other three field types. Comments must begin with the semicolon character.

Comments do not affect assembly processing or program execution, but are useful in source listings for later analysis, debugging, or documentation purposes.

## 2.2 FORMAT CONTROL

Horizontal or line formatting of the source program is controlled by the space and tab characters. These characters have no effect on the assembly process unless they are embedded within a symbol, number, or ASCII text; or unless they are used as the operator field terminator. Thus, these characters can be used to provide an orderly source program. A statement should be formatted to conform to the DOS/BATCH standard,

```
LABEL:  MOV     (SP)+,TAG; POP VALUE OFF STACK*

LABEL:  MOV     (SP)+,TAG       ;POP VALUE OFF STACK*
```

(See section 6.1.6 for a description of page formatting with respect to macros, and section 6.1.3 for a description of assembly listing output.)

---------------

*Appendix E details code formatting standards used in all DOS/BATCH Monitor software.

# CHAPTER 3

# SYMBOLS AND EXPRESSIONS

This Chapter describes the various components of legal MACRO-11 expressions; the Assembler character set, symbol construction, numbers, operators, terms, and expressions.

## 3.1 CHARACTER SET

The following characters are legal in MACRO-11 source programs:

1. The letters A through Z. Both upper and lower case letters are acceptable, although, upon input, lower case letters are converted to upper case letters. Lower case letters can only be output by sending their ASCII values to the output device. This conversion is not true for .ASCII, .ASCIZ, ' (single quote) or " (double quote) statements if .ENABL LC is in effect.

2. The digits 0 through 9.

3. The characters . (period or dot) and $ (dollar sign) which are reserved for use in system program symbols.

4. The following special characters:

| Character | Designation | Function |
|---|---|---|
| :: <br> == | double colon <br> double equal sign | Either the double colon or double equal sign may be used to define a symbol as a global symbol (refer to section 6.10). |
| : | colon | label terminator |
| = | equal sign | direct assignment indicator |
| % | percent sign | register term indicator |
|  | tab | item or field terminator |
|  | space | item or field terminator |
| # | number sign | immediate expression indicator |
| @ | at sign | deferred addressing indicator |
| ( | left parenthesis | initial register indicator |
| ) | right parenthesis | terminal register indicator |
| , | 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 autoincrement indicator |
| - | minus sign | arithmetic subtraction operator or autodecrement indicator |
| * | asterisk | arithmetic multiplication operator |
| / | slash | arithmetic division operator |
| & | ampersand | logical AND operator |
| ! | exclamation | logical inclusive OR operator |
| " | double quote | double ASCII character indicator |
| ' | single quote | single ASCII character indicator |

| | |
|---|---|
| up arrow or circumflex | universal unary operator, argument indicator |
| backslash | macro numeric argument indicator |

### 3.1.1 Separating and Delimiting Characters

Reference is made in the remainder of the manual to legal separating characters and legal argument delimiters. These terms are defined below in Tables 3-1 and 3-2.

Table 3-1
Legal Separating Characters

| Character | Definition | Usage |
|---|---|---|
| space | one or more spaces and/or tabs | A space is a legal separator only for argument operands. Spaces within expressions are ignored (see section 3.8). |
| , | comma | A comma is a legal separator for both expressions and arguments. |

Table 3-2
Legal Delimiting Characters

| Character | Definition | Usage |
|---|---|---|
| <...> | paired angle brackets | Paired angle brackets are used to enclose an argument, particularly when that argument contains separating characters. Paired angle brackets may be used anywhere in a program to enclose an expression for treatment as a term. |
| ↑\...\ | Up arrow construction where the up arrow character is followed by an argument bracketed by any paired printing characters. | This construction is equivalent in function to the paired angle brackets and is generally used only where the argument contains angle brackets. |

Where argument delimiting characters are used, they must bracket the first (and, optionally, any following) argument(s). The character < and the characters ↑x, where x is any printing character, can be considered unary operators which cannot be immediately preceded by another argument. For example:

```
        .MACRO    TEM <AB>C
```

indicates a macro definition with two arguments, while

```
        .MACRO    TEL C<AB>
```

has only one argument. The closing , or matching character where the up arrow construction is used, acts as a separator. The opening argument delimiter does not act as an argument separator.

Angle brackets can be nested as follows:

```
        <A<B>C>
```

which reduces to:

```
        A<B>C
```

and which is considered to be one argument in both forms.

## 3.1.2 Illegal Characters

A character can be illegal in one of two ways:

1. A character which is not recognized as an element of the MACRO-11 character set is always an illegal character and causes immediate termination of the current line at that point, plus the output of an error flag (I) in the assembly listing. For example:

   ```
   LABEL←*A:  MOV  A,B
   ```

   Since the backarrow is not a recognized character, the entire line is treated as a:

   ```
           .WORD   LABEL
   ```

   statement and is flagged in the listing.

2. A legal MACRO-11 character may be illegal in context. Such a character generates a Q error on the assembly listing.

## 3.1.3 Operator Characters

Legal unary operators under MACRO-11 are as follows:

| Unary Operator | Explanation | | Example |
|---|---|---|---|
| + | plus sign | +A | (positive value of A, equivalent to A) |
| - | minus sign | -A | (negative 2's complement value of A) |
| ↑ | up arrow, universal unary operator | ↑F3.0 | (interprets 3.0 as a 1-word floating-point number) |

(this usage is described in greater detail in sections 6.4.2 and 6.6.2).

| | |
|---|---|
| ↑C24 | (interprets the 1's complement value of 24(octal); 18, not 24) |
| ↑D127 | (interprets 127 as a decimal number) |
| ↑O34 | (interprets 34 as an octal number) |
| ↑B11000111 | (interprets 11000111 as a binary value) |

The unary operators as described above can be used adjacent to each other in a term. For example:

```
        -%5
        ↑C↑O12
```

Legal binary operators under MACRO-11 are as follows:

| Binary Operator | Explanation | Example |
|---|---|---|
| + | addition | A+B |
| - | subtraction | A-B |
| * | multiplication | A*B (16-bit product returned) |
| / | division | A/B (16-bit quotient returned) |
| & | logical AND | A&B |
| ! | logical inclusive OR | A!B |

All binary operators have the same priority. Items can be grouped for evaluation within an expression by enclosure in angle brackets. Terms in angle brackets are evaluated first, and remaining operations are performed left to right. For example:

```
        .WORD   1+2*3           ;IS 11 OCTAL
        .WORD   1+<2*3>         ;IS 7 OCTAL
```

## 3.2 MACRO-11 SYMBOLS

There are three types of symbols: permanent, user-defined and macro. MACRO-11 maintains three types of symbol tables: the Permanent Symbol Table (PST), the User Symbol Table (UST), and the Macro Symbol Table (MST). The PST contains all the permanent symbols and is part of the MACRO-11 Asembler load module. The UST and MST are constructed as the source program is assembled; user-defined symbols are added to the table as they are encountered.

### 3.2.1 Permanent Symbols

Permanent symbols consist of the instruction mnemonics and assembler directives (Chapter 6 and 7, Appendix B). These symbols are a permanent part of the Assembler and need not be defined before being used in the source program.

### 3.2.2 User-Defined and Macro Symbols

User-defined symbols are those used as labels (section 2.1.1) or defined by direct assignment (section 3.3). These symbols are added to the User Symbol Table as they are encountered during the first pass of the assembly. Macro symbols are those symbols used as macro names (section 7.1). These symbols are added to the Macro Symbol Table as they are encountered during the assembly.

User-defined and macro symbols can be composed of alphanumeric characters, dollar signs, and periods only; any other character is illegal.

The $ and . characters are reserved for system software symbols (e.g., READ$, a system macro) and should not be inserted as a user-defined or macro symbol.

The following rules apply to the creation of user-defined and macro symbols:

1. The first character must not be a number (except in the case of local symbols, see section 3.5).

2. Each symbol must be unique within the first six characters.

3. A symbol can be written with more then six legal characters, but the seventh and subsequent characters are only checked for legality, and are not otherwise recognized by the Assembler.

4. Spaces, tabs, and illegal characters must not be embedded within a symbol.

The value of a symbol depends upon its use in the program. A symbol in the operator field may be any one of the three symbol types. To determine the value of the symbol, the Assembler searches the three symbol tables in the following order:

1. Macro Symbol Table

2. Permanent Symbol Table

3. User-Defined Symbol Table

A symbol found in the operand field is sought in the

1. User-Defined Symbol Table

2. Permanent Symbol Table

in that order. The Assembler never expects to find a macro name in an operand field.

These search orders allow redefinition of Permanent Symbol Table entries as user-defined or macro symbols. The same name can also be assigned to both a macro and a label.

User-defined symbols are either internal or external (global). All user-defined symbols are internal unless they remain undefined internally or unless explicitly defined as being global with the .GLOBL directive or by the double-colon, or double-equal sign (see Section 6.10).

Global symbols provide links between object modules. A global symbol which is defined as a label is generally called an entry point (to a section of code). Such symbols are referenced from other object modules to transfer control throughout the load module (which may be composed of a number of object modules).

Since MACRO-11 provides program sectioning capabilities (section 6.9), two types of internal symbols must be considered:

1. Symbols that belong to the current program section; and

2. Symbols that belong to other program sections.

In both cases, the symbol must be defined within the current assembly; the significance of the distinction is critical in evaluating expressions involving type (2) above (see section 3.9).

## 3.3 DIRECT ASSIGNMENT

A direct assignment statement associates a symbol with a value. When a direct assignment statement defines a symbol for the first time, that symbol is entered into the user symbol table. A symbol may be redefined by assigning a new value to a previously defined symbol. The latest assigned value replaces any previous value assigned to a symbol.

The general format for a direct assignment statement is:

```
symbol = expression

or

symbol == expression
```

which also defines symbol as a global definition.

Symbols take on the relocatable or absolute attribute of their defining expression. However, if the defining expression is global, the symbol is not global unless explicitly defined as such in a .GLOBL directive, by a label delimited by a double colon or by the double equal sign (see section 6.10). Global references in an expression assigned to a symbol are illegal, and are flagged with an A error flag.

For example:

```
A = 1                           ;THE SYMBOL A IS EQUATED TO THE
                                ;VALUE 1.

B = 'A-1&MASKLOW                ;THE SYMBOL B IS EQUATED TO THE
                                ;VALUE OF THE EXPRESSION

C:     D = 3                    ;THE SYMBOL D IS EQUATED TO 3.

E:     MOV     #1,ABLE          ;LABELS C AND E ARE EQUATED TO THE
                                ;LOCATION OF THE MOV COMMAND
```

The following conventions apply to direct assignment statements:

1. An equal sign (=) or double equal (==) must separate the symbol from the expression defining the symbol value.

2. A direct assignment statement is usually placed in the label field and may be followed by a comment.

3. Only one symbol can be defined in a single direct assignment statement.

4. Only one level of forward referencing is allowed.

Example of two levels of forward referencing (illegal):

```
X = Y

Y = Z

Z = 1
```

## 3.4 REGISTER SYMBOLS

The eight general registers of the PDP-11 are numbered 0 through 7 and can be expressed in the source program as:

```
        %0
        %1
         .
         .
         .
        %7
```

where the digit indicating the specific register can be replaced by any legal term which can be evaluated during the first assembly pass.

It is recommended that the programmer use symbolic names for all register references. Unless the .DSABL REG statement has been encountered, the definitions as shown in the following example are defined by default, or, a register symbol may be defined in a direct assignment statement, among the first statements in the program. The defining expression of a register symbol must be absolute. For example:

```
 Line     Octal
Number  Expansion        Source Code                       Comments
  1                        .SBTTL  SECTOR INITIALIZATION
  2
  3        000000'         .CSECT  IMPURE           ;IMPURE STORAGE AREA
  4 000000        IMPURE:
  5        000000'         .CSECT  IMPPAS           ;CLEARED EACH PASS
  6 000000        IMPPAS:
  7        000000'         .CSECT  IMPLIN           ;CLEARED EACH LINE
  8 000000        IMPLIN:
  9
 10        000000'         .CSECT  XCTPRG           ;PROGRAM INITIALIZATION CODE
 11  00000        XCTPRG:
 12  00000 012700          MOV     #IMPURE,R0
       000000'
 13  00004 005020 1$:      CLR     (R0)+            ;CLEAR IMPURE AREA
 14  00006 022700          CMP     #IMPTOP,R0
       000240'
 15  00012 101374          BHI     1$
 16
 17        000000'         .CSECT  XCTPAS           ;PASS INITIALIZATION CODE
 18  00000        XCTPAS:
 19  00000 012700          MOV     #IMPPAS,R0
       000000'
 20  00004 005020 1$:      CLR     (R0)+            ;CLEAR IMPURE PART
 21  00006 022700          CMP     #IMPTOP,R0
       000240'
 22  00012 101374          BHI     1$
 23
 24        000000'         .CSECT  XCTLIN           ;LINE INITIALIZATION CODE
 25  00000        XCTLIN:
 26  00000 012700          MOV     #IMPLIN,R0
       000000'
 27  00004 005020 1$:      CLR     (R0)+
 28  00006 022700          CMP     #IMPTOP,R0
       000240'
 29  00012 101374          BHI     1$
 30
 31        000000'         .CSECT  MIXED            ;MIXED MODE SECTOR
```

Figure 3-3

Assembly Source Listing of MACRO-11 Code Showing Local Symbol Blocks

## 3.6 ASSEMBLY LOCATION COUNTER

The period (.) is the symbol for the assembly location counter. When used in the operand field of an instruction, it represents the address of the first word of the instruction. When used in the operand field of an assembler directive, it represents the address of the current byte or word. For example:

```
A:      MOV     #.,R0               ;. REFERS TO LOCATION A,
                                    ;I.E., THE ADDRESS OF THE
                                    ;MOV INSTRUCTION.
```

(# is explained in section 5.9.)

At the beginning of each assembly pass, the Assembler clears the location counter. Normally, consecutive memory locations are assigned to each byte of object data generated. However, the location where the object data is stored may be changed by a direct assignment altering the location counter:

```
          .=expression
```

Similar to other symbols, the location counter symbol has a mode associated with it, either absolute or relocatable. However, the mode cannot be external. The existing mode of the location counter cannot be changed by using a defining expression of a different mode.

The mode of the location counter symbol can be changed by the use of the .ASECT,.CSECT or .PSECT directives as explained in section 6.9.

The expression defining the location counter must not contain forward references or symbols that vary from one pass to another.

Examples:

```
        .ASECT

.=500                               ;SET LOCATION COUNTER TO
                                    ;ABSOLUTE 500

FIRST:  MOV     .+10,COUNT          ;THE LABEL FIRST HAS THE VALUE
                                    ;500(OCTAL)
                                    ;.+10 EQUALS 510(OCTAL). THE
                                    ;CONTENTS OF THE LOCATION
                                    ;510(OCTAL) WILL BE DEPOSITED
                                    ;IN LOCATION COUNT.

.=520                               ;THE ASSEMBLY LOCATION COUNTER
                                    ;NOW HAS A VALUE OF
                                    ;ABSOLUTE 520(OCTAL).

SECOND: MOV     .,INDEX             ;THE LABEL SECOND HAS THE
                                    ;VALUE 520(OCTAL)
                                    ;THE CONTENTS OF LOCATION
                                    ;520(OCTAL), THAT IS, THE BINARY
                                    ;CODE FOR THE INSTRUCTION
                                    ;ITSELF, WILL BE DEPOSITED IN
                                    ;LOCATION INDEX.
```

## 3.9 EXPRESSIONS

Expressions are combinations of terms joined together by binary operators and which reduce to a 16-bit value. The operands of a .BYTE directive (see section 6.3.1) are evaluated as word expressions before truncation to the low-order eight bits. Prior to truncation, the high-order byte must be zero or all ones (when byte value is negative, the sign bit is propagated). The evaluation of an expression includes the evaluation of the mode of the resultant expression; that is, absolute, relocatable or external. Expression modes are further defined below.

Expressions are evaluated left to right with no operator hierarchy rules except that unary operators take precedence over binary operators. A term preceded by a unary operator can be considered as containing that unary operator. (Terms are evaluated, where necessary, before their use in expressions.) Multiple unary operators are valid and are treated as follows:

```
          -+-A
```

is equivalent to:

```
          -<+<-A>>
```

A missing term, expression, or external symbol is interpreted as a zero. A missing operator is interpreted as +. A Q error flag is generated for each missing term or operator. For example (here TAG is OR'ed with LA +177777):

```
          TAG ! LA 177777
```

is evaluated as

```
          TAG ! LA+177777
```

with a Q error flag on the assembly listing line.

The value of an external expression is the value of the absolute part of the expression; e.g., EXTERNAL+A has a value of A. This is modified by LINK to become EXTERNAL+A.

Expressions, when evaluated, are either absolute, relocatable, or external. For the programmer writing position-independent code, the distinction is important.

1. An expression is absolute if its value is fixed. An expression whose terms are numbers and ASCII conversions will have an absolute value. A relocatable expression minus a relocatable term, where both items belong to the same program section, is also absolute.

2. An expression is relocatable if its value is fixed relative to a base address but will have an offset value added at Task Build time. Expressions whose terms contain labels defined in relocatable sections and periods, (in relocatable sections) will have a relocatable value.

3. An expression is external (or global) if its value is only partially defined during assembly and its definition is completed at LINK linking time. An expression whose terms contain a global symbol not defined in the current program is an external expression. External expressions have relocatable values at execution time, if the global symbol is defined as being relocatable; or absolute, if the global symbol is defined as absolute.

# CHAPTER 4

# RELOCATION AND LINKING

The output of the MACRO-11 Asembler is an object module which must be processed by LINK before loading and execution. (See DOS/BATCH Linker (LINK) Programmer's Manual for details.) LINK essentially fixes (i.e., makes absolute) the values of external or relocatable symbols and turns the object module into a load module.

To enable the the Linker Program to fix the value of an expression, the Assembler issues certain directives to LINK, together with required parameters. In the case of relocatable expressions, LINK adds the base of the associated relocatable section (the location in memory of relocatable 0) to the value of the relocatable expression provided by the Assembler. In the case of an external expression, the value of the external term in the expression is determined by LINK (since the external symbol must be defined in one of the other object modules which are being linked together) and adds it to the value of the external expression provided by the Assembler.

All instructions that are to be modified (as described in the previous paragraph) are marked with an apostrophe in the assembly listing (see also section 1.2). Thus, the binary text output looks like the following:

```
005065  CLR     EXTERNAL(5)
000000'                             ;VALUE OF EXTERNAL SYMBOL
                                    ;ASSEMBLED ZERO; WILL BE
                                    ;MODIFIED BY LINK.

005065  CLR     EXTERNAL+6(5)       ;THE ABSOLUTE PORTION OF THE
000006'                             ;EXPRESSION (000006) IS ADDED
                                    ;BY LINK TO THE VALUE
                                    ;OF THE EXTERNAL SYMBOL

005065  CLR     RELOCATABLE(5)      ;ASSUMING WE ARE IN A
                                    ;RELOCATABLE
000040'                             ;SECTION AND THE VALUE OF
                                    ;RELOCATABLE SYMBOL IS RELOCATABLE 40
                                    ;LINK WILL ADD
                                    ;THE RELOCATION BIAS TO 40
```

# CHAPTER 5

# ADDRESSING MODES

The program counter (PC, register 7 of the eight general registers) always contains the address of the next word to be fetched; i.e., the address of the next instruction to be executed, or the second or third word of the current instruction.

In order to understand how the address modes operate and how they assemble, the action of the program counter must be understood. The key rule is:

> Whenever the processor implicitly uses the program counter to fetch a word from memory, the program counter is automatically incremented by two after the fetch.

That is, when an instruction is fetched, the PC is incremented by two, so that it is pointing to the next word in memory; and, if an instruction uses indexing (sections 5.7, 5.9 and 5.11), the processor uses the program counter to fetch the base from memory. Hence, using the rule above, the PC increments by two, and now points to the next word.

1. Let E be any expression as defined in Chapter 3.

2. Let R be a register expression. This is any expression containing a term preceded by a % character or a symbol previously equated to such a term.

   Examples:

   ```
   R0=%0                          ;GENERAL REGISTER 0
   R1=R0+1                        ;GENERAL REGISTER 1
   R2=1+%1                        ;GENERAL REGISTER 2
   ```

3. Let ER be a register expression or an expression in the range 0 to 7 inclusive.

4. Let A be a general address specification which produces a 6-bit mode address field as described in sections 3.1 and 3.2 of the PDP-11 Processor Handbook (both 11/40 and 11/45 versions).

   The addressing specification, A, can be explained in terms of E, R, and ER as defined above. Each is illustrated with the single operand instruction CLR or double operand instruction MOV.

## 5.1 REGISTER MODE

The register contains the operand.

Format for A: R

Examples:

```
R0=%0                                  ;DEFINE R0 AS REGISTER 0
        CLR     R0                     ;CLEAR REGISTER 0
```

## 5.2 REGISTER DEFERRED MODE

The register contains the address of the operand.

Format for A: @R or (ER)

Examples:

```
        CLR     @R1                    ;BOTH INSTRUCTIONS CLEAR
        CLR     (R1)                   ;THE WORD AT THE ADDRESS
                                       ;CONTAINED IN REGISTER 1
```

## 5.3 AUTOINCREMENT MODE

The contents of the register are incremented immediately after being used as the address of the operand. (See note below.)

Format for A: (ER)+

Examples:

```
        CLR     (R0)+                  ;EACH INSTRUCTION CLEARS
        CLR     (R0+3)+                ;THE WORD AT THE ADDRESS
        CLR     (R2)+                  ;CONTAINED IN THE SPECIFIED
                                       ;REGISTER AND INCREMENTS
                                       ;THAT REGISTER'S CONTENTS
                                       ;BY TWO
```

***NOTE***

Both JMP and JSR instructions using non-deferred autoincrement mode, autoincrement the register before its use on the PDP-11/20 (but not on the PDP-11/45 or 11/05). In double operand instructions of the addressing form Rn or Rn,-(Rn) where the source and destination registers are the same, the source operand is evaluated as the autoincremented or autodecremented value; but the destination register, at the time it is used, still contains the originally intended effective address. In the following two examples, as executed on the PDP-11/20, R0 originally contains 100.

```
        MOV     R0,(R0)+               ;THE QUANTITY 102 IS MOVED
                                        ;TO LOCATION 100

        MOV     R0,-(R0)               ;THE QUANTITY 76 IS MOVED
                                        ;TO LOCATION 76
```

The use of these forms should be avoided as they are not compatible with the PDP-11/05 and 11/45.

A Z error code is printed with each instruction which is not compatible among all members of the PDP-11 family. This is merely a warning code.

## 5.4 AUTOINCREMENT DEFERRED MODE

The register contains the pointer to the address of the operand. The contents of the register are incremented after being used.

Format for A: @(ER)+

Example:

```
        CLR     @(R3)+          ;CONTENTS OF REGISTER 3 POINT
                                ;TO ADDRESS OF WORD TO BE
                                ;CLEARED BEFORE BEING
                                ;INCREMENTED BY TWO
```

## 5.5 AUTODECREMENT MODE

The contents of the register are decremented before being used as the address of the operand (see note under autoincrement mode).

Format for A: -(ER)

Examples:

```
        CLR     -(R0)           ;DECREMENT CONTENTS OF
                                ;REGISTERS
        CLR     -(R0+3)         ;0, 3 AND 2 BY TWO BEFORE
                                ;USING THEM
        CLR     -(R2)           ;AS ADDRESSES OF A WORD TO BE
                                ;CLEARED.
```

## 5.6 AUTODECREMENT DEFERRED MODE

The contents of the register are decremented before being used as the pointer to the address of the operand.

Format for A: @-(ER)

Example:

```
        CLR     @-(R2)          ;DECREMENT CONTENTS OF
                                ;REGISTER 2 BY TWO BEFORE
                                ;USING AS POINTER
                                ;TO ADDRESS OF WORD TO BE
                                ;CLEARED.
```

## 5.7 INDEX MODE

The value of an expression E is stored as the second or third word of the instruction. The effective address is calculated as the value of E plus the contents of register ER. The value E is called the base.

Format for A:    E(ER)

Examples:
```
        CLR     X+2(R1)         ;EFFECTIVE ADDRESS IS X+2 PLUS
                                ;THE CONTENTS OF REGISTER 1.
        CLR     -2(R3)          ;EFFECTIVE ADDRESS IS -2 PLUS
                                ;THE CONTENTS OF REGISTER 3.
```

## 5.8 INDEX DEFERRED MODE

An expression plus the contents of a register gives the pointer to the address of the operand.

Format for A:    @E(ER)

Example:
```
        CLR     @114(R4)        ;IF REGISTER 4 HOLDS 100 AND
                                ;LOCATION 214 HOLDS 2000,
                                ;LOCATION 2000 IS CLEARED.
```

## 5.9 IMMEDIATE MODE

The immediate mode allows the operand itself to be stored as the second or third word of the instruction. It is assembled as an autoincrement of register 7, the PC.

Format for A:    #E

Examples:
```
        MOV     #100,R0         ;MOVE AN OCTAL 100 TO REGISTER
                                ;0
        MOV     #X, R0          ;MOVE THE VALUE OF SYMBOL X TO
                                ;REGISTER 0
```

The operation of this mode is explained as follows:

The statement MOV #100,R3 assembles as two words. These are:

```
        0 1 2 7 0 3

        0 0 0 1 0 0
```

Just before this instruction is fetched and executed, the PC points to the first word of the instruction. The processor fetches the first word and increments the PC by two. The source operand mode is 27 (autoincrement the PC). Thus, the PC is used as a pointer to fetch the operand (the second word of the instruction) before being incremented by two, to point to the next instruction.

## 5.10 ABSOLUTE MODE

Absolute mode is the equivalent of immediate mode deferred. @#E specifies an absolute address which is stored in the second or third word of the instruction. Absolute mode is assembled as an autoincrement deferred of register 7, the PC.

Format for A: @#E

Examples:

```
        MOV     @#100,R0          ;MOVE THE VALUE OF THE
                                  ;CONTENTS
                                  ;OF LOCATION 100 TO REGISTER R0.
        CLR     @#X               ;CLEAR THE CONTENTS OF THE
                                  ;LOCATION WHOSE ADDRESS IS X.
```

## 5.11 RELATIVE MODE

Relative mode is the normal mode for memory references.

Format for A: E

Examples:

```
        CLR     100               ;CLEAR LOCATION 100.
        MOV     X,Y               ;MOVE CONTENTS OF LOCATION X
                                  ;TO LOCATION Y.
```

Relative mode is assembled as index mode, using register 7, the PC, as the index register. The base of the address calculation, which is stored in the second or third word of the instruction, is not the address of the operand (as in index mode), but the number which, when added to the PC, becomes the address of the operand. Thus, the base is X-PC, which is called an offset. The operation is explained as follows:

If the statement MOV 100,R3 is assembled at absolute location 20, the assembled code is:

```
          Location 20:          0 1 6 7 0 3
          Location 22:          0 0 0 0 5 4
```

The processor fetches the MOV instruction and adds two to the PC so that it points to location 22. The source operand mode is 67; that is, indexed by the PC. To pick up the base, the processor fetches the word pointed to by the PC and adds two to the PC. The PC now points to location 24. To calculate the address of the source operand, the base is added to the designated register. That is, BASE+PC=54+24=100, the operand address.

Since the Assembler considers "." as the address of the first word of the instruction, an equivalent index mode statement would be:

```
        MOV     100-.-4(PC),R3
```

This mode is called relative because the operand address is calculated relative to the current PC. The base is the distance or offset (in bytes) between the operand and the current PC. If the operator and its operand are moved in memory so that the distance between the operator and data remains constant, the instruction will operate correctly anywhere in core.

## 5.12 RELATIVE DEFERRED MODE

Relative deferred mode is similar to relative mode, except that the expression, E, is used as the pointer to the address of the operand.

Format for A: @E

Example:
```
        MOV     @X,R0           ;MOVE THE CONTENTS OF THE
                                ;LOCATION WHOSE ADDRESS IS IN
                                ;X INTO REGISTER 0.
```

## 5.13 TABLE OF MODE FORMS AND CODES

Each instruction takes at least one word. Operands of the first six forms listed below do not increase the length of an instruction. Each operand in one of the other modes, however, increases the instruction length by one word.

| Form | Mode | Meaning |
|---|---|---|
| R | 0n | Register mode |
| @R or (ER) | 1n | Register deferred mode |
| (ER)+ | 2n | Autoincrement mode |
| @(ER)+ | 3n | Autoincrement deferred mode |
| -(ER) | 4n | Autodecrement mode |
| @-(ER) | 5n | Autodecrement deferred mode |

where n is the register number.

Any of the following forms adds one word to the instruction length:

| Form | Mode | Meaning |
|---|---|---|
| E (ER) | 6n | Index mode |
| @E(ER) | 7n | Index deferred mode |
| #E | 27 | Immediate mode |
| @#E | 37 | Absolute memory reference mode |
| E | 67 | Relative mode |
| @E | 77 | Relative deferred reference mode |

where n is the register number. Note that in the last four forms, register 7 (the PC) is referenced.

***NOTE***

An alternate form for @R is (ER). However, the form @(ER) is equivalent to @0(ER).

The form @#E differs from the form E in that the second or third word of the instruction contains the absolute address of the operand rather than the relative distance between the operand and the PC. Thus, the instruction CLR @#100 clears absolute location 100 even if the instruction is moved from the point at which it was assembled. See the description of the .ENABLE AMA function in section 6.2, which directs the assembly of all relative mode addresses as absolute mode addresses.

## 5.14 BRANCH INSTRUCTION ADDRESSING

The branch instructions are 1-word instructions. The high byte contains the op code and the low byte contains an 8-bit signed offset (seven bits plus sign) which specifies the branch address relative to the PC. The hardware calculates the branch address as follows:

1. Extend the sign of the offset through bits 8-15.
2. Multiply the result by 2. This creates a word offset rather than a byte offset.
3. Add the result to the PC to form the final branch address.

The Assembler performs the reverse operation to form the byte offset from the specified address. Remember that when the offset is added to the PC, the PC is pointing to the word following the branch instruction; hence the factor -2 in the calculation.

Byte offset = (E-PC)/2 truncated to eight bits.

Since PC = .+2, we have

Byte offset = (E-.-2)/2 truncated to eight bits.

***NOTE***

> It is illegal to branch to a location specified as an external symbol, or to a relocatable symbol from within an absolute section, or to an absolute symbol or a relocatable symbol or another program section from within a relocatable section.

The EMT and TRAP instructions do not use the low-order byte of the word. This allows information to be transferred to the trap handlers in the low-order byte. If EMT or TRAP is followed by an expression, the value is put into the low-order byte of the word. However, if the expression is too big ( 377(octal)) it is truncated to eight bits and a T error flag is generated.

# CHAPTER 6

# GENERAL ASSEMBLER DIRECTIVES

## 6.1 LISTING CONTROL DIRECTIVES

### 6.1.1 .LIST and .NLIST

Listing options can be specified in the text of a MACRO-11 program through the .LIST and .NLIST directives. These are of the form:

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

where: arg represents one or more optional arguments.

When used without arguments, the listing directives alter the listing level count. The listing level count causes the listing to be suppressed when it is negative. The count is initialized to zero, incremented for each .LIST and decremented for each .NLIST. For example:

```
        .MACRO  LTEST           ;LIST TEST
; A-THIS LINE SHOULD LIST
        .NLIST
; B-THIS LINE SHOULD NOT LIST
        .NLIST
; C-THIS LINE SHOULD NOT LIST
        .LIST
; D-THIS LINE SHOULD NOT LIST (LEVEL NOT BACK TO ZERO)
        .LIST
; E-THIS LINE SHOULD LIST (LEVEL BACK TO ZERO)
        .ENDM

        LTEST                   ;CALL THE MACRO

; A-THIS LINE SHOULD LIST
        .NLIST
        .LIST
; E-THIS LIST SHOULD LIST (LEVEL BACK TO ZERO)
```

The primary purpose of the level count is to allow macro expansions to be selectively listed and yet exit with the level returned to the status current during the macro call.

The use of arguments with the listing directives does not affect the level count; however, use of .LIST and .NLIST can be used to override the current listing control. For example:

```
        .MACRO XX
          .
          .
          .
        .LIST                   ;LIST NEXT LINE
X=.
        .NLIST                  ;DO NOT LIST REMAINDER
          .
          .                     ;OF MACRO EXPANSION
          .
        .ENDM
        .NLIST ME               ;DO NOT LIST MACRO EXPANSIONS
        XX
        .LIST                   ;LIST NEXT LINE
X=.
```

Allowable arguments for use with the listing directives are as follows (these arguments can be used singly or in combination)

| Argument | Default | Function |
|---|---|---|
| SEQ | list | Controls the listing of source line sequence numbers. Error flags are normally printed on the line preceding the questionable source statement. |
| LOC | list | Controls the listing of the location counter (this field would not normally be suppressed). |

| Argument | Default | Function |
|---|---|---|
| BIN | list | Controls the listing of generated binary code. |
| BEX | list | Controls listing of binary extensions; that is, those locations and binary contents beyond the first binary word (per source statement). This is a subset of the BIN argument. |
| SRC | list | Controls the listing of the source code. |
| COM | list | Controls the listing of comments. This is a subset of the SRC argument and can be used to reduce listing time and/or space where comments are unnecessary. |
| MD | list | Controls listing of macro definitions and repeat range expansions. |
| MC | list | Controls listing of macro calls and repeat range expansions. |
| ME | no list | Controls listing of macro expansions. |
| MEB | no list | Controls listing of macro expansion binary code. A LIST MEB causes only those macro expansion statements producing binary code to be listed. This is a subset of the ME argument. |
| CND | list | Controls the listing of unsatisfied conditions and all .IF and .ENDC statements. This argument permits conditional assemblies to be listed without including unsatisfied code. |
| LD | no list | Control listing of all listing directives having no arguments (those used to alter the listing level count). |
| TOC | list | Control listing of table of contents on pass 1 of the assembly (see section 6.1.4 describing the .SBTTL directive). The full assembly listing is printed during pass 1 of the assembly. |
| TTM | Console mode | Control listing output format. The TTM argument (the default case) causes output lines to be truncated to 72 characters. Binary code is printed with the binary extensions below the first binary word. The alternative (.NLIST TTM) to Teletype mode is line printer mode, which is shown in Figure 6-1. |
| SYM | list | Controls the listing of the symbol table for the assembly. |

An example of an assembly listing as sent to a 132-column line printer is shown in Figure 6-1. Notice that binary extensions for statements generating more than one word are spread horizontally on the source line. An example of an assembly listing as sent to a teleprinter is shown in Figure 6-2. Notice that binary extensions for statements generating more than one word are printed on subsequent lines.

The listing options can also be specified through switches on the listing file specification in the command string to the MACRO-11 Assembler. These switches are

```
/LI:arg
/NL:arg
```

|  |  |  |
|---|---|---|
| where: | arg | is any one or more of the arguments defined in the .LIST and .NLIST directive. |

```
MACRO   VØØ3A,1 24-MAY-72       MACRO VØØ3A,1 26-MAY-72 ØØ:Ø6 PAGE 28
ASSEMBLER PROPER

     1 ØØ1766                              GETLIN:                               ;GET AN INPUT LINE
     2 ØØ1766                                        SAVREG
     3 ØØ1772  Ø167ØØ  ØØØØ2Ø'             1$:       MOV      FFCNT,RØ           ;ANY RESERVED FF'S?
     4 ØØ1776  ØØ142Ø                                BEQ      31$                ;  NO
     5 ØØ2ØØØ  Ø6ØØ67  ØØØØ22'                       ADD      RØ,PAGNUM          ;YES, UPDATE PAGE NUMBER
     6 ØØ2ØØ4  Ø12767  177777  ØØØØ26'               MOV      #-1,PAGEXT
     7 ØØ2Ø12  ØØ5Ø67  ØØØØ12'                       CLR      LINNUM             ;INIT NEW CREF SEQUENCE
     8 ØØ2Ø16  ØØ5Ø67  ØØØØ2Ø'                       CLR      FFCNT
     9 ØØ2Ø22  ØØ5Ø67  ØØØØ16'                       CLR      SEQEND
    1Ø ØØ2Ø26  ØØ5767  ØØØØØØ'                       TST      PASS
    11 ØØ2Ø32  ØØ14Ø2                                BEQ      31$
    12 ØØ2Ø34  ØØ5Ø67  ØØØØ1Ø'                       CLR      LPPCNT
    13 ØØ2Ø4Ø  Ø127Ø2  ØØ1712'             31$:      MOV      #LINBUF,R2
    14 ØØ2Ø44  Ø10267  ØØØØ12'                       MOV      R2,LCBEGL          ;SEAT UP BEGINNING
    15 ØØ2Ø5Ø  Ø12767  ØØ2116' ØØØØ14'               MOV      #LINEND,LCENDL     ;  AND END OF LINE MARKERS
    17 ØØ2Ø56  ØØ5767  ØØØ2ØØ'                       TST      SMLCNT             ;IN SYSTEM MACRO?
    18 ØØ2Ø62  ØØ1145                                BNE      4Ø$                ;  YES, SPECIAL
    21 ØØ2Ø64  Ø167Ø1  ØØ2214'                       MOV      MSBMRP,R1          ;ASSUME MACRO IN PROGRESS
    22 ØØ2Ø7Ø  ØØ1166                                BNE      1Ø$                ;BRANCH IF SO
    24 ØØ2Ø72  Ø127Ø1  ØØØ756'                       MOV      #SRCBUF,R1
    25 ØØ2Ø76                                        .WAIT    #SRCLNK
    26 ØØ21Ø4  ØØ5267  ØØØØ12'                       INC      LINNUM
    27 ØØ211Ø  1167ØØ  ØØØ753'                       MOVB     SRCHDR+3,RØ        ;GET CODE BYTE
    28 ØØ2114  Ø327ØØ  ØØØØ47                        BIT      #Ø47,RØ            ;ANYTHING BAD?
    29 ØØ212Ø  ØØ14Ø3                                BEQ      32$                ;  NO
    3Ø ØØ2122                                        ERROR    L                  ;YES, ERROR
    31 ØØ213Ø  1061ØØ                      32$:      ROLB     RØ                 ;EOF?
    32 ØØ2132  1ØØØ14                                BPL      2$                 ;  NO
    33 ØØ2134  Ø56767  ØØØØØ6' ØØØØØ4'               BIS      CSISAV,ENDFLG
    34 ØØ2142  ØØ1ØØ3                                BNE      34$
```

Example of MACRO-11 Line Printer Listing
(132 column line printer)

FIGURE 6-1

## 6.1.2 Page Headings

The MACRO-11 Assembler outputs each page in the format shown in Figure 6-2 (Teletype listing). On the first line of each listing page the Assembler prints (from left to right):

1. Title taken from .TITLE directive

2. Assembler version identification

3. Date

4. Time-of-day

5. Page number

The second line of each listing page contains the subtitle text specified in the last encountered .SBTTL directive.

## 6.1.3 .TITLE

The .TITLE directive is used to assign a name to the object module. The name is the first symbol following the directive and must be six Radix-50 characters or less (any characters beyond the first six are ignored). Non Radix-50 characters are not acceptable. For example:

```
        .TITLE PROG TO PERFORM DAILY ACCOUNTING
```

causes the object module of the assembled program to be named PROG (this name is distinguished from the filename of the object module specified in the command string to the Assembler). The name of the object module appears in the LINK load map and on the listing.

If there is no .TITLE statement, the default name assigned to the object module is

```
        .MAIN.
```

The first tab or space following the .TITLE directive is not considered part of the object module name or header text, although subsequent tabs and spaces are significant.

If there is more than one .TITLE directive, the last .TITLE directive in the program conveys the name of the object module.

## 6.1.4 .SBTTL

The .SBTTL directive is used to provide the elements for a printed table of contents of the assembly listing. The text following the directive is printed as the second line of each of the following assembly listing pages until the next occurrence of a .SBTTL directive. For example:

```
        .SBTTL CONDITIONAL ASSEMBLIES
```

The text:

```
          CONDITIONAL ASSEMBLIES
```

is printed as the second line of each of the following assembly listing pages.

During pass 1 of the assembly process, MACRO-11 automatically prints a table of contents for the listing containing the line sequence number and text of each .SBTTL directive in the program. Such a table of contents is inhibited by specifying the /NL:TOC switch option to the assembly listing file specification (or a .NLIST TOC directive within the source). For example:

```
          #OBJFIL,LISTM/NL:TOC=SRCFIL
```

In this case the table of contents normally generated prior to the assembly listing is inhibited.

An example of the table of contents is shown in Figure 6-3. Note that the first word of the subtitle heading is not limited to six characters since it is not a module name.

## 6.1.5 .IDENT

The .IDENT directive provides another means of labeling the object module produced as a result of a MACRO-11 assembly. In addition to the name assigned to the object module with the .TITLE directive, a character string (up to six characters, treated like a RAD50 string) can be specified between paired delimiters. For example:

```
        .IDENT  /V005A/
```

Table 6-1

Functions: Symbolic Arguments

| Argument | Default | Function |
|---|---|---|
| ABS | disable | Enabling of this function produces absolute binary output; i.e., input to the Paper Tape Software System Absolute Loader. |
| AMA | disable | Enabling of this function directs the assembly of all relative addresses (address mode 67) as absolute addresses (address mode 37). This switch is useful during the debugging phase of program development. |
| CDR | disable | The statement .ENABL CDR causes source columns 73 and greater to be treated as comment. This accommodates sequence numbers in card columns 72-80. |
| FPT | disable | Enabling of this function causes floating point truncation, rather than rounding, as is otherwise performed. .DSABL FPT returns to floating point rounding mode. |
| LC | disable | Enabling of this function causes the Assembler to accept lower case ASCII input instead of converting it to upper case. |
| LSB | disable | Enable or disable a local symbol block. While a local symbol block is normally entered by encountering a new symbolic label or .PSECT directive, .ENABL LSB forces a local symbol block which is not terminated until a label or .PSECT directive following the .DSABL LSB statement is encountered. (Refer to Figure 6-4.) |
| PNC | enable | The statement .DSABL PNC inhibits binary output until an .ENABL PNC is encountered. |

TABLE 6-1 (Cont'd)

| Argument | Default | Function |
|---|---|---|
| REG | enable | The statement .DSABL REG inhibits the default register definitions. That is, until .DSABL REG is seen, the following code is implied as being present:<br><br>R0=%0<br>R1=%1<br>R2=%2<br>R3=%3<br>R4=%4<br>R5=%5<br>SP=%6<br>PC=%7<br><br>The .ENABL REG statement may be used to re-enable these definitions. Such use is not recommended. |
| GBL | enable | The statement .DSABL GBL inhibits attempts to resolve references which remain undefined at the end of pass 1, as being global references. |

An incorrect argument causes the directive containing it to be flagged as an error.

Once a program has been written using these functions, or not using them, the functions can be controlled through switches specified in the command string to the MACRO-11 Assembler. These switches are:

```
/EN:arg
/DS:arg
```

where: arg is any of the arguments defined for the .ENABL and .DSABL directives.

Use of these switches overrides the enabling or disabling of all occurrences of that argument in the program. They are used in the same manner as /LI, /NL, but in general apply mainly to source files.

```
ØØ463Ø                          LABEL:                         ;LABEL PROCESSOR
                                        .ENABL  LSB
ØØ463Ø  026767  ØØØØØ2' ØØØØ24'         CMP     SYMBOL,R5ØDOT  ;PERIOD?
ØØ4636  ØØ147Ø                          BEQ     4$             ;  YES, ERROR
                                        .IF NDF XEDLSB
ØØ464Ø                                  CALL    LSBSET         ;FLAG START OF NEW LOCAL SYMBOL BLOCK
                                        .ENDC
ØØ4644                                  SSRCH                  ;NO, SEARCH THE SYMBOL TABLE
ØØ465Ø                                  CRFDEF
ØØ4654                          LABELF: SETXPR                 ;SET EXPRESSION REGISTERS
ØØ4662  ØØ5Ø46                          CLR     -(SP)          ;CLEAR GLOBAL FLAG
ØØ4662                                  GETNB                  ;GET NEXT NON BLANK
ØØ4666  ØØØ527  ØØØØ72                  CMP     R5,#CH.COL     ;ANOTHER COLON?
ØØ4672  ØØ1ØØ4                          BNE     1Ø$            ;IF NE NO
ØØ4674  Ø12716  ØØØ1ØØ                  MOV     #GLBFLG,(SP)   ;SET GLOBAL FLAG
ØØ47ØØ                                  GETNB                  ;GET NEXT NON BLANK
ØØ47Ø4                          1Ø$:                           ;REF LABEL
ØØ47Ø4  Ø32713  ØØØØ1Ø                  BIT     #DEFFLG,(R3)   ;ALREADY DEFINED?
ØØ471Ø  ØØ1Ø2Ø                          BNE     1$             ;  YES
ØØ4712  Ø167ØØ  ØØØØ26'                 MOV     CLCFGS,RØ      ;NO, GET CURRENT LOCATION CHARACTERISTIC
ØØ4716  Ø427ØØ  ØØØ337                  BIC     #377-<RELFLG>,RØ        ;CLEAR ALL BUT RELOCATION FLAG
ØØ4722  Ø527ØØ  ØØØØ12                  BIS     #DEFFLG!LBLFLG,RØ       ;FLAG AS LABEL
ØØ4726  Ø516ØØ                          BIS     (SP),RØ        ;INCLUDE PREVIOUS FLAGS FROM ABOVE
ØØ473Ø  Ø32713  ØØØØ2Ø                  BIT     #DFGFLG,(R3)   ;DEFAULTED GLOBAL FROM REFERENCE?
ØØ4734  ØØ14Ø2                          BEQ     2Ø$            ;IF EQ NO
ØØ4736  Ø42713  ØØØ12Ø                  BIC     #DFGFLG!GLBFLG,(R3);CLEAR DEFAULT GLOBAL FLAGS
ØØ4742                          2Ø$:                           ;REF LABEL
ØØ4742  Ø5ØØ13                          BIS     RØ,(R3)        ;SET MODE
ØØ4744  Ø16714  ØØØØ3Ø'                 MOV     CLCLOC,(R4)    ;  AND CURRENT LOCATION
ØØ475Ø  ØØØ416                          BR      3$             ;INSERT

ØØ4752  Ø32713  ØØØØØ2          1$:     BIT     #LBLFLG,(R3)   ;DEFINED, AS LABEL?
ØØ4756  ØØ14Ø6                          BEQ     2$             ;  NO, INVALID
ØØ476Ø  Ø26714  ØØØØ3Ø'                 CMP     CLCLOC,(R4)    ;HAS ANYBODY MOVED?
ØØ4764  ØØ1ØØ3                          BNE     2$             ;  YES
ØØ4766  126712  ØØØØ27'                 CMPB    CLCSEC,(R2)    ;SAME SECTOR?
ØØ4772  ØØ14Ø5                          BEQ     3$             ;  YES, OK
ØØ4774                          2$:     ERROR   P              ;NO, FLAG ERROR
ØØ5ØØ2  Ø52713  ØØØØØ4                  BIS     #MDFFLG,(R3)   ;FLAG AS MULTIPLY DEFINED
ØØ5ØØ6                          3$:     INSERT                 ;INSERT/UPDATE
ØØ5Ø12                                  SETPFØ                 ;BE SURE TO PRINT LOCATION FIELD
ØØ5Ø16  ØØØ4Ø4                          BR      5$

ØØ5Ø2Ø                          4$:     ERROR   Q
ØØ5Ø26  ØØØ4Ø1                          BR      6$             ;NO NEED TO POP STACK
ØØ5Ø3Ø  ØØ5726                  5$:     TST     (SP)+          ;CLEAN STACK
ØØ5Ø32                          6$:     SETNB                  ;SET NONBLANK
ØØ5Ø36  Ø16767  ØØØØØØ' ØØØØ16'         MOV     CHRPNT,LBLEND  ;MARK END OF LABEL
ØØ5Ø44                                  BRJMP   STMNT          ;TRY FOR MORE

                                        .DSABL  LSB
```

Figure 6-4  Example of .ENABL, .DSABL Directives

## 6.3 DATA STORAGE DIRECTIVES

A wide range of data and data types can be generated with the following directives and assembly characters:

```
        .BYTE
        .WORD
        '
        "
        .ASCII
        .ASCIZ
        .RAD50
        ↑B
        ↑D
        ↑O
```

These facilities are explained in the following sections.

### 6.3.1 .BYTE

The .BYTE directive is used to generate successive bytes of data. The directive is of the form:

```
        .BYTE   exp             ;WHICH STORES THE OCTAL
                                ;EQUIVALENT OF THE EXPRESSION
                                ;EXP IN THE NEXT BYTE.

        .BYTE expl,exp2,...     ;WHICH STORES THE OCTAL
                                ;EQUIVALENTS OF THE LIST OF
                                ;EXPRESSIONS IN SUCCESSIVE BYTES.
```

A legal expression must have an absolute value (or contain a reference to an external symbol) and must result in eight bits or less of data. The 16-bit value of the expression must have a high-order byte (which is truncated) that is either all zeros or all ones. Each operand expression is stored in a byte of the object program. Multiple operands are separated by commas and stored in successive bytes. For example:

```
SAM=5
.=410
        .BYTE ↑D48,SAM          ;060 (OCTAL EQUIVALENT OF 48
                                ;DECIMAL) IS STORED IN LOCATION
                                ;410, 005, IS STORED IN
                                ;LOCATION 411.
```

If the high-order byte of the expression equates to a value other than 0 or -1, it is truncated to the low-order eight bits and flagged with a T error code. If the expression is relocatable, an A-type warning flag is given.

At Link time it is likely that relocation will result in an expression of more than eight bits, in which case, LINK prints a truncation error message. For example:

```
        .BYTE   23                  ;STORES OCTAL 23 IN NEXT BYTE.
A:
        .BYTE   A                   ;RELOCATABLE VALUE CAUSES AN "A"
                                    ;ERROR FLAG.

        .GLOBL  X
X=3
        .BYTE   X                   ;STORES 3 IN NEXT BYTE.
```

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

```
.=420
        .BYTE ,,                    ;ZEROS ARE STORED IN BYTES 420, 421,
                                    ;AND 422.
```

## 6.3.2 .WORD

The .WORD directive is used to generate successive words of data. The directive is of the form:

```
        .WORD   EXP                 ;WHICH STORES THE OCTAL
                                    ;EQUIVALENT OF THE EXPRESSION
                                    ;EXP IN THE NEXT WORD.

        .WORD expl,exp2,...         ;WHICH STORES THE OCTAL
                                    ;EQUIVALENTS OF THE LIST OF
                                    ;EXPRESSIONS IN SUCCESSIVE
                                    ;WORDS.
```

A legal expression must result in 16 bits or less of data. Each operand expression is stored in a word of the object program. Multiple operands are separated by commas and stored in successive words. For example:

```
SAL=0
.=500
        .WORD   177535,.+4,SAL      ;STORES 177535, 506 AND 0 IN
                                    ;WORDS 500, 502 AND 504.
```

If an expression equates to a value of more than 16 bits, it is truncated and flagged with a T error code.

If an operand following the .WORD directive is null, it is interpreted as zero. For example:

```
.=500
        .WORD   ,5,                 ;STORES 0, 5, and 0 in LOCATIONS
                                    ;500, 502, and 504.
```

A blank operator field (any operator not recognized as a macro call, op-code, directive or semicolon) is interpreted as an implicit .WORD directive. Use of this convention is discouraged because it may not be the default case in future PDP-11 Assemblers. The first term of the first expression in the operand field must not be an instruction mnemonic or assembler directive unless preceded by a + or - operator. For example:

```
.=440                                   ;THE OP-CODE FOR MOV, WHICH
                                        ;IS 010000, IS STORED ON
LABEL:  +MOV,LABEL                      ;LOCATION 440.  440 IS
                                        ;STORED IN LOCATION 442.
```

Note that the default .WORD directive occurs whenever there is a leading arithmetic or logical operator, or whenever a leading symbol is encountered which is not recognized as a macro call, an instruction mnemonic or assembler directive. Therefore, if an instruction mnemonic, macro call or assembler directive is misspelled, the .WORD directive is assumed and errors will result. Assume that MOV is spelled incorrectly as MOR:

```
        MOR     A,B
```

Two error codes result: Q occurs because an expression operator is missing between MOR and A, and a U occurs if MOR is undefined. The U error occurs only if GBL is disabled and MOR is undefined, else MOR is classed as a global. Two words are then generated; one for MOR A and one for B.

## 6.3.3 ASCII Conversion of One or Two Characters

The ' and " characters are used to generate text characters within the source text. A single apostrophe followed by a character results in a word in which the 7-bit ASCII representation of the character is placed in the low-order byte and zero is placed in the high-order byte. For example:

```
        MOV     #'A,R0
```

results in the following 16 bits being moved into R0:

```
              -------------------
              !     000!     101!
              -------------------
                        octal ASCII value of A
```

```
STMNT:
        GETSYM
        BEQ     4$
        CMPB    @CHRPNT,#':         ;COLON DELIMITS LABEL FIELD.

        BEQ     LABEL
        CMPB    @CHRPNT,#'=         ;EQUAL DELIMITS
        BEQ     ASGMT               ;ASSIGNMENT PARAMETER.
```

A double quote followed by two characters results in a word in which the 7-bit ASCII representations of the two characters are placed in the word. For example:

```
        MOV     #"AB,R0
```

results in the following word being moved into R0:

```
                -------------------
                !    102 !     101!
                -----↑---------↑----
                     !         
                     !       ---octal ASCII of A
                     --octal ASCII of B
```

```
;DEVICE NAME TABLE

DEVNAM: .WORD   "DF             ;RF DISK
        .WORD   "DK             ;RK DISK
        .WORD   "DP             ;RP DISK
DEVNKB: .WORD   "KB             ;TTY KEYBOARD
        .WORD   "DT             ;DECTAPE
        .WORD   "LP             ;LINE PRINTER
        .WORD   "PR             ;PAPER TAPE READER
        .WORD   "PP             ;PAPER TAPE PUNCH
        .WORD   "CR             ;CARD READER
        .WORD   "MT             ;MAGTAPE
        .WORD   0               ;TABLE'S END
```

## 6.3.4 .ASCII

The .ASCII directive translates character strings into their 7-bit ASCII equivalents for use in the source program. The format of the .ASCII directive is as follows:

```
        .ASCII                  /character string/
```

where: character string — is a string of any acceptable printable ASCII characters. The string may not include null (blank) characters, rubout, return, line feed, vertical tab, or form feed. Nonprinting characters can be expressed in digits of the current radix and delimited by angle brackets. (Any legal, defined expression is allowed between angle brackets.)

/ / — these are delimiting characters and may be any printing characters other than ; < and = characters and any character within the string.

As an example:

```
A:      .ASCII /HELLO/          ;STORES ASCII REPRESENTATION OF
                                ;THE LETTERS H.E.L.L.O IN
                                ;CONSECUTIVE BYTES.

        .ASCII /ABC/<15><12>/DEF/
                                ;STORES A,B,C,15,12,D,E,F IN
                                ;CONSECUTIVE BYTES.
        .ASCII /<AB>/           ;STORES <,A,B,> IN CONSECUTIVE
                                ;BYTES.
```

The ; and = characters are not illegal delimiting characters, but are preempted by their significance as a comment indicator and assignment operator, respectively. For other than the first group, semicolons are treated as beginning a comment field. For example:

```
        .ASCII  ;ABC;/DEF/      ;STORES A,B,C,D,E,F
                                ;NOT RECOMMENDED PRACTICE

        .ASCII  /ABC/;DEF;      ;STORES A,B,C. DEF TREATED
                                ;AS A COMMENT

        .ASCII  /ABC/=DEF=      ;SAME AS CASE 1

        .ASCII  =DEF=           ;THE ASSIGNMENT
                                ;.ASCII=DEF
                                ;IS PERFORMED AND A Q ERROR GENERATED
                                ;UPON ENCOUNTERING
                                ;THE SECOND =.
```

## 6.3.5 .ASCIZ

The .ASCIZ directive is equivalent to the .ASCII directive with a zero byte automatically inserted as the final character of the string. For example:

> When a list or text string has been created with a .ASCIZ directive, a search for the null character can determine the end of the list. For example:

```
        .
        .
        .
        MOV     #HELLO,R1
        MOV     #LINBUF,R2
X:      MOVB    (R1)+,(R2)+
        BNE     X
        .
        .
        .
HELLO:  .ASCIZ  <CR><LF>/MACRO-11 V001A/<CR><LF>   ;INTRO MESSAGE
```

## 6.3.6 .RAD50

The .RAD50 directive allows the user the capability to handle symbols in Radix-50 coded form (this form is sometimes referred to as MOD40 and is used in PDP-11 system programs). Radix-50 form allows three characters to be packed into sixteen bits; therefore, any 6-character symbol can be held in two words. The form of the directive is:

```
.RAD50    /string/
```

| | | |
|---|---|---|
| where: | / / | delimiters can be any printing characters other than the =, <, and ; characters. |
| | string | is a list of the characters to be converted (three characters per word) and which may consist of the characters A through Z, 0 through 9, dollar ($), dot (.) and space ( ). If there are fewer than three characters (or if the last set is fewer than three characters) they are considered to be left justified and trailing spaces are assumed. Illegal nonprinting characters are replaced with a ? character and cause an I error flag to be set. Illegal printing characters set the Q error flag. |

The trailing delimiter may be a semicolon, or matching delimiter. For example:

```
.RAD50  /ABC/         ;PACK ABC INTO ONE WORD.
.RAD50  /AB/          ;PACK AB (SPACE) INTO ONE WORD.
.RAD50  /ABCD/        ;PACK ABC INTO FIRST WORD AND
                      ;D SPACE SPACE INTO SECOND WORD.
```

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

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

The character code for 35 is currently undefined.

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

Radix 50 value = ((C*50)+C2)*50+C3

For example:

Radix-50 value of ABC is ((1*50)+2)*50+3 or 3223

See Appendix A for a table of Radix-50 equivalents.

Use of angle brackets is encouraged in the .ASCII, .ASCIZ, and .RAD50 statements whenever leaving the text string to insert special codes. For example:

```
        .ASCII   <101>              ;EQUIVALENT TO .ASCII/A/

        .RAD50   /AB/<35>           ;STORES 3255 IN NEXT WORD

CHR1=1
CHR2=2
CHR3=3
            .
            .
            .
        .RAD50   <CHR1><CHR2><CHR3>
                                    ;EQUIVALENT TO .RAD50/ABC/
```

## 6.4 RADIX CONTROL

### 6.4.1 .RADIX

Numbers used in a MACRO-11 source program are initially considered to be octal numbers. However, the programmer has the option of declaring the following radices:

```
          2, 4, 8, 10
```

This is done via the .RADIX directive, of the form:

```
          .RADIX  n
```

where:  n   is one of the acceptable radices.

The argument to the .RADIX directive is always interpreted in decimal radix. Following any radix directive, that radix is the assumed base for any number specified until the following .RADIX directive.

The default radix at the start of each program, and the argument assumed if none is specified, is 8 (i.e., octal). For example:

```
        .RADIX 10                   ;BEGINS SECTION OF CODE WITH
                                    ;DECIMAL
                                    ;RADIX
            .
            .
            .
        .RADIX                      ;REVERTS TO OCTAL RADIX
```

In general it is recommended that macro definitions not contain or rely on radix settings from the .RADIX directive. The temporary radix control characters should be used within a macro definition. (↑D, ↑O, and ↑B are described in the following section.) A given radix is valid 5dughout a program until changed. Where a possible conflict exists within a macro definition or in possible future uses of that code module, it is suggested that the user specify values using the ᐩemporary radix controls (see below).

```
COUNT           BUFF-2

                BUFF
```

## 6.5.3 .BLKB and .BLKW

Blocks of storage can be reserved using the .BLKB and .BLKW directives. .BLKB is used to reserve byte blocks and .BLKW reserves word blocks. The two directives are of the form:

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

| | | |
|---|---|---|
| where: | exp | is the number of bytes or words to reserve. If no argument is present, 1 is the assumed default value. Any legal expression which is completely defined at assembly time and produces an absolute number is legal. Using these directives without arguments is not recommended. |

For example:

```
1          000000'        .CSECT   IMPURE
2
3 000000          PASS:   .BLKW
4                                               ;NEXT GROUP MUST STAY TOGETHER
5 000002          SYMBOL: .BLKW    2            ;SYMBOL ACCUMULATOR
6 000006          MODE:
7 000006          FLAGS:  .BLKB    1            ;FLAG BITS
8 000007          SECTOR: .BLKB    1            ;SYMBOL/EXPRESSION TYPE
9 000010          VALUE:  .BLKW    1            ;EXPRESSION VALUE
10 00012          RELLVL: .BLKW    1
11                        .BLKW    2            ;END OF GROUPED DATA
12
13 00020          CLCNAM: .BLKW    2            ;CURRENT LOCATION COUNTER SYMBOL
14 00024          CLCFGS: .BLKB    1
15 00025          CLCSEC: .BLKB    1
16 00026          CLCLOC: .BLKW    1
17 00030          CLCMAX: .BLKW    1
```

The .BLKB directive has the same effect as:

```
.=.+exp
```

but is easier to interpret in the context of source code.

## 6.6 NUMERIC CONTROL

Several directives are available to simplify the use of the floating-point hardware on the PDP-11.

A floating-point number is represented by a string of decimal digits. The string (which can be a single digit in length) may optionally contain a decimal point, and may be followed by an optional exponent indicator in the form of the letter E and a signed decimal exponent. The list of number representations below contains seven distinct, valid representations of the same floating-point number:

```
3
3.
3.0
3.0E0
3E0
.3E1
300E-2
```

As can be quickly inferred, the list could be extended indefinitely (e.g., 3000E-3, .03E2, etc.). A leading plus sign is ignored (e.g., +3.0 is considered to be 3.0). A leading minus sign complements the sign bit. No other operators are allowed (e.g., 3.0+N is illegal).

Floating-point number representations are valid only in the contexts described in the remainder of this section.

Floating-point numbers are normally rounded. That is, when a floating-point number exceeds the limits of the field in which it is to be stored, the high-order excess bit is added to the low-order retained bit. For example, if the number is to be stored in a 2-word field, but more than 32 bits are needed for its value, the highest bit carried out of the field is added to the least significant position. The .ENABL FPT directive is used to enable floating-point truncation, and .DSABL FPT is used to return to floating-point rounding (see section 6.2).

### 6.6.1 .FLT2 and .FLT4

Like the .WORD directive, the two floating-point storage directives cause their arguments to be stored in-line with the source program. These two directives are of the form:

```
.FLT2   arg1,arg2,...
.FLT4   arg1,arg2,...
```

where: arg1,arg2,... represent one or more floating point numbers separated by commas.

.FLT2 causes two words of storage to be generated for each argument, while .FLT4 generates four words of storage.

## 6.9 PROGRAM SECTION DIRECTIVES

### 6.9.1 .PSECT Directive

Program sections are defined by the .PSECT directive, which is formatted as:

```
.PSECT [NAME][,RO/RW][,I/D][,GBL/LCL][,ABS/REL][,CON/OVR][,HGH/LOW]
```

The brackets ([]) are for purposes of illustrating optional parameters, and are not included in the parameter specifications. The slash (/) indicates that a choice is to be made between the parameters. The program section attribute parameters are summarized in Table 6-2.

Table 6-2

.PSECT Directive Parameters

| Parameter | Default | Meaning |
|---|---|---|
| NAME | Blank | Program section name, in Radix-50 format, specified as one to six characters. If omitted, a comma must appear in the first parameters position. |
| RO/RW | RW | Program section access mode;<br>RO=Read Only<br>RW=Read/Write |
| I/D | I | Program section type;<br>I=Instruction<br>D=Data |
| GBL/LCL | LCL | The scope of the program section, as interpreted by LINK;<br>GBL=Global<br>LCL=Local |
| ABS/REL | REL | Defines relocation of the program section;<br>ABS=Absolute (no relocation)<br>REL=Relocatable (a relocation bias is required) |
| CON/OVR | OVR | Program section allocation;<br>CON=Concatenated<br>OVR=Overlaid |

| | | |
|---|---|---|
| HGH/LOW | LOW | Program section memory type;<br><br>HGH=High-speed<br>LOW=Core |

***NOTE***
The HGH/LOW attribute is currently ignored by LINK.

The only parameter that is position-dependent is NAME. If it is omitted, a comma must be used in its place. For example,

```
        .PSECT ,RO
```

This example shows a PSECT with a blank name and the Read Only access parameter. Defaults are used for the remaining parameters.

LINK interprets the .PSECT directive's parameters as follows:

| | |
|---|---|
| RO/RW | Defines the type of access to the program section permitted which is; Read Only, or Read/Write. |
| I/D | Allows LINK to differentiate global symbols that are entry points (I) from global symbols that are data values (D). |
| GBL/LCL | Defines the scope of a program section. A global program section's scope crosses segment (overlay) boundaries; a local program section's scope is within a single segment. In single-segment programs, the GBL/LCL parameter is ignored. |
| ABS/REL | When ABS is specified, the program section is absolute. No relocation is necessary (i.e., the program section is assembled starting at absolute virtual 0). When REL is specified, a relocation bias is calculated by LINK, and added to all references in the section. |
| CON/OVR | CON causes LINK to collect all allocation references to the program section from different modules and concatenate them to form the total allocation for the program section. OVR indicates that all allocation references to the program section overlay one another. Thus, the total allocation of the program section is determined by the largest request made by a module that references it. |

Once the attributes of a named .PSECT are declared in a module, the MACRO-11 Assembler assumes that this .PSECT's attributes hold for all subsequent declarations of the named .PSECT in the same module. Thus, the attributes may be declared once, and later .PSECT's with the same name will have the same attributes, when specified within the same module.

The Assembler provides for 255(10) program sections: One absolute section, one blank relocatable section, and 253(10) named relocatable sections are permitted. The .PSECT directive enables the user to:

location by LINK All other program sections (those with the attribute CON) are concatenated.

Note that there is no conflict between internal symbolic names and program section names; that is, it is legal to use the same symbolic name for both purposes. In fact, considering FORTRAN again, this is necessary to accommodate the FORTRAN statement:

```
     COMMON /X/A,B,C,X
```

where the symbol X represents the base of this program section and also the fourth element of this program section.

Program section names should not duplicate .GLOBL names. In FORTRAN language, COMMON block names and SUBROUTINE names should not be the same.

## 6.9.2 .ASECT and .CSECT Directives

DOS/BATCH assembly language programs use the .PSECT directive exclusively, as it affords all the capabilities of the .ASECT and .CSECT directives defined for other PDP-11 assemblers. The Macro Assembler will accept .ASECT and .CSECT but assembles them as if they were .PSECT's with the default attributes listed below. Also, compatibility exists between non-DOS/BATCH MACRO-11 programs and LINK, because LINK recognizes .ASECT and .CSECT directives that appear in such programs. LINK accepts these directives from non-DOS/BATCH programs, and assigns default values as shown in Table 6-3.

Table 6-3

Non-DOS/BATCH Program Section Defaults

| Attribute | Default Value | | |
|---|---|---|---|
| | .ASECT | .CSECT (named) | .CSECT |
| Name | ABS | name | Blank |
| Access | RW | RW | RW |
| Type | I | I | I |
| Scope | GBL | GBL | LCL |
| Relocation | ABS | REL | REL |
| Allocation | OVR | OVR | CON |
| Memory | LOW | LOW | LOW |

The allowable syntactical forms of .ASECT and .CSECT are:

```
          .ASECT
          .CSECT
          .CSECT   symbol
```

Note that

```
          .CSECT  JIM
```

is identical to

```
          .PSECT  JIM,GBL,OVR
```

## 6.10 SYMBOL CONTROL: .GLOBL

The Assembler produces a relocatable object module and a listing file containing the assembly listing and symbol table. LINK joins separately assembled object modules into a single load module. Object modules are relocated as a function of the specified base of the load module. The object modules (where there are more than one) are linked via global symbols, such that a global symbol in one module (either defined by direct assignment or as a label) can be referenced from another module.

A global symbol may be specified in a .GLOBL directive.

In addition, symbols referenced but not defined within a module are assumed to be global references. The .GLOBL directive is provided to reference (and provide linkage to) symbols not otherwise referenced within a module. For example, one might include a .GLOBL directive to cause linkage to a library. When defining a global definition, the .GLOBL A,B,C directive is equivalent to

```
          A==value  (or A::value)
          B==value  (or B::value)
          C==value  (or C::value)
```

The form of the .GLOBL directive is:

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

| | | |
|---|---|---|
| where: | sym1,sym2,... | are legal symbolic names, separated by commas or spaces where more than one symbol is specified. |

Symbols appearing in a .GLOBL directive are either defined within the current program or are external symbols, in which case they are defined in another program which is to be linked with the current program by LINK prior to execution.

A .GLOBL directive line may contain a label in the label field and comments in the comment field.

At the end of assembly pass 1, MACRO-11 has determined whether a given global symbol is defined within the program or is expected to be an external symbol. All internal symbols to a given program, then, must be defined by the end of pass 1 or they will be assumed to be global references (see .ENABL, .DSABL of globals in section 6.1.6).

For example:

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

assembles if both SYM1 and SYM2 are defined.

### 6.11.1 Subconditionals

Subconditionals may be placed within conditional blocks to indicate:

1. Assembly of an alternate body of code when the condition of the block indicates that the code within the block is not to be assembled.
2. Assembly of a non-contiguous body of code within the conditional block depending upon the result of the conditional test to enter the block.
3. Unconditional assembly of a body of code within a conditional block.

There are three subconditional directives, as follows:

| Subconditional Directives | Function |
|---|---|
| .IFF | The code following this statement up to the next subconditional or end of the conditional block is included in the program providing the value of the condition tested upon entering the conditional block was false. |
| .IFT | The code following this statement up to the next subconditional or end of the conditional block is included in the program providing the value of the condition tested upon entering the conditional block was true. |
| .IFTF | The code following this statement up to the next subconditional or the end of the conditional block is included in the program regardless of the value of the condition tested upon entering the conditional block. |

The implied argument of the subconditionals is the value of the condition upon entering the conditional block. Subconditionals are used within outer level conditional blocks. Subconditionals are ignored within nested, unsatisfied conditional blocks.

For example:

```
        .IF DF   SYM              ;ASSEMBLE BLOCK IF SYM IS DEFINED
        .IFF
          .                       ;ASSEMBLE THE FOLLOWING CODE ONLY IF
          .                       ;SYM IS UNDEFINED.
          .
        .IFT                      ;ASSEMBLE THE FOLLOWING CODE ONLY IF
          .                       ;SYM IS DEFINED.
          .
          .
        .IFTF                     ;ASSEMBLE THE FOLLOWING CODE
          .                       ;UNCONDITIONALLY.
          .
          .
        .ENDC


        .IF DF   X                ;ASSEMBLY TESTS FALSE
        .IF DF   Y                ;TESTS FALSE
        .IFF                      ;NESTED CONDITIONAL
          .                       ;IGNORED
          .
          .
        .IFT                      ;NOT SEEN
          .
          .
          .
        .ENDC
        .ENDC
```

However,

```
        .IF DF   X                ;TESTS TRUE
        .IF DF   Y                ;TESTS FALSE
        .IFF                      ;IS ASSEMBLED
          .
          .
          .
        .IFT                      ;NOT ASSEMBLED
          .
          .
          .
        .ENDC
        .ENDC
```

## 6.11.2 Immediate Conditionals

An immediate conditional directive is a means of writing a 1-line conditional block. In this form, no .ENDC statement is required and the condition is completely expressed on the line containing the conditional directive. Immediate conditions are of the form:

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

```
.ENDM name
```

where:

name	is an optional argument, being the name of the macro terminated by the statement.

For example:

```
.ENDM     (terminates the current macro definition)

.ENDM ABS (terminates the definition of the macro ABS)
```

If specified, the symbolic name in the .ENDM statement must correspond to that in the matching .MACRO statement. Otherwise the statement is flagged and processing continues. Specification of the macro name in the .ENDM statement permits the Assembler to detect missing .ENDM statements or improperly nested macro definitions.

The .ENDM statement may contain a comment field, but must not contain a label.

An example of a macro definition is shown below:

```
.MACRO  TYPMSG MESSGE    ;TYPE A MESSAGE
JSR     R5,TYPMSG
.WORD   MESSGE
.ENDM
```

## 7.1.3 .MEXIT

In order to implement alternate exit points from a macro (particularly nested macros), the .MEXIT directive is provided. .MEXIT terminates the current macro as though an .ENDM directive were encountered. Use of .MEXIT bypasses the complications of conditional nesting and alternate paths. For example:

```
.MACRO  ALTR  N,A,B
   .
   .
   .
.IF EQ,N                  ;START CONDITIONAL BLOCK
   .
   .
   .
.MEXIT                   ;EXIT FROM MACRO DURING CONDITIONAL
                         ;BLOCK
.ENDC                    ;END CONDITIONAL BLOCK
   .
   .
   .
.ENDM                    ;NORMAL END OF MACRO
```

In an assembly where N=0, the .MEXIT directive terminates the macro expansion.

Where macros are nested, a .MEXIT causes an exit to the next higher level. A .MEXIT encountered outside a macro definition is flagged as an error.

### 7.1.4 MACRO Definition Formatting

A form feed character used as the only character on a line causes a page eject. Used within a macro definition, a form feed character causes a page eject. A page eject is not performed when the macro is invoked.

Used within a macro definition, the .PAGE directive is ignored, but a page eject is performed at invocation of that macro.

## 7.2 MACRO CALLS

A macro must be defined prior to its first reference. Macro calls are of the general form:

```
            label:    name, real arguments
```

| | | |
|---|---|---|
| where: | label | represents an optional statement label. |
| | name | represents the name of the macro specified in the .MACRO directive preceding the macro definition. |
| | | represents any legal separator (comma, space, or tab). No separator is necessary where there are no real arguments. |
| | real arguments | are those symbols, expressions, and values which replace the dummy arguments in the .MACRO statement. Where more than one argument is used, they are separated by any legal separator. |

Where a macro name is the same as a user label, the appearance of the symbol in the operation field designates a macro call, and the occurrence of the symbol in the operand field designates a label reference. For example:

```
ABS:    MOV     @R0,R1          ;ABS IS USED AS LABEL
         .
         .
         .
        BR      ABS             ;ABS IS CONSIDERED A LABEL
         .
         .
         .
        ABS     #4,ENT,LAR      ;CALL MACRO ABS WITH 3 ARGUMENTS
```

Arguments to the macro call are treated as character strings whose usage is determined by the macro definition.

## 7.3 ARGUMENTS TO MACRO CALLS AND DEFINITIONS

Arguments within a macro definition or macro call are separated from other arguments by any of the separating characters described in Section 3.1.1.

For example:

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

Arguments which contain separating characters are enclosed in paired angle brackets. An up-arrow construction is provided to allow angle brackets to be passed as arguments. Bracketed arguments are seldom used in a macro definition, but are more likely in a macro call. For example:

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

This call would cause the entire statement:

```
        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 until their use in the macro expansion.

The up-arrow construction could have been used in the above macro call as follows:

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

which is equivalent to:

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

Since spaces are ignored preceding an argument, they can be used to increase legibility of bracketed contructions.

The form:

```
        REN #44,WEV↑/MOV X,Y/
```

however, contains only two arguments: #44 and WEV↑/MOV X,Y/ (see section 3.1.1) because ↑ is a unary operator.

### 7.3.1 Macro Nesting

Macro nesting (nested macro calls), where the expansion of one macro includes a call to another macro, causes one set of angle brackets to be removed from an argument with each nesting level. The depth of nesting allowed is dependent upon the amount of core space used by the program being assembled. To pass an argument containing legal argument delimiters to nested macros, the argument should be enclosed in one set of angle brackets for each level of nesting, as shown below:

```
        .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:

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

causes the following expansion:

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

where macro definitions are nested (that is, a macro definition is entirely contained within the definition of another macro) the inner definition is not defined as a callable macro until the outer macro has been called and expanded. For example:

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

The LV2 macro cannot be called by name until after the first call to the LV1 macro. Likewise, any macro defined within the LV2 macro definition cannot be referenced directly until LV2 has been called.

## 7.3.2 Special Characters

Arguments may include special characters without enclosing the argument in a bracket construction if that argument does not contain spaces, tabs, semicolons, or commas. For example:

```
        .MACRO  PUSH ARG
        MOV     ARG,-(SP)
        .ENDM
```

```
         .
         .
         .
        PUSH    X+3(%2)
```

generates the following code:

```
        MOV     X+3(%2),-(SP)
```

## 7.3.3 Numeric Arguments Passed as Symbols

When passing macro arguments, a useful capability is to pass a symbol which can be treated by the macro as a numeric string. An argument preceded by the unary operator backslash (\) is treated as a number in the current radix. The ASCII characters representing the number are inserted in the macro expansion; their function is defined in context. For example:

```
        B=0
        .MACRO  INC A,B
        CNT     A, \B
        N=N+1
        .ENDM
        .MACRO  CON A,B
A'B     .WORD
        .ENDM
          .
          .
          .
        INC     X,C
```

The macro call would expand to:

```
X0:     .WORD   4
```

A subsequent identical call to the same macro would generate:

```
X1:     .WORD   4
```

and so on for later calls. The two macros are necessary because the dummy value of B cannot be updated in the CNT macro. In the CNT macro, the number passed is treated as a string argument. (Where the value of the real argument is 0, a single 0 character is passed to the macro expansion.)

The number being passed can also be used to make source listings somewhat clearer. For example, versions of programs created through conditional assembly of a single source can identify themselves as follows:

```
        .MACRO   IDT  SYM          ;ASSUME THAT THE SYMBOL ID TAKES
        .IDENT   /SYM/             ;ON A UNIQUE 2-DIGIT VALUE FOR
        .ENDM                      ;EACH POSSIBLE CONDITIONAL ASSEMBLY
        .MACRO   OUT ARG           ;OF THE PROGRAM
        IDT      005A'ARG              .
        .ENDM                          .
            .                          .
            .
            .                      ;WHERE 005A IS THE UPDATE
        OUT      \ID               ;VERSION OF THE PROGRAM
                                   ;AND ARG INDICATES THE
                                   ;CONDITIONAL ASSEMBLY VERSION.
```

The above macro call expands to

```
        .IDENT   /005AXX/
```

where XX is the conditional value of ID.

Two macros are necessary since the text delimiting characters in the .IDENT statement would inhibit the concatenation of a dummy argument.

## 7.3.4 Number of Arguments

If more arguments appear in the macro call than in the macro definition, the excess arguments are ignored. If fewer arguments appear in the macro call than in the definition, missing arguments are assumed to be null (consist of no characters). The conditional directives .IF B and .IF NB can be used within the macro to detect unnecessary arguments.

A macro can be defined with no arguments.

## 7.3.5 Automatically Created Symbols

MACRO-11 can create symbols of the form n$ where n is a decimal integer number such that 64<n<127. Created symbols are always local symbols between 64$ and 127$. (For a description of local symbols, see Section 3.5.) Such local symbols are created by the Assembler in numerical order, i.e.:

```
          64$
          65$
           .
           .
           .
          126$
          127$
```

Created symbols are particularly useful where a label is required in the expanded macro. Such a label must otherwise be explicitly stated as an argument with each macro call or the same label is generated with each expansion (resulting in a multiply-defined label). Unless a label is referenced from outside the macro, there is no reason for the programmer to be concerned with that label.

The symbol is separated from the character string argument by any legal separator.

<character string> is a string of printing characters which should only be enclosed in angle brackets if it contains a legal separator. A semicolon also terminates the character string.

The .NCHR directive can occur anywhere in a MACRO-11 program.

The .NTYPE directive enables the macro being expanded to determine the addressing mode of any argument, and is of the form:

label: .NTYPE symbol, arg

where:

| | |
|---|---|
| label | is an optional statement label |
| symbol | is any legal symbol, the value of which is equated to the 6-bit addressing mode of the argument. The symbol is separated from the argument by a legal separator. This symbol can be used by itself or in expressions. |
| arg | is any legal macro argument (dummy argument) as defined in section 7.3. |

The .NTYPE directive can occur only within a macro definition. An example of .NTYPE usage in a macro definition is shown below:

```
.MACRO  SAVE ARG
.NTYPE  SYM,ARG
.IF     EQ,SYM&70
MOV     ARG,TEMP            ;REGISTER MODE
.IFF
MOV     #ARG,TEMP           ;NON-REGISTER MODE
.ENDC
.ENDM
```

## 7.5 .ERROR and .PRINT

The .ERROR directive is used to output messages to the command output device during assembly pass 2. A common use is to provide diagnostic announcements of a rejected or erroneous macro call. The form of the .ERROR directive is as follows:

label: .ERROR expr;text

where

| | |
|---|---|
| label | is an optional statement label |
| expr | is an optional legal expression whose value is output to the command device when the .ERROR directive is encountered. Where expr is not specified, the text only is output to the command device. |
| ; | denotes the beginning of the text string to be output. |

text        is the string to be output to the command device.

Upon encountering an .ERROR directive anywhere in a MACRO-11 program, the Assembler outputs a single line containing:

1. The sequence number of the .ERROR directive line;

2. The current value of the location counter;

3. The value of the expression if one is specified; and,

4. The text string specified.

For example:

```
        .ERROR    A;UNACCEPTABLE MACRO ARGUMENT
```

causes a line similar to the following to be output:

```
        Seq# l.c. A value       Text

        512  5642 000076     ;UNACCEPTABLE MACRO ARGUMENT
```

This message is being used to indicate an inability of the subject macro to cope with the argument A which is detected as being indexed deferred addressing mode (mode 7) with the stack pointer (%6) used as the index register.

The line is flagged on the assembly listing with a P error code.

The .PRINT directive is identical to .ERROR except that it is not flagged with a P error code.

## 7.6 INDEFINITE REPEAT BLOCK: .IRP AND .IRPC

An indefinite repeat block is a structure very similar to a macro definition. An indefinite repeat is essentially a macro definition which has only one dummy argument and is expanded once for every real argument supplied. An indefinite repeat block is coded in-line with its expansion rather than being referenced by name as a macro is referenced. An indefinite repeat block is of the form:

```
label:  .IRP arg,<real arguments>
          .
          .
          .
        (range of the indefinite repeat)
   .
          .
          .
        .ENDM
```

where: label is an optional statement label. A label may not appear on any .IRP statement within another macro definition, repeat range or indefinite repeat range, or on any .ENDM statement.

arg is a dummy argument which is successively replaced with the real arguments in the .IRP statement.

<real argument> is a list of arguments to be used in the expansion of the indefinite repeat range and enclosed in angle-brackets. Each real argument is a string of zero or more characters or a list of real arguments (enclosed in angle brackets). The real arguments are separated by commas.

range is the block of code to be repeated once for each real argument in the list. The range may contain macro definitions, repeat ranges, or other indefinite repeat ranges. Note that only created symbols should be used as labels within an indefinite repeat range.

An indefinite repeat block can occur either within or outside macro definitions, repeat ranges, or indefinite repeat ranges. The rules for creating an indefinite repeat block are the same as for the creation of a macro definition (for example, the .MEXIT statement is allowed in an indefinite repeat block). Indefinite repeat arguments follow the same rules that apply to macro arguments.

```
1                                 .TITLE   IRPTST
2                                 .LIST    MD,MC,ME
3                                 .MCALL   .PARAM
4 000000                          .PARAM
          000000 R0=%^O0
          000001 R1=%^O1
          000002 R2=%^O2
          000003 R3=%^O3
          000004 R4=%^O4
          000005 R5=%^O5
          000006 R6=%^O6
          000007 R7=%^O7
          000006 SP=%^O6
          000007 PC=%^O7
          177776 PSW=^O177776
          177570 SWR=^O177570
5 000000  012700                  MOV      #TABLE,R0
          000056'
6
7                                 .IRP     X,<A,B,C,D,E,F>
8
9                                 MOV      X,(R0)+
10
11                                .ENDM

    00004 016720                  MOV      A,(R0)+
          000032


    00010 016720                  MOV      B,(R0)+
          000030


    00014 016720                  MOV      C,(R0)+
          000026


    00020 016720                  MOV      D,(R0)+
          000024


    00024 016720                  MOV      E,(R0)+
          000022


    00030 016720                  MOV      F,(R0)+
          000020
```

Figure 7-1

.IRP and .IRPC Example

A second type of indefinite repeat block is available which handles character substitution rather than argument substitution. The .IRPC directive is used as follows:

```
label:  .IRPC arg,string
          .
          .
          .
        (range of indefinite repeat)
          .
          .
          .
        .ENDM
```

On each iteration of the indefinite repeat range, the dummy argument (arg) assumes the value of each successive character in the string.

## 7.7 REPEAT BLOCK: .REPT

Occasionally it is useful to duplicate a block of code a number of times in line with other source code. This is performed by creating a repeat block of the form:

```
.label: .REPT expr
          .
          .
          .
        (range of repeat block)
          .
          .
          .
        .ENDM                   ;OR .ENDR
```

| where: | | |
|---|---|---|
| | label | is an optional statement label. The .ENDR or .ENDM directive may not have a label. A .REPT statement occurring within another repeat block, indefinite repeat block, or macro definition may not have a label associated with it. |
| | expr | is any legal expression controlling the number of times the block of code is assembled. Where expr =0, the range of the repeat block is not assembled. |
| | range | is the block of code to be repeated expr number of times. The range may contain macro definitions, indefinite repeat ranges, or other repeat ranges. Note that no statements within a repeat range can have a label. |

The last statement in a repeat block can be an .ENDM or .ENDR statement. The .ENDR statement is provided for compatibility with previous assemblers.

The .MEXIT statement is also legal within the range of a repeat block.

## 7.8 MACRO LIBRARIES: .MCALL

All macro definitions must occur prior to their referencing within the user program. MACRO-11 provides a selection mechanism for the programmer to indicate in advance those system macro definitions required by his program.

The .MCALL directive is used to specify the names of all system macro definitions not defined in the current program but required by the program. The .MCALL directive must appear before the first occurrence of a macro call for an externally defined macro. The .MCALL directive is of the form:

```
        .MCALL arg1,arg2,...
```

| | | |
|---|---|---|
| where | arg1,arg2,... | are the names of the macro definitions required in the current program. |

When this directive is encountered, MACRO-11 searches the system library SYSMAC.SML to find the requested definition(s).

# CHAPTER 8

# OPERATING PROCEDURES

The MACRO-11 Asembler assembles one or more ASCII source files containing MACRO-11 statements into a single relocatable binary object file. The output of the Assembler consists of a binary object file and an assembly listing followed by the symbol table listing. A CREF (cross reference) listing can be specified as part of the assembly output by means of a switch option.

## 8.1 LOADING MACRO-11

MACRO-11 is loaded with the Disk Monitor RUN command as follows:

```
        $RUN MACRO
```

(Characters printed by the system are underlined to differentiate them from characters typed by the user.) The Assembler responds by identifying itself and its version number, followed by a # character to indicate readiness to accept a command input string:

```
        MACRO Vxxx

        #
```

## 8.2 COMMAND INPUT STRING

In response to the # printed by the Assembler, the user types the output file specification(s), followed by a left angle bracket, followed by the input file specification(s):

```
        #object,listing<source1,source2,...,sourceN
```

where:

| | |
|---|---|
| object | is the binary object file |
| listing | is the assembly listing file containing the assembly listing and symbol table and, optionally, a separate CRF listing file can be appended to the assembly listing or output as a separate file. |
| source1,source2, ...,sourceN | are the ASCII source files containing the MACRO-11 source program(s). No limit is set on the number of source input files, except as the Assembler is limited by the size of the user-defined and macro symbol tables. |

If an error is made in typing the command string, typing the RUBOUT key erases the immediately preceding character. Repeated typing of the RUBOUT key erases one character for each RUBOUT up to the beginning of the line. Typing CTRL/U erases the entire line.

A null specification in any of the file fields signifies that the associated input or output file is not desired. Each file specification contains the following information (and follows the standard DOS conventions for file specifications):

```
dev:filnam.ext[uic]/option:arg
```

One or more switch options can be specified with each file specification to provide the Assembler with information about that file. The switch options are described in Section 8.3.

A syntactical error detected in the command string causes the Assembler to output the command string up to and including the point where the error was detected, followed by a ? character. The Assembler then reprints the # character and waits for a new command string to be entered. The following command string errors are detected:

| Error | Error Message |
|---|---|
| Illegal switch<br>Too many switches<br>Illegal switch value<br>Too many switch values | ILLEGAL SWITCH |
| Too many output file specifications | TOO MANY OUTPUT FILES |
| No input file specification | INPUT FILE MISSING |

The default value for each file specification is noted below:

| | dev | filnam | ext | uic |
|---|---|---|---|---|
| object | system device | last source file name | .OBJ | current |
| listing | device used for object output | last source file name | .LST | current |
| CREF intermediate | system device | last source file name | .CRF | current |
| source1 | system device | - | .MAC<br>.PAL<br>.null | current |
| source2<br>.<br>.<br>sourceN | device used for source1 (last source file specified) | - | .MAC<br>.PAL<br>.null | current |
| system macro<br><br>file | system device | SYSMAC | .SML | current<br>[1,1] |

## 8.3 SWITCH OPTIONS

There are four types of switch options: listing options, functions, CREF specifications, and pass assembly controls. The listing options are described in detail in Section 6.1.1. The function options are described in detail in Section 6.2. Rather than repeat this information here, the reader is advised to turn to these sections or the summary contained in Appendix B. The switch options are specified in the form:

| Specification | Function |
|---|---|
| /LI<br>/LI:arg<br>/NL:<br>/NL:arg | Listing Control |
| /EN:arg<br>/DS:arg | Function Control |
| /CRF<br>/CRF:arg | Produce cross reference table |
| /PA:1 | Assemble file during Pass 1 only |
| /PA:2 | Assemble file during Pass 2 only |

Switch options specified on the output side apply to both the object and listing files. Switch options specified on the input side apply to the particular file which the switch follows and all subsequent files.

## 8.4 CREF, CROSS-REFERENCE TABLE GENERATION

A cross-reference listing of all or a subset of all symbols used in the source program can be obtained by a call to the CREF routine. CREF can be used in two ways:

a. CREF can be called automatically following an assembly. In order to do this, the /CRF switch is specified following the assembly listing file specification. For example:

```
#,LP:/CRF<FILE1,FILE2
```

This command string sends the assembly listing (FILE2.LST) to the line printer. An intermediate CREF file is created and temporarily stored on the system device (FILE2.CRF) under the current UIC. The CREF routine takes this intermediate file, generates a CREF listing and routes that listing to the line printer. (The CREF listing is appended to the file FILE2.LST.) The CREF intermediate file is then deleted; there is no way to preserve this file when CREF is being called automatically.

b. If no CREF listing is desired immediately, the intermediate CREF file can be saved on the system device; the CREF listing can be generated at a later date. In order to preserve the intermediate CREF file, the MACRO command string is given as follows:

```
#,LP:/CRF:NG<FILE1,FILE2
```

This command string sends the assembly listing (FILE2.LST) to the line printer. The CREF intermediate file (FILE2.CRF) is sent to the system device under the current UIC. (The :NG argument is a mnemonic for "No Go" to CREF; i.e., no automatic transfer to the CREF routine following the output of the assembly listing.)

In order to generate the CREF listing, the CREF routine is run and given a command string indicating the input file specification(s) and a single output file specification. For example:

```
$RU CREF
CREF V001A
#LP:<FILE2.CRF
```

In this case the intermediate file created automatically in the example above is processed to obtain a CREF listing which is then sent to the line printer. The CREF intermediate file is then automatically deleted. If it is desired to preserve the intermediate file, the command string should be given as:

```
#LP:<FILE2.CRF/SA
```

Unless the /SA switch is specified, the default case is always to delete the CREF intermediate file.

The CREF listing is organized into one to five sections, each listing a different type of symbol. The sections are as follows:

| Section Type | Argument |
|---|---|
| user-defined symbols | :S |
| macro symbolic names | :M |
| permanent symbols (instructions, directives) | :P |
| .CSECT symbolic names | :C |
| error codes | :E |

Where no arguments are specified following the /CRF switch, all of the above sections except the permanent symbols are cross referenced. However, then any one argument is specified (other than :NG), no other default sections are assumed or provided. For example, in order to obtain a CREF listing for all five section types, the following switch option specification is used:

```
/CRF:S:M:P:C:E
```

The order in which the agruments are specified does not affect the order of their output, which is as listed above.

Figure 8-1 contains a segment of source code and Figure 8-2 contains a segment of a CREF listing with some references to the code in Figure 8-1.

In the CREF listing, each cross-referenced symbol is printed in the left-hand column, followed by a list of the page-line numbers of the locations in which that symbol appears. A # character following a page-line number indicates the point at which the associated symbol is defined. An @ character disignates a page-line number at which the contents of that symbol are altered.

```
MACRO   Vxxx    17-JUL-73        MACRO Vxxx    17-JUL-73 19:09 PAGE72
OBJECT CODE HANDLERS

                      .SBTTL  OBJECT CODE HANDLERS

012026         ENDP:                                   ;END OF PASS HANDLER
012026                CALL    SETMAX
012026 004767         JSR     PC,SETMAX
       174240
012032 005767         TST     PASS                     ;PASS ONE?
       000000'
012036 001142         BNE     ENDP2                    ;BRANCH IF PASS 2
012040                ENTOVR  4
012040 005767         TST     OBJLNK                   ;PASS ONE, ANY OBJECT?
       001416'
012044 001517         BEQ     30$                      ;  NO
 12046 012767         MOV     #BLKT01,BLKTYP           ;SET BLOCK TYP1 1
       000001
       000542'
 12054                CALL    OBJINI                   ;INIT THE POINTERS
 12054 004767         JSR     PC,OBJINI
       001542
 12060 012701         MOV     #PRGTTL,R1               ;SET "FROM" INDEX
       000050'
 12064 016702         MOV     RLDPNT,R2                ;  AND "TO" INDEX
       000540'
 12070                CALL    GSDDMP                   ;OUTPUT GSD BLOCK
 12070 004767         JSR     PC,GSDDMP
       000660
 12074 005046         CLR     -(SP)                    ;INIT FOR SECTOR SCAN
 12076 012667 10$:    MOV     (SP)+,ROLUPD             ;SET SCAN MARKER
       000006'
 12102                NEXT    SECROL                   ;GET THE NEXT SECTOR
 12102 012700         MOV     #SECROL,R0
       000010
 12106 004767         JSR     PC,NEXT
       005400
 12112 001450         BEQ     20$                      ;BRANCH IF THROUGH
 12114 016746         MOV     ROLUPD,-(SP)             ;SAVE MARKER
       000006'
 12120 012701         MOV     #MODE,R1
       000006'
 12124 011105         MOV     (R1),R5                  ;SAVE SECTOR
 12126 042705         BIC     #377,R5                  ;ISOLATE IT
       000377
 12132 000305         SWAB    R5                       ;  AND PLACE IN RIGHT
 12134 042711         BIC     #-1-<RELFLG>,(R1) ;CLEAR ALL BUT REL BIT
       177737
 12140 052721         BIS     #<GSDT01>+DEFFLG.(R1)+ ;SET TO TYPE 1, DEFINED
       000410
 12144 010521         MOV     R5,(R1)+                 ;ASSURE ABS
 12146 001401         BEQ     11$                      ;  OOPS!
 12150 011141         MOV     (R1),-(R1)               ;  REL, SET MAX
 12152 005067 11$:    CLR     ROLUPD                   ;SET FOR INNER SCAN
       000006'
 12156 012701 12$:    MOV     #SYMBOL,R1
       000002'
 12162                CALL    GSDDMP                   ;OUTPUT THIS BLOCK
 12162 004767         JSR     PC,GSDDMP
```

```
             000566
   32 12166         13$:      NEXT     SYMBOL           ;FETCH THE NEXT SYMBOL
      12166 012700            MOV      #SYMBOL.R0
            000000
      12172 004767            JSR      PC,NEXT
            005314
   33 12176 001737            BEQ      10$              ;  FINISHED WITH THIS GUY
   34 12200 032767            BIT      #GLBFLG,MODE     ;GLOBAL?
            000100
            000006'
   35 12206 001767            BEQ      13$              ;  NO
   36 12210 126705            CMPB     SECTOR,R5        ;YES, PROPER SECTOR?
            000007'
   37 12214 001364            BNE      13$              ;  NO
   38 12216 042767            BIC      #-1-<DEFFLG!RELFLG!GLBFLG>,MODE ;CLEAR MOST
            177627
            000006'
   39 12224 052767            BIS      #GSDT04,MODE     ;SET TYPE 4
            002000
            000006'
   40 12232 000751            BR       12$              ;OUTPUT IT
```

Figure 8-1

Assembly Listing

```
ENDMAC    27-40   109-33#
ENDP      23-23    72- 3#
ENDP1M    73-16    72-22#
ENDP2     72- 6    74- 1#
   .
   .
   .
MDFFLG    12- 7#   35-28     92- 8     92-24
MEXIT    116- 1#  116-41#
MODE      14- 6#   22-29@    34-12     35-17@    36-12     37- 4     40-43
          45- 6@   48-16@    58-38@    64-23     70-10     72-20     72-34
          72-38@   72-39@    74-34     75-37     86- 8     91-20@   106-27
         116-34@
MOVBYT    18- 5    18- 9     28-44     74-41     83-11     83-20    108-19#
MPDP     109-42   121-17#
MPUSH    109-26   110-48    121- 1#
MSBARG    27- 9   121-18    121-40#
MSBBLK   121- 4   121-28    121-36#
MSBCNT    27-15   109-33    116- 6    121-41#
MSBEND   121- 9   121-28    121-43#
MSBMRP    25-19    27-25@   110-49@   121-42#
```

Figure 8-2

Excerpts from CREF Listing to Accompany Figure 8-1.
Note particularly the CREF references for ENDP,
ENDP2, and MODE.

## 8.5 ERROR MESSAGES

The MACRO-11 Assembler outputs the following messages when one of the related errors is detected.

```
COMMAND I/O ERROR
ILLEGAL SWITCH
INPUT FILE MISSING
INSUFFICIENT MEMORY TO COMPLETE ASSEMBLY
I/O ERROR ON OUTPUT FILE
OPEN FAILURE ON INPUT FILE
OPEN FAILURE ON OUTPUT FILE
OUTPUT DEVICE FULL
TOO MANY OUTPUT FILES
```

The error messages are self-explanatory.

# APPENDIX A

# MACRO-11 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, duplicated by 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. duplicated by 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/B (DC0). |
| 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 (TAPE, 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 |

## 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, thus,

```
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 is (arithmetic is performed in octal):

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

| Single Char. or First Char. | | Second Character | | Third Character | |
|---|---|---|---|---|---|
| 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 | 0 | 001130 | 0 | 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 | 000024 |
| 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

# MACRO-11 ASSEMBLY LANGUAGE AND ASSEMBLER

## B.1 SPECIAL CHARACTERS

| Character | Function |
|---|---|
| vertical tab | Source line terminator |
| : | Label terminator |
| = | Direct assignment indicator |
| % | Register term indicator |
| tab | Item terminator<br>Field terminator |
| space | Item terminator<br>Field terminator |
| # | Immediate expression indicator |
| @ | Deferred addressing indicator |
| ( | Initial register indicator |
| ) | Terminal register indicator |
| , (comma) | Operand field separator |
| ; | Comment field indicator |
| + | Arithmetic addition operator or auto increment indicator |
| - | Arithmetic subtraction operator or auto decrement indicator |
| * | Arithmetic multiplication operator |
| / | Arithmetic division operator |
| & | Logical AND operator |
| ! | Logical OR operator |
| " | Double ASCII character indicator |
| ' (apostrophe) | Single ASCII character indicator |
| . | Assembly location counter |
| < | Initial argument indicator |
| > | Terminal argument indicator |
| ↑ | Universal unary operator<br>Argument indicator |
| \ | MACRO numeric argument indicator |

## B.2 ADDRESS MODE SYNTAX

n is an integer between 0 and 7 representing a register. R is a register expression, E is an expression, ER is either a register expression or an expression in the range 0 to 7.

| Format | Address Mode Name | Address Mode Number | Meaning |
|---|---|---|---|
| R | Register | 0n | Register R contains the operand. R is a register expression. |
| @R or (ER) | Deferred Register | 1n | Register R contains the operand address. |
| (ER)+ | Autoincrement | 2n | The contents of the register specified by ER are incremented after being used as the address of the operand. |
| @(ER)+ | Deferred Auto-increment | 3n | ER contains the pointer to the address of the operand. ER is incremented after use. |
| -(ER) | Autodecrement | 4n | The contents of register ER are decremented before being used as the address of the operand. |
| @-(ER) | Deferred Auto-decrement | 5n | The contents of register ER are decremented before being used as the pointer to the address of the operand. |
| E(ER) | Index | 6n | E plus the contents of the register specified, ER, is the address of the operand. |
| #E | Immediate | 27 | E is the operand. |
| @#E | Absolute | 37 | E is the address of the operand. |
| E | Relative | 67 | E is the address of the operand. |
| @E | Deferred Relative | 77 | E is the pointer to the address of the operand. |

## B.3 ASSEMBLER DIRECTIVES

| Form | Described in Manual Section | Operation |
|---|---|---|
| ' | 6.3.3 | A single quote character (apostrophe) followed by one ASCII character generates a word containing the 7-bit ASCII representation of the character in the low-order byte and zero in the high-order byte. |
| " | 6.3.3 | A double quote character followed by two ASCII characters generates a word containing the 7-bit ASCII representation of the two characters. |
| ↑Bn | 6.4.2 | Temporary radix control; causes the number n to be treated as a binary number. |
| ↑Cn | 6.6.2 | Creates a word containing the one's complement of n. |
| ↑Dn | 6.4.2 | Temporary radix control; causes the number n to be treated as a decimal number. |
| ↑Fn | 6.6.2 | Creates a one-word floating point quantity to represent n. |
| ↑On | 6.4.2 | Temporary radix control; causes the number n to be treated as an octal number. |
| .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 | 6.3.5 | 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 following the specified string. |
| .ASECT | 6.9 | Begin or resume absolute section. |
| .BLKB exp | 6.5.3 | Reserves a block of storage space exp bytes long. |
| .BLKW exp | 6.5.3 | Reserves a block of storage space exp words long. |

| | | |
|---|---|---|
| .BYTE exp1,exp2,.. | 6.3.1 | Generates successive bytes of data containing the octal equivalent of the expression(s) specified. |
| .CSECT symbol | 6.9 | Begin or resume named or unnamed relocatable section. |
| .DSABL arg | 6.2 | Disables the assembler function specified by the argument. |
| .ENABL arg | 6.2 | Provides the assembler function specified by the rgument. |
| .END<br>.END exp | 6.7.1 | Indicates the physical end of source program. An optional argument specifies the transfer address. |
| .ENDC | 6.11 | Indicates the end of a condition block. |
| .ENDM<br>.ENDM symbol | 7.1.2 | Indicates the end of the current repeat block, indefinite repeat block, or macro. The optional symbol, if used, must be identical to the macro name. |
| .EOT | 6.7.2 | Ignored. Indicates End-of-Tape which is detected automatically by the hardware. |
| .ERROR exp,string | 7.5 | Causes a text string to be output to the command device containing the optional expression specified and the indicated text string. |
| .EVEN | 6.5.1 | Ensures that the assembly location counter contains an even address by adding 1 if it is odd. |
| .FLT2 arg1,arg2,.. | 6.6.1 | Generates successive two-word floating-point equivalents for the floating-point numbers specified as arguments. |
| .FLT4 arg1,arg2,.. | 6.6.1 | Generates successive four-word floating-point equivalents for the floating-point numbers specified as arguments. |
| .GLOBL sym1,sym2,.. | 6.10 | Defines the symbol(s) specified as global symbol(s). |
| .IDENT symbol | 6.1.5 | Provides a means of labeling the object module with the program version number. The symbol is the version number between paired delimiting characters. |

| | | |
|---|---|---|
| .IF cond,arg1, arg2,... | 6.11 | Begins a conditional 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 | 6.11.1 | Appears only within a conditional block and indicates the beginning of a section of code to be assembled if the condition tested false. |
| .IFT | 6.11.1 | Appears only within a conditional block and indicates the beginning of a section of code to be assembled if the condition tested true. |
| .IFTF | 6.11.1 | Appears only within a conditional block and indicates the beginning of a section of code to be unconditionally assembled. |
| .IFF cond,arg, statement | 6.11.2 | Acts as a one-line conditional 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 | Indicates the beginning of an indefinite repeat block in which the symbol specified is replaced with successive elements of the real argument list (which is enclosed in angle brackets). |
| .IRPC sym,string | 7.6 | Indicates the beginning of an indefinite repeat block in which the symbol specified takes on the value of successive characters in the character string. |
| .LIMIT | 6.8 | Reserves two words into which the Task Builder inserts the low and high addresses of the relocated code. |
| .LIST<br>.LIST arg | 6.1.1 | Without an argument, .LIST increments the listing level count by 1. With an argument, .LIST does not alter the listing level count but formats the assembly listing according to the argument specified. |
| .MACRO sym,arg1, arg2,... | 7.1.1 | Indicates the start of a macro named sym containing the dummy arguments specified. |

| | | |
|---|---|---|
| .MEXIT | 7.1.3 | Causes an exit from the current macro or indefinite repeat block. |
| .NARG symbol | 7.4 | Appears only within a macro definition and equates the specified symbol to the number of arguments in the macro call currently being expanded. |
| .NCHR sym,string | 7.4 | Can appear anywhere in a source program; equates the symbol specified to the number of characters in the string (enclosed in delimiting characters). |
| .NLIST<br>.NLIST arg | 6.1.1 | Without an argument, .NLIST decrements the listing level count by 1. With an argument, .NLIST deletes the portion of the listing indicated by the argument. |
| .NTYPE sym,arg | 7.4 | Appears only in a macro definition and equates the low-order six bits of the symbol specified to the six-bit addressing mode of the argument. |
| .ODD | 6.5.1 | Ensures that the assembly location counter contains an odd address by adding 1 if it is even. |
| .PAGE | 6.1.6 | Causes the assembly listing to skip to the top of the next page. |
| .PSECT | 6.9 | Begin or resume a program section. |
| .PRINT exp,string | 7.5 | Causes a text string to be output to the command device containing the optional expression specified and the indicated text string. |
| .RADIX n | 6.4.1 | Alters the current program radix to n, where n can be 2, 4, 8, or 10. |
| .RAD50 string | 6.3.6 | Generates a block of data containing the Radix-50 equivalent of the character string (enclosed in delimiting characters). |
| .REPT exp | 7.7 | Begins a repeat block. Causes the section of code up to the next .ENDM or .ENDR to be repeated exp times. |

|  |  |  |
|---|---|---|
| .SBTTL string | 6.1.4 | Causes the string to be printed as part of the assembly listing page header. The string part of each .SBTTL directive is collected into a table of contents at the beginning of the assembly listing. |
| .TITLE string | 6.1.3 | Assigns the first symbolic name in the string to the object module and causes the string to appear on each page of the assembly listing. One .TITLE directive should be issued per program. |
| .WORD exp1,exp2,.. | 6.3.1 | Generates successive words of data containing the octal equivalent of the expression(s) specified. |

# APPENDIX C

## PERMANENT SYMBOL TABLE (PST)

The Permanent Symbol Table (PST) defines values for each symbol that is automatically recognized by MACRO. The symbols defined include op-codes and macro-calls. A listing of the Permanent Symbol Table forms the balance of this Appendix.

# APPENDIX D

# ERROR MESSAGE SUMMARY

## D.1 MACRO-11 ERROR CODES

MACRO-11 error codes are printed following a field of six asterisk characters and on the line preceding the source line containing the error. For example:

```
******A
26 00236  000002'   .WORD REL1+REL2
```

The addition of two relocatable symbols is flagged as an A error.

| Error Code | Meaning |
|---|---|
| A | Addressing error. An address within the instruction is incorrect. Also may indicate a relocation error. |
| B | Bounding error. Instructions or word data are being assembled at an odd address in memory. The location counter is updated by +1. |
| D | Doubly-defined symbol referenced. Reference was made to a symbol which is defined more than once. |
| E | End directive not found. (A listing is generated.) |
| I | Illegal character detected. Illegal characters which are also non-printing are replaced by a ? on the listing. The character is then ignored. |
| L | Line buffer overflow, i.e., input line greater than 132 characters. Extra characters on a line, (more than 72(10)) are ignored. |
| M | Multiple definition of a label. A label was encountered which was equivalent (in the first six characters) to a previously encountered label. |
| N | Number containing 8 or 9 has decimal point missing. |
| O | Opcode error. Directive out of context. |
| P | Phase error. A label's definition of value varies from one pass to another. A P error code also appears if a .ERROR directive is assembled. |
| Q | Questional syntax. There are missing arguments or the instruction scan was not completed or a carriage return was not immediately followed by a line feed or form feed. |

| | |
|---|---|
| R | Register-type error. An invalid use of or reference to a register has been made. |
| T | Truncation error. A number generated more than 16 bits of significance or an expression generated more than 8 bits of significance during the use of the .BYTE directive. |
| U | Undefined symbol. An undefined symbol was encountered during the evaluation of an expression. Relative to the expression, the undefined symbol is assigned a value of zero. |
| Z | Instruction which is not compatible among all members of the PDP-11 family (11/15, 11/20, 11/45). |

# APPENDIX E

# RECOMMENDED PROGRAMMING STANDARDS

## INTRODUCTION

Standards eliminate variablility and the requirement to make a decision; they need not be optimal. Much of the difficulty in establishing standards stems from the notion that they should be optimal (but everyone has differing opinions regarding the optimality criteria). For the DOS/BATCH group, standards represent an agreement on certain aspects of the programming process.

This Appendix represents a minimal beginning, pointing toward an engineering discipline for software development. All DIGITAL and user programmers are encouraged to participate actively in its continuing evolution through suggestions for improvement.

## E.1 LINE FORMAT

All source lines shall consist of from one to a maximum of 80 characters. Assembly language code lines shall have the following format:

1. Label Field - if present the label shall start at tab stop 0 (column 1).
2. Operation field - the operation field shall start at tab stop 1 (column 9).
3. Operand field - the operand field shall start at tab stop 2 (column 17).
4. Comments field - the comments field shall start at tab stop 4 (column 33) and may continue to column 80.

Comment lines that are included in the code body shall be delimited by a line containing only a leading semicolon. The comment itself contains a leading semicolon and starts in column 3. Indents shall be 1 tab.

If the operand field extends beyond Tab Stop 4 (column 33) simply leave a space and start the comment. Comments which apply to an instruction but require continuation should always line up with the character position which started the comment.

## E.2 COMMENTS

Comment all coding to convey the global role of an instruction and not simply a literal translation of the instruction into English. In general this will consist of a comment per line of code. If a particularly difficult, obscure, or elegant instruction sequence is used, a paragraph of comments shall immediately precede that section of code.

Preface text describing formats, algorithms, program-local variables, etc. will be delimited by the character sequence ;+ at the start of the text and ;- at the end. The comment will start in column 3.

For example:

```
;+

; The invert routine accepts

; a list of random numbers and

; applies the Kolmogorov algorithm

; to alphabetize them.

;-
```

Target labels for branches that exist solely for positional reference will use local labels of the form

```
<num>$:
```

Use of non-local labels is restricted, within reason, to those cases where reference to the code occurs external to the code. Local-labeling is formatted such that the numbers proceed sequentially down the page and from page to page.

## E.4 PROGRAM MODULES

### E.4.1 General Comments on Programs

In DOS/BATCH, a program provides a single distinct function. No limits exist on size, but the single function limitation should make modules larger than 1K a rarity. Since DOS/BATCH may eventually exploit the virtual memory capacity of the 11/40 and 11/45, programs should make every attempt to maintain a dense reference locus (don't promiscuously branch over page boundaries or over a large absolute address distance).

All code is read-only. Code and data areas are distinct and each contains explanatory text. Read-only data should be segregated from read-write data.

### E.4.2 The Module Preface

Program modules adhere to a strict format. This format adds to the readability and understandability of the module. The following sections are included in each module:

For the Code Section:

1. A .TITLE statement that specifies the name of the module.

2. A .PSECT statement that defines the program section in which the module resides. If a module contains more than one routine, subtitles may be used.

3. A copyright statement, and the disclaimer.

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

4. The version number of the file.
   Note: Items 1-5 must appear on the same page. The PDP-11 version number standard is described in Section 9.0.

5. The name of the principal author and the date on which the module was first created.

6. The name of each modifying author and the date of modification, name and modification dates appear one per line and in chronological order.

7. A brief statement of the function of the module.

8. A list of the definitions of all equated local symbols used in the module. These definitions appear one per line and in alphabetical order.

9. All local Macro definitions, preferably in alphabetical order by name.

10. All local data. The data should indicate

    a. Description of each element (type, size, etc.)
    b. Organization (functional, alpha, adjacent, etc.)
    c. Adjacency requirements

11. A list of the inputs expected by the module. This includes the calling sequence, condition code settings, and global data settings.

12. A list of the outputs produced as a result of entering this module. These include delivered results, condition code settings, but not side effects. (All these outputs are visible to the caller.)

13. A list of all effects (including side effects) produced as a result of entering this module. Effects include alterations in the state of the system not explicitly expected in the calling sequence, or those not visible to the caller.

14. A more detailed definition of the function of the module.

15. The module code.

### E.4.3 Formatting the Module Preface

Rules

1. The first five items appear on the same page and will not have explicit headings.

2. Titles start at the left margin*; descriptive text is indented 1 tab position.

3. Items 7-14 will have headings which start at the left margin, preceded and followed by blank lines. Items which do not

---------------

*The left margin consists of a ; a space then the heading, so the text of the heading begins in column 3.

## E.5.0 FORMATTING STANDARDS

### E.5.1 Program Flow

Programs will be organized on the listing such that they flow down the page, even at the cost of an extra branch or jump.

```
                           -----------
                           !         !
                           ! process !
                           !         !
                           -----------
                                !\
                               /  \
                              /    \
              ----------- /TEST \-----------
                   !          \    /          !
         ----------          \  /         ----------
         !        !           \/          !        !
         !  BBB   !                       !  AAA   !
         !        !                       !        !
         ----------                       ----------
              !                                !
              !          -----------          !
              !          !         !           !
              ---------!  COMMON !----------
                         !         !
                         -----------
```

shall appear on the listing as:

```
        TST
        BNE BBB
AAA:.......
        .......
        .......
        .......
        B   CMN

BBB:.......
        .......
        .......
CMN:.......
        .......
        .......
```

Rather than:

```
        TST
        BE   BBB
AAA:.......
        .......
        .......
        .......
CMN:.......
        .......
        .......
        .......
BBB:.......
        .......
        .......
        .......
        B   CMN
```

E.5.2  Common Exits

A common exit appears as the last code sequence on the listing. The flow chart

```
 ----------  ----------    -----------  ----------
 !        !  !        !    !         !  !        !
 !   1    !  !   2    !    !    3    !  !   4    !
 !        !  !        !    !         !  !        !
 ----------  ----------    -----------  ----------
      !           !              !            !
      !           !              !            !
      !           !   ---------- !            !
      !           !   !        ! !            !
      ---------------->! EXIT   !<-------------
                      !        !
                      ----------
```

will appear on the listing as:

```
PR1:.......
          .......
          .......
          B  EXIT

PR2:.......
          .......
          .......
          B  EXIT

PR3:.......
          .......
          .......
          B  EXIT

PR4:.......
          .......
          .......

EXIT:
```

And not as

```
PR1:.......
          .......
          .......

EXIT:.......
          .......
          .......

PR2:.......
          .......
          .......
          B  EXIT

PR3:.......
          .......
          .......
          B  EXIT

PR4:.......
          .......
          .......
          B  EXIT
```

## E.5.3 Code with Interrupts Inhibited

Code that is executed with interrupts inhibited shall be flagged by a three semi-colon (;;;) comment delimiter.

```
                            ; EXEC INTERRUPT

     ..ERTZ:                ; ENABLE BY RETURNING
                            ; BY SYSTEM SUBROUTINES,

     BIS    #000340,PSEXP   ;;; INHIBIT INTERRUPTS
     BIT    #000340,+2(SP)  ;;; C
     BEQ    10$             ;;;  O
     RTT                    ;;;   M
                            ;;;    M
                            ;;;     E
                            ;;;      N
                            ;;;       T
                            ;;;        S
```

## E.6 PROGRAM SOURCE FILES

Source creation and maintenance shall be done in base levels. A base level is defined as a point at which the program source files have been frozen. From the freeze point to the next base level, corrections will not be made directly to the base level itself. Rather a file of corrections shall be accumulated for each file in the base level. Whenever an updated source file is desired, the correction file will be applied to the base file.

The accumulation of corrections shall proceed until a logical breaking point has occurred (i.e. a milestone or significant implementation point has been reached). At this time all accumulated corrections shall be applied to the previous base level to create a new base level. Correction files will then be started anew for the new base level.

## E.7 FORBIDDEN INSTRUCTION USAGE

1. The use of instructions or index words as literals of the previous instruction. For example:

   ```
   MOV    @PC,Register

   BIC    Src,Dst
   ```

   uses the bit clear instruction as a literal. This may seem to be a very "neat" way to save a word but what about maintaining a program using this trick? To compound the pathology, it will not execute properly if I/D space is enabled on the 11/45. In this case @PC is a D bank reference.

2. The use of the MOV instruction instead of a JMP instruction to transfer program control to another location. For example:

   ```
   MOV    #ALPHA,PC
   ```

   transfers control to location ALPHA. Besides taking longer to execute (2.3 microseconds for MOV vs. 1.2 for JMP) the use of MOV instead of JMP makes it nearly impossible to pick up someone else's program and tell where transfers of control take place. What if one would like to get a jump trace of the execution of a program (anybody every hear of a move trace?)? As a more general issue, perhaps even other operations such as ADD and SUB from PC should be discouraged. Possibly one or two words can be saved by using these operations but how many occurrences are there?

3. The seemingly "neat" use of all single word instructions where a one double-word instruction could be used and would execute faster. Consider the following instruction sequence:

```
CMP    -(R1),(-R1)

CMP    -(R1), -(R1)
```

The intent of this instruction sequence is to subtract 8 from register R1 (not to set condition codes). This can be accomplished in approximately 1/3 the time via a SUB instruction (9.4 vs. 3.8 microseconds) at no additional cost in memory space. Another question here is also, what if R1 is odd? SUB always wins since it will always execute properly and is always faster!

## E.8 RECOMMENDED CODING PRACTICE

### E.8.1 Conditional Branches

When using the PDP-11 conditional branch instructions, it is imperative that the correct choice be made between the signed and the unsigned branches.

| Signed | Unsigned |
|---|---|
| BGE | BHIS (BCC) |
| BLT | BLO |
| BGT | BHI |
| BLE | BLOS (BCS) |

A common pitfall is to use a signed branch (e.g. BGT) when comparing two memory addresses. All goes well until the two addresses have opposite signs; that is, one of them goes across the 16K (100000(8)) bound. This type of coding error usually shows itself as a result of re-linking at different addresses and/or a change in size of the program.

## E.9 PDP-11 VERSION NUMBER STANDARD

This is the PDP-11 Version Number Standard. It applies to all modules, parameter files, complete programs, and libraries which are written or caused to be written, as part of the PDP-11 Software Development effort. It is used to provide unique identification of all released, pre-released, and in-house software.

It is limited in that, as currently specified, only six characters of identification are used. Future implementations of the Macro Assembler, Task Builder, and Librarian should provide for at least nine characters, and possibly twelve. It is expected that this standard will be enhanced as the need arises.

Version Identifier = <form> <version> <edit> <patch>

| | |
|---|---|
| <form> | Used to identify a particular form of a module or program, where applicable, as in the case of |

LINK-11. One alphabetic character, if used, and null (i.e., a binary 0) if not used.

| | |
|---|---|
| <version> | Used to identify the release, or generation, of a program. Two decimal digits, starting at 00, and incremented at the discretion of the project in order to reflect what, in their opinion, is a major change. |
| <edit> | Used to identify the level to which a particular release, or generation, of a program or module has been edited. An edit is defined to be an alteration to the source form. Two decimal digits, beginning at 01, and incremented with each edit; null if no edits. |
| <patch> | Used to identify the level to which a particular release, or generation, of a program or module has been patched. A patch is defined as an alteration to a binary form. One alphabetic character, starting at B, and running sequentially toward Z, each time a set of patches is released; null if no patches. |

These fields are interrelated. When <version> is changed, then <patch> and <edit> must be reset to nulls. It is intended that when <edit> is incremented, then <patch> will be re-set to null, because the various bugs have been fixed.

## E.9.1 Displaying the Version Identifier

The visible output of the version identifier should appear as:

Key <letter> <form> <version> - <edit> <patch>,

where the following Key Letters have been identified:

| | |
|---|---|
| V | released or frozen version |
| X | in-house experimental version |
| Y | field test, pre-release, or in-house release version |

Note that 'X' corresponds roughly to individual support, 'Y' to group support, and 'V' to company support.

The dash which separates <version> from <edit> is used only if <edit> and/or <patch> is not null. When a version identifier is displayed as part of program identification, then the format is:

```
Program
        <space><key-letter><form><version>-<edit><patch>
Name
```

Examples:

```
PIP X03
LINK VB04-C
MACRO Y05-01
```

## E.9.2 Use of the Version Number in the Program

All sources must contain the version number in an .IDENT directive. For programs (or libraries) which consist of more than one module, each individual module will follow this version number standard. The version number of the program or library is not necessarily related to the version numbers of the constituent modules; it is perfectly reasonable, for example, that the first version of a new FORTRAN library, V00, contain an existing SIN routine, say V05-01.

Parameter files are also required to contain the version number in an .IDENT directive. Because the assembler records the last .IDENT seen, parameter files must precede the program.

Entities which consist of a collection of modules or programs, e.g., the FORTRAN Library, will have an identification module in the first position. An identification module exists solely to provide identification, and normally consists of something like:

```
;OTS IDENTIFICATION
.TITLE FTNLIB
.IDENT /003010/
.END
```

# APPENDIX F

# WRITING POSITION-INDEPENDENT CODE - A TUTORIAL

It is possible to write a source program that can be loaded and run in any section of virtual memory. Such a program is said to consist of position-independent code. The construction of position independent code is dependent upon the proper usage of PDP-11 addressing modes. (Addressing modes are described in detail in Chapter 5. The remainder of this Appendix assumes the reader is familiar with the various addressing modes.)

All addressing modes involving only register references are position-independent. These modes are as follows:

| | |
|---|---|
| R | register mode |
| @R | deferred register mode |
| (R)+ | autoincrement mode |
| @(R)+ | deferred autoincrement mode |
| -(R) | autodecrement mode |
| @-(R) | deferred autodecrement mode |

When using these addressing modes, position-independence is guaranteed providing the contents of the registers have been supplied such that they are not dependent upon a particular core location.

The relative addressing modes are generally position independent. These modes are as follows:

| | |
|---|---|
| A | relative mode |
| @A | relative deferred mode |

Relative modes are not position-independent when A is an absolute address (that is, a non-relocatable address) which is referenced from a relocatable module.

Index modes can be either position-independent or nonposition-independent, according to their use in the program. These modes are:

| | |
|---|---|
| X(R) | index mode |
| @X(R) | index deferred mode |

If the base, X, is position-independent, the reference is also position-independent. For example:

```
        MOV     2(SP),R0        ;POSITION-INDEPENDENT
        N=4
        MOV     N(SP),R0        ;POSITION-INDEPENDENT
        CLR     ADDR(R1)        ;NONPOSITION-INDEPENDENT
```

Caution must be exercised in the use of index modes in position independent code.

Immediate mode can also be either position-independent or not, according to its usage. Immediate mode references are formatted as

If the symbol is absolute, the reference is flagged and is not position-independent.

4. Immediate mode references to symbolic labels are always flagged with an ' character.

```
MOV  #3,R0           ;ALWAYS POSITION-INDEPENDENT.
MOV  #ADDR,R1        ;NON-PIC WHEN ADDR IS RELOCATABLE.
```

Examples of assembly listings contining the ' character are shown below:

```
011744          ENDP2:                             ;END OF PASS 2
                         .IF NDF XCREF
011744 016702            MOV     CRFPNT,R2         ;ANY CREF IN PROGRESS?
       000142'
011750 001402            BEQ     8$                ;  NO
011752                   CALL    CRFDMP            ;YES, DUMP AND CLOSE BUFFER
011756          8$:
                         .ENDC
011756 005767            TST     BLKTYP            ;ANY OBJECT OUTPUT?
       000542'
011762 001423            BEQ     1$                ;  NO
 11764                   CALL    OBJDMP            ;YES, DUMP IT
 11770 012767            MOV     #BLKT06,BLKTYP    ;SET END
       000006
       000542'
 11776                   CALL    RLDDMP            ;DUMP IT
                         .IF NDF XEDABS
 12002 032767            BIT     #ED.ABS,EDMASK    ;ABS OUTPUT?
       000002
       000124'
 12010 001010            BNE     1$                ;  NO
 12012 016700            MOV     OBJPNT,R0
       000536'
 12016 016720            MOV     ENDVEC+6,(R0)+    ;SET END VECTOR
       000244'
 12022 010067            MOV     R0,OBJPNT
       000536'
 12026                   CALL    OBJDMP
                         .ENDC
 12032 105767 1$:        TSTB    LLTBL+2           ;ANY LISTING OUTPUT?
       000546'
 12036 001474            BEQ     15$               ;  NO
 12040 032767            BIT     #LC.SYM,LCMASK    ;SYMBOL TABLE SUPPRESSION?
       040000
       000110'
```

# INDEX

## HOW TO OBTAIN SOFTWARE INFORMATION

Announcements for new and revised software, as well as programming notes, software problems, and documentation corrections, are published by Software Information Service in the following newsletters.

DIGITAL Software News for the PDP-8 and PDP-12
DIGITAL Software News for the PDP-11
DIGITAL Software News for 18-bit Computers

These newsletters contain information applicable to software available from DIGITAL'S Software Distribution Center. Articles in DIGITAL Software News update the cumulative Software Performance Summary which is included in each basic kit of system software for new computers. To assure that the monthly DIGITAL Software News is sent to the appropriate software contact at your installation, please check with the Software Specialist or Sales Engineer at your nearest DIGITAL office.

Questions or problems concerning DIGITAL'S software should be reported to the Software Specialist. If no Software Specialist is available, please send a Software Performance Report form with details of the problems to:

Digital Equipment Corporation
Software Information Service
Software Engineering and Services
Maynard, Massachusetts 01754

These forms, which are provided in the software kit, should be fully completed and accompanied by terminal output as well as listings or tapes of the user program to facilitate a complete investigation. An answer will be sent to the individual, and appropriate topics of general interest will be printed in the newsletter.

Orders for new and revised software manuals, additional Software Performance Report forms, and software price lists should be directed to the nearest DIGITAL field office or representative. USA customers may order directly from the Software Distribution Center in Maynard. When ordering, include the code number and a brief description of the software requested.

Digital Equipment Computer Users Society (DECUS) maintains a user library and publishes a catalog of programs as well as the DECUSCOPE magazine for its members and non-members who request it. For further information, please write to:

Digital Equipment Corporation
DECUS
Software Engineering and Services
Maynard, Massachusetts 01754

DOS/BATCH
Assembler (MACRO)
Programmer's Manual
DEC-11-LASMA-A-D

READER'S COMMENTS

Digital Equipment Corporation maintains a continuous effort to improve the quality and usefulness of its publications. To do this effectively we need user feedback--your critical evaluation of this document.

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

______________________________________________________________________

______________________________________________________________________

______________________________________________________________________

______________________________________________________________________

______________________________________________________________________

How can this document be improved?

______________________________________________________________________

______________________________________________________________________

______________________________________________________________________

______________________________________________________________________

______________________________________________________________________

How does this document compare with other technical documents you have read?

______________________________________________________________________

______________________________________________________________________

______________________________________________________________________

______________________________________________________________________

______________________________________________________________________

Job Title______________________________________Date:__________________

Name:______________________________Organization:_____________________

Street:____________________________Department:_______________________

City:_____________________State:__________Zip or Country_____________

------------------------------------------------------------ Fold Here ------------------------------------------------------------

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

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

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

Postage will be paid by:

digital

**Digital Equipment Corporation**
**Software Information Service**
**Software Engineering and Services**
**Maynard, Massachusetts 01754**

CONTENTS

For example:

```
A = 1                               ;THE SYMBOL A IS EQUATED TO THE
                                    ;VALUE 1.

B = 'A-1&MASKLOW                    ;THE SYMBOL B IS EQUATED TO THE
                                    ;VALUE OF THE EXPRESSION

C:     D = 3                        ;THE SYMBOL D IS EQUATED TO 3.

E:     MOV      #1,ABLE             ;LABELS C AND E ARE EQUATED TO THE
                                    ;LOCATION OF THE MOV COMMAND
```

The following conventions apply to direct assignment statements:

1. An equal sign (=) or double equal (==) must separate the symbol from the expression defining the symbol value.

2. A direct assignment statement is usually placed in the label field and may be followed by a comment.

3. Only one symbol can be defined in a single direct assignment statement.

4. Only one level of forward referencing is allowed.

Example of two levels of forward referencing (illegal):

```
X = Y

Y = Z

Z = 1
```

## 3.4 REGISTER SYMBOLS

The eight general registers of the PDP-11 are numbered 0 through 7 and can be expressed in the source program as:

```
        %0
        %1
         .
         .
         .
        %7
```

where the digit indicating the specific register can be replaced by any legal term which can be evaluated during the first assembly pass.

It is recommended that the programmer use symbolic names for all register references. Unless the .DSABL REG statement has been encountered, the definitions as shown in the following example are defined by default, or, a register symbol may be defined in a direct assignment statement, among the first statements in the program. The defining expression of a register symbol must be absolute. For example:

```
R0=%0                                   ;REGISTER DEFINITION
R1=%1
R2=%2
R3=%3
R4=%2
R5=%5
SP=%6
PC=%7
```

The user can reassign the register expressions, if he wishes.

The symbolic names assigned to the registers in the example above are the conventional names used in all PDP-11 system programs. Since these names are mnemonic, it is suggested the user follow this convention. Note that registers 6 and 7 are given special names because of their special functions.

All register symbols must be defined before they are referenced. A forward reference to a register symbol is flagged as an error.

The % character may be used with any term or expression to specify a register. (A register expression less than 0 or greater than 7 is flagged with an R error code.) For example:

```
        CLR     %3+1
```

is equivalent to

```
        CLR     %4
```

and clears the contents of register 4, while

```
        CLR     4
```

clears the contents of memory address 4.

## 3.5 LOCAL SYMBOLS

Local symbols are specially formatted symbols used as labels within a given range.

Local symbols provide a convenient means of generating labels for branch instructions, etc. Use of local symbols reduces the possibility of multiply-defined symbols within a user program and separates entry point symbols from local references. Local symbols may not be referenced from other object modules or even from outside their local symbol block. The rules for delimiting a local symbol block appear shortly.

Local symbols are of the form n$ where n is a decimal integer from 1 to 65535, inclusive, and can only be used on word boundaries (i.e., at even addresses). Local symbols include:

```
             1$
            27$
            59$
           104$
```

Within a local symbol block, local symbols can be defined and referenced. However, a local symbol cannot be referenced outside the block in which it is defined. There is no conflict with labels of the same name in other local symbol blocks.

Local symbols 64$ throgh 127$ can be generated automatically as a feature of the macro processor (see section 7.3.5 for further details). When using local symbols the user is advised to first use the range from 1$ to 63$, or the range from 128$ to 65535$.

A local symbol block is delimited in one of the following ways:

1. The range of a single local symbol block can consist of those statements between two normally constructed symbolic labels. (Note that a statement of the form

   ```
   LABEL=.
   ```

   is a direct assignment, does not create a label in the strict sense, and does not delimit a local range.)

2. The range of a local symbol block is always terminated upon encountering a .PSECT, .CSECT, or .ASECT directive.

3. The range of a single local symbol block can be delimited with .ENABL LSB and the first symbolic label or .PSECT, .CSECT, or .ASECT directive following .DSABL LSB directive. The default for LSB is off.

For examples of local symbols and local symbol blocks, see Figure 3-3.

```
 Line     Octal
Number  Expansion        Source Code                          Comments
  1                        .SBTTL  SECTOR INITIALIZATION
  2
  3         000000'        .CSECT  IMPURE           ;IMPURE STORAGE AREA
  4 000000         IMPURE:
  5         000000'        .CSECT  IMPPAS           ;CLEARED EACH PASS
  6 000000         IMPPAS:
  7         000000'        .CSECT  IMPLIN           ;CLEARED EACH LINE
  8 000000         IMPLIN:
  9
 10         000000'        .CSECT  XCTPRG           ;PROGRAM INITIALIZATION CODE
 11  00000         XCTPRG:
 12  00000 012700          MOV     #IMPURE,R0
            000000'
 13  00004 005020  1$:     CLR     (R0)+            ;CLEAR IMPURE AREA
 14  00006 022700          CMP     #IMPTOP,R0
            000040'
 15  00012 101374          BHI     1$
 16
 17         000000'        .CSECT  XCTPAS           ;PASS INITIALIZATION CODE
 18  00000         XCTPAS:
 19  00000 012700          MOV     #IMPPAS,R0
            000000'
 20  00004 005020  1$:     CLR     (R0)+            ;CLEAR IMPURE PART
 21  00006 022700          CMP     #IMPTOP,R0
            000040'
 22  00012 101374          BHI     1$
 23
 24         000000'        .CSECT  XCTLIN           ;LINE INITIALIZATION CODE
 25  00000         XCTLIN:
 26  00000 012700          MOV     #IMPLIN,R0
            000000'
 27  00004 005020  1$:     CLR     (R0)+
 28  00006 022700          CMP     #IMPTOP,R0
            000040'
 29  00012 101374          BHI     1$
 30
 31         000000'        .CSECT  MIXED            ;MIXED MODE SECTOR
```

Figure 3-3

Assembly Source Listing of MACRO-11 Code Showing Local Symbol Blocks

## 3.6 ASSEMBLY LOCATION COUNTER

The period (.) is the symbol for the assembly location counter. When used in the operand field of an instruction, it represents the address of the first word of the instruction. When used in the operand field of an assembler directive, it represents the address of the current byte or word. For example:

```
A:      MOV     #.,R0           ;. REFERS TO LOCATION A,
                                ;I.E., THE ADDRESS OF THE
                                ;MOV INSTRUCTION.
```

(# is explained in section 5.9.)

At the beginning of each assembly pass, the Assembler clears the location counter. Normally, consecutive memory locations are assigned to each byte of object data generated. However, the location where the object data is stored may be changed by a direct assignment altering the location counter:

```
        .=expression
```

Similar to other symbols, the location counter symbol has a mode associated with it, either absolute or relocatable. However, the mode cannot be external. The existing mode of the location counter cannot be changed by using a defining expression of a different mode.

The mode of the location counter symbol can be changed by the use of the .ASECT,.CSECT or .PSECT directives as explained in section 6.9.

The expression defining the location counter must not contain forward references or symbols that vary from one pass to another.

Examples:

```
        .ASECT

.=500                           ;SET LOCATION COUNTER TO
                                ;ABSOLUTE 500

FIRST:  MOV     .+10,COUNT      ;THE LABEL FIRST HAS THE VALUE
                                ;500(OCTAL)
                                ;.+10 EQUALS 510(OCTAL). THE
                                ;CONTENTS OF THE LOCATION
                                ;510(OCTAL) WILL BE DEPOSITED
                                ;IN LOCATION COUNT.

.=520                           ;THE ASSEMBLY LOCATION COUNTER
                                ;NOW HAS A VALUE OF
                                ;ABSOLUTE 520(OCTAL).

SECOND: MOV     .,INDEX         ;THE LABEL SECOND HAS THE
                                ;VALUE 520(OCTAL)
                                ;THE CONTENTS OF LOCATION
                                ;520(OCTAL), THAT IS, THE BINARY
                                ;CODE FOR THE INSTRUCTION
                                ;ITSELF, WILL BE DEPOSITED IN
                                ;LOCATION INDEX.
        .PSECT

.=.+20                              ;SET LOCATION COUNTER TO
                                    ;RELOCATABLE 20 OF THE
                                    ;UNNAMED PROGRAM SECTION.

THIRD:  .WORD   0                   ;THE LABEL THIRD HAS THE
                                    ;VALUE OF RELOCATABLE 20.
```

Storage area may be reserved by advancing the location counter. For example, if the current value of the location counter is 1000, the direct assignment statements:

```
.=.+40

; or
        .BLKB 40
; or
        .BLKW 20
```

reserve 40(octal) bytes of storage space in the program. The next instruction is stored at 1100. (The .BLKB and .BLKW directives are recommended as the preferred ways to reserve space. Refer to section 6.5.3.)

## 3.7 NUMBERS

The MACRO-11 Assembler assumes all numbers in the source program are to be interpreted in octal radix unless otherwise specified. The assumed radix can be altered with the .RADIX directive (see section 6.4.1) or individual numbers can be treated as being of decimal, binary, or octal radix (see section 6.4.2).

Octal numbers consist of the digits 0 through 7 only. A number not specified as a decimal number and containing an 8 or 9 is flagged with an N error code and treated as a decimal number.

Negative numbers are preceded by a minus sign (the Assembler translates them into two's complement form). Positive numbers may be preceded by a plus sign, although this is not required.

A number which is too large to fit into 16 bits (177777<n) is truncated from the left and flagged with a T error code in the assembly listing.

Numbers are always considered absolute quantities (that is, not relocatable).

Single-word floating-point numbers may be generated with the ↑F operator (see section 6.6.2) and are stored in the following format:

Refer to PDP-11/45 Processor Handbook for details of the floating-point format.

## 3.8 TERMS

A term is a component of an expression. A term may be one of the following:

1. A number, as defined in section 3.7, whose 16-bit value is used.
2. A symbol, as defined earlier in the Chapter. Symbols are interpreted according to the following hierarchy:
   a. A period causes the value of the current location counter to be used.
   b. A permanent symbol's basic value is used but its arguments (if any) are ignored;
   c. An undefined symbol is assigned a value of zero and inserted in the user-defined symbol table as an undefined global reference. If the .DSABL GBL directive is in effect, the automatic global reference default function is inhibited, in which case the symbol is not defined as a global reference. It is simply undefined. Refer to section 6.2.
3. An ASCII conversion using either an apostrophe followed by a single ASCII character, or a double quote followed by two ASCII characters, which results in a word containing the 7-bit ASCII value of the character(s). (This construction is explained in greater detail in section 6.3.3.)
4. A term may also be an expression or term enclosed in angle brackets. Any quantity enclosed in angle brackets is evaluated before the remainder of the expression in which it is found. Angle brackets are used to alter the left-to-right evaluation of expressions (to differentiate between A*B+C and A* B+C ) or to apply a unary operator to an entire expression (- A+B , for example).

## 3.9 EXPRESSIONS

Expressions are combinations of terms joined together by binary operators and which reduce to a 16-bit value. The operands of a .BYTE directive (see section 6.3.1) are evaluated as word expressions before truncation to the low-order eight bits. Prior to truncation, the high-order byte must be zero or all ones (when byte value is negative, the sign bit is propagated). The evaluation of an expression includes the evaluation of the mode of the resultant expression; that is, absolute, relocatable or external. Expression modes are further defined below.

Expressions are evaluated left to right with no operator hierarchy rules except that unary operators take precedence over binary operators. A term preceded by a unary operator can be considered as containing that unary operator. (Terms are evaluated, where necessary, before their use in expressions.) Multiple unary operators are valid and are treated as follows:

```
-+-A
```

is equivalent to:

```
-<+<-A>>
```

A missing term, expression, or external symbol is interpreted as a zero. A missing operator is interpreted as +. A Q error flag is generated for each missing term or operator. For example (here TAG is OR'ed with LA +177777):

```
TAG ! LA 177777
```

is evaluated as

```
TAG ! LA+177777
```

with a Q error flag on the assembly listing line.

The value of an external expression is the value of the absolute part of the expression; e.g., EXTERNAL+A has a value of A. This is modified by LINK to become EXTERNAL+A.

Expressions, when evaluated, are either absolute, relocatable, or external. For the programmer writing position-independent code, the distinction is important.

1. An expression is absolute if its value is fixed. An expression whose terms are numbers and ASCII conversions will have an absolute value. A relocatable expression minus a relocatable term, where both items belong to the same program section, is also absolute.

2. An expression is relocatable if its value is fixed relative to a base address but will have an offset value added at Task Build time. Expressions whose terms contain labels defined in relocatable sections and periods, (in relocatable sections) will have a relocatable value.

```
        .ASCII  <101>               ;EQUIVALENT TO .ASCII/A/

        .RAD50  /AB/<35>            ;STORES 3255 IN NEXT WORD

CHR1=1
CHR2=2
CHR3=3
              .
              .
              .
        .RAD50  <CHR1><CHR2><CHR3>
                                    ;EQUIVALENT TO .RAD50/ABC/
```

## 6.4 RADIX CONTROL

### 6.4.1 .RADIX

Numbers used in a MACRO-11 source program are initially considered to be octal numbers. However, the programmer has the option of declaring the following radices:

```
          2, 4, 8, 10
```

This is done via the .RADIX directive, of the form:

```
          .RADIX  n
```

where: n is one of the acceptable radices.

The argument to the .RADIX directive is always interpreted in decimal radix. Following any radix directive, that radix is the assumed base for any number specified until the following .RADIX directive.

The default radix at the start of each program, and the argument assumed if none is specified, is 8 (i.e., octal). For example:

```
        .RADIX 10                   ;BEGINS SECTION OF CODE WITH
                                    ;DECIMAL
                                    ;RADIX
              .
              .
              .
        .RADIX                      ;REVERTS TO OCTAL RADIX
```

In general it is recommended that macro definitions not contain or rely on radix settings from the .RADIX directive. The temporary radix control characters should be used within a macro definition. (↑D, ↑O, and ↑B are described in the following section.) A given radix is valid 5dughout a program until changed. Where a possible conflict exists within a macro definition or in possible future uses of that code module, it is suggested that the user specify values using the temporary radix controls (see below).

### 6.4.2 Temporary Radix Control: ↑D, ↑O, and ↑B

Once the user has specified a radix for a section of code, or has determined to use the default octal radix, he may discover a number of cases where an alternate radix is more convenient (particularly within macro definitions). For example, the creation of a mask word might best be done in the binary radix.

MACRO-11 has three unary operators to provide a single interpretation in a given radix within another radix as follows:

```
↑Dx  (x is treated as being in decimal radix)
↑Ox  (x is treated as being in octal radix)
↑Bx  (x is treated as being in binary radix)
```

For example:

```
↑D123
↑0 47
↑B 00001101
↑0<A+3>
```

Notice that while the up arrow and radix specification characters may not be separated, the radix operator can be physically separated from the number by spaces or tabs for formatting purposes. Where a term or expression is to be interpreted in another radix, it should be enclosed in angle brackets.

These numeric quantities may be used any place where a numeric value is legal.

PAL-11R contains a feature, which is maintained for compatibility in MACRO-11, allowing a temporary radix change from octal to decimal by specifying a decimal radix number with a "decimal point". For example:

```
 100.     (144(8))
1376.     (2540(8))
 128.     (200(8))
```

## 6.5 LOCATION COUNTER CONTROL

The four directives which control movement of the location counter are .EVEN and .ODD, which move the counter a maximum of one byte, and .BLKB and .BLKW, which allow the user to specify blocks of a given number of bytes or words to be skipped in the assembly.

## 6.5.1 .EVEN

The .EVEN directive ensures that the assembly location counter contains an even memory address by adding one if the current address is odd. If the assembly location counter is even, no action is taken. Any operands following an .EVEN directive are ignored.

The .EVEN directive is used as follows:

```
        .ASCIZ /THIS IS A TEST/
        .EVEN                      ;ASSURES NEXT STATEMENT
                                   ;BEGINS ON A WORD BOUNDARY.
        .WORD XYZ
```

## 6.5.2 .ODD

The .ODD directive ensures that the assembly location counter is odd by adding one if it is even. For example:

```
; CODE TO MOVE DATA FROM AN INPUT LINE
; TO A BUFFER

N=5                                ;BUFFER HAS 5 WORDS
        .
        .
        .
        .ODD
        .BYTE   N*2                ;COUNT=2N BYTES
BUFF:   .BLKW   N                  ;RESERVE BUFFER OF N WORDS
        .
        .
        .
        MOV     #BUFF,R2           ;ADDRESS OF EMPTY BUFFER IN R2
        MOV     #LINE,R1           ;ADDRESS OF INPUT LINE IS IN R1
        MOVB    -1(R2),R0          ;GET COUNT STORED IN BUFF-1 IN R0
AGAIN:  MOVB    (R1)+,(R2)+        ;MOVE BYTE FROM LINE INTO BUFFER
        BEQ     DONE               ;WAS NULL CHARACTER SEEN?
        DEC     R0                 ;DECREMENT COUNT
        BNE     AGAIN              ;NO = 0, GET NEXT CHARACTER
        .
        .
        .
        CLRB    -(R2)              ;OUT OF ROOM IN BUFFER, CLEAR LAST
DONE:                              ;WORD
        .
        .
        .
LINE:   .ASCIZ  /TEXT/
```

In this case, .ODD is used to place the buffer byte count in the byte preceding the buffer, as follows:

```
          COUNT             BUFF-2

                            BUFF
```

### 6.5.3 .BLKB and .BLKW

Blocks of storage can be reserved using the .BLKB and .BLKW directives. .BLKB is used to reserve byte blocks and .BLKW reserves word blocks. The two directives are of the form:

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

where: exp — is the number of bytes or words to reserve. If no argument is present, 1 is the assumed default value. Any legal expression which is completely defined at assembly time and produces an absolute number is legal. Using these directives without arguments is not recommended.

For example:

```
          000000'         .CSECT   IMPURE

  000000         PASS:    .BLKW
                                                ;NEXT GROUP MUST STAY TOGETHER
  000002         SYMBOL:  .BLKW    2            ;SYMBOL ACCUMULATOR
  000006         MODE:
  000006         FLAGS:   .BLKB    1            ;FLAG BITS
  000007         SECTOR:  .BLKB    1            ;SYMBOL/EXPRESSION TYPE
  000010         VALUE:   .BLKW    1            ;EXPRESSION VALUE
   00012         RELLVL:  .BLKW    1
                          .BLKW    2            ;END OF GROUPED DATA

   00020         CLCNAM:  .BLKW    2            ;CURRENT LOCATION COUNTER SYMBOL
   00024         CLCFGS:  .BLKB    1
   00025         CLCSEC:  .BLKB    1
   00026         CLCLOC:  .BLKW    1
   00030         CLCMAX:  .BLKW    1
```

The .BLKB directive has the same effect as:

```
          .=.+exp
```

but is easier to interpret in the context of source code.

## 6.6 NUMERIC CONTROL

Several directives are available to simplify the use of the floating-point hardware on the PDP-11.

A floating-point number is represented by a string of decimal digits. The string (which can be a single digit in length) may optionally contain a decimal point, and may be followed by an optional exponent indicator in the form of the letter E and a signed decimal exponent. The list of number representations below contains seven distinct, valid representations of the same floating-point number:

```
3
3.
3.0
3.0E0
3E0
.3E1
300E-2
```

As can be quickly inferred, the list could be extended indefinitely (e.g., 3000E-3, .03E2, etc.). A leading plus sign is ignored (e.g., +3.0 is considered to be 3.0). A leading minus sign complements the sign bit. No other operators are allowed (e.g., 3.0+N is illegal).

Floating-point number representations are valid only in the contexts described in the remainder of this section.

Floating-point numbers are normally rounded. That is, when a floating-point number exceeds the limits of the field in which it is to be stored, the high-order excess bit is added to the low-order retained bit. For example, if the number is to be stored in a 2-word field, but more than 32 bits are needed for its value, the highest bit carried out of the field is added to the least significant position. The .ENABL FPT directive is used to enable floating-point truncation, and .DSABL FPT is used to return to floating-point rounding (see section 6.2).

### 6.6.1 .FLT2 and .FLT4

Like the .WORD directive, the two floating-point storage directives cause their arguments to be stored in-line with the source program. These two directives are of the form:

```
.FLT2   arg1,arg2,...
.FLT4   arg1,arg2,...
```

| | | |
|---|---|---|
| where: | arg1,arg2,... | represent one or more floating point numbers separated by commas. |

.FLT2 causes two words of storage to be generated for each argument, while .FLT4 generates four words of storage.

## 6.6.2 Temporary Numeric Control: ↑F and ↑C

Like the temporary radix control operators, operators are available to specify either a 1-word floating-point number (↑F) or the 1's complement of a 1-word number (↑C). The ↑F operator can only be used within an instruction operand expression. ↑C can be used in any expression. For example:

```
FL3.7:  MOV     #↑F3.7,R0
```

creates a 1-word floating-point number at location FL3.7+2 containing the value 3.7 formatted as follows:

```
 15       6     0
------------------
!SEEEEEEEEMMMMMMM!
------------------
 ↑    ↑       ↑
 !    !       !
 !    !       ---Mantissa (bits 0-6)
 !    !
 !    ---Exponent (bits 7-14)
 !
 ---Sign (bit 15)
```

This 1-word floating-point number is the first word of the 2- or 4-word floating-point number format shown in the PDP-11 Processor Handbook, and the statement:

```
CMP151: .WORD   ↑C151
```

stores the 1's complement of 151 in the current radix (assume current radix is octal) as follows (177626 shown in binary)

```
------------------
!1111111110010110!
------------------
 1  7  7  6  2  6
```

Since these control operators are unary operators, their arguments may be terms, and the operators may be expressed recursively. For example:

```
↑F<1.2E3>
↑C<D25>    or    ↑C31      or    177746
```

The term created by the unary operator and its argument is then a term which can be used by itself or in an expression. For example:

```
↑C2+6
```

is equivalent to:

```
<↑C2>+6    or    177775+6   or    000003
```

For this reason, the use of angle brackets is advised. Expressions used as terms or arguments of a unary operator must be explicitly grouped.

An example of the importance of ordering with respect to unary operators is shown below:

```
            ↑F1.0      - 020400
            ↑F-1.0     - 120400

           -↑F1.0      = 157400
           -↑F-1.0     = 057400
```

The argument of the ↑F operator must not be an expression and must be of the same format as arguments to the .FLT2 and .FLT4 directives (see section 6.6.1).

## 6.7 TERMINATING DIRECTIVES

### 6.7.1 .END

The .END directive indicates the physical end of the source program. The .END directive is of the form:

```
            .END        exp
```

where: exp — is an optional argument which, if present, indicates the program entry point, i.e., the transfer address.

When the load module is loaded, program execution begins at the transfer address indicated by the .END exp directive. In a runtime system (the load module output of LINK) an .END exp statement should terminate the first object module and .END statements should terminate any other object modules.

### 6.7.2 .EOT

Under the DOS/BATCH Monitor, the .EOT directive is ignored.

## 6.8 PROGRAM BOUNDARIES DIRECTIVE: .LIMIT

It is often important to know the boundaries of the load module's relocatable code. The .LIMIT directive reserves two words into which LINK puts the low and high addresses of the relocated code. The low address (inserted into the first word) is the address of the first byte of code. The high address is the address of the first free byte following the relocated code. These addresses are always even since all relocatable sections are loaded at even addresses. (If a relocatable section consists of an odd number of bytes, LINK adds one to the size to make it even.)

## 6.9 PROGRAM SECTION DIRECTIVES

### 6.9.1 .PSECT Directive

Program sections are defined by the .PSECT directive, which is formatted as:

```
.PSECT [NAME][,RO/RW][,I/D][,GBL/LCL][,ABS/REL][,CON/OVR][,HGH/LOW]
```

The brackets ([]) are for purposes of illustrating optional parameters, and are not included in the parameter specifications. The slash (/) indicates that a choice is to be made between the parameters. The program section attribute parameters are summarized in Table 6-2.

Table 6-2

.PSECT Directive Parameters

| Parameter | Default | Meaning |
|---|---|---|
| NAME | Blank | Program section name, in Radix-50 format, specified as one to six characters. If omitted, a comma must appear in the first parameters position. |
| RO/RW | RW | Program section access mode;<br>RO=Read Only<br>RW=Read/Write |
| I/D | I | Program section type;<br>I=Instruction<br>D=Data |
| GBL/LCL | LCL | The scope of the program section, as interpreted by LINK;<br>GBL=Global<br>LCL=Local |
| ABS/REL | REL | Defines relocation of the program section;<br>ABS=Absolute (no relocation)<br>REL=Relocatable (a relocation bias is required) |
| CON/OVR | OVR | Program section allocation;<br>CON=Concatenated<br>OVR=Overlaid |

| | | |
|---|---|---|
| HGH/LOW | LOW | Program section memory type;<br><br>HGH=High-speed<br>LOW=Core |

***NOTE***
The HGH/LOW attribute is currently ignored by LINK.

The only parameter that is position-dependent is NAME. If it is omitted, a comma must be used in its place. For example,

```
        .PSECT ,RO
```

This example shows a PSECT with a blank name and the Read Only access parameter. Defaults are used for the remaining parameters.

LINK interprets the .PSECT directive's parameters as follows:

| | |
|---|---|
| RO/RW | Defines the type of access to the program section permitted which is; Read Only, or Read/Write. |
| I/D | Allows LINK to differentiate global symbols that are entry points (I) from global symbols that are data values (D). |
| GBL/LCL | Defines the scope of a program section. A global program section's scope crosses segment (overlay) boundaries; a local program section's scope is within a single segment. In single-segment programs, the GBL/LCL parameter is ignored. |
| ABS/REL | When ABS is specified, the program section is absolute. No relocation is necessary (i.e., the program section is assembled starting at absolute virtual 0). When REL is specified, a relocation bias is calculated by LINK, and added to all references in the section. |
| CON/OVR | CON causes LINK to collect all allocation references to the program section from different modules and concatenate them to form the total allocation for the program section. OVR indicates that all allocation references to the program section overlay one another. Thus, the total allocation of the program section is determined by the largest request made by a module that references it. |

Once the attributes of a named .PSECT are declared in a module, the MACRO-11 Assembler assumes that this .PSECT's attributes hold for all subsequent declarations of the named .PSECT in the same module. Thus, the attributes may be declared once, and later .PSECT's with the same name will have the same attributes, when specified within the same module.

The Assembler provides for 255(10) program sections: One absolute section, one blank relocatable section, and 253(10) named relocatable sections are permitted. The .PSECT directive enables the user to:

1. Create his program (object module) in sections; and,

2. Share code and data.

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

1. Section name;

2. Contents of the program counter;

3. Maximum program counter value encountered; and,

4. Section attributes, (the six .PSECT attributes).

### 6.9.1.1 Creating Program Sections

A given program section is defined completely upon its first reference. Thereafter, the section can be referenced by completely specifying the section attributes or by specifying the name only. For example, a section can be specified as:

```
          .PSECT    ALPHA,ABS,OVR
```

and later referenced as:

```
          .PSECT    ALPHA
```

By maintaining separate location counters for each section, the Assembler allows the user to write statements which are not physically contiguous but are loaded contiguously, as shown in the following example:

```
        .PSECT  SEC1,REL          ;START A RELOCATABLE SECTION NAMED
A:      .WORD   0                 ;SEC1 ASSEMBLED AT RELOCATABLE 0,
B:      .WORD   0                 ;RELOCATABLE 2 AND
C:      .WORD   0                 ;RELOCATABLE 4,
ST:     CLR A                     ;ASSEMBLE CODE AT
        CLR B                     ;RELOCATABLE ADDRESSES
        CLR C                     ;6 THROUGH 21
        .PSECT  SECA,ABS          ;START AN ABSOLUTE SECTION NAMED SECA
.=4                               ;ASSEMBLE CODE AT
        .WORD   .+2,HALT          ;ABSOLUTE 4 THROUGH 7,
        .PSECT  SEC1              ;RESUME THE RELOCATABLE SECTION
        INC A                     ;ASSEMBLE CODE AT
        BR ST                     ;RELOCATABLE 22 THROUGH 27
        .END
```

The first appearance of a .PSECT directive with a given name assumes the location counter is at relocatable or absolute zero. The scope of each directive extends until a directive beginning a different section is given. Further occurrences of a section name in a subsequent .PSECT statement resume assembling where the section previously ended.

```
        .PSECT  COM1,REL          ;DECLARE RELOCATABLE SECTION COM1
A:      .WORD   0                 ;ASSEMBLED AT RELOCATABLE 0,
B:      .WORD   0                 ;ASSEMBLED AT RELOCATABLE 2,
C:      .WORD   0                 ;ASSEMBLED AT RELOCATABLE 4,
        .PSECT  COM2,REL          ;DECLARE RELOCATABLE SECTION COM2
X:      .WORD   0                 ;ASSEMBLED AT RELOCATABLE 0
Y:      .WORD   0                 ;ASSEMBLED AT RELOCATABLE 2,
        .PSECT  COM1              ;RETURN TO COM1
D:      .WORD   0                 ;ASSEMBLED AT RELOCATABLE 6,
        .END
```

All labels in an absolute section are absolute; all labels in a relocatable section are relocatable. The location counter symbol, ".", is relocatable or absolute when referenced in a relocatable or absolute section, respectively. An undefined internal symbol is a global reference. It essentially has no attributes except global reference. Any labels appearing on a .PSECT (or .ASECT or .CSECT) statement are assigned the value of the location counter before the .PSECT (or other) directive takes effect. Thus, if the first statement of a program is:

```
A:      .PSECT  ALT,REL
```

then A is assigned to relocatable zero and is associated with the relocatable section ALT.

Since it is not known at assembly time where the program sections are to be loaded, all references between sections in a single assembly are translated by the Assembler to references relative to the base of that section. The Assembler provides LINK with the necessary information to resolve the linkage.

Note that this is not necessary when making a reference to an absolute section (the Assembler knows all load addresses of an absolute section).

In the following example, references to X and Y are translated into references relative to the base of the relocatable section SEN.

```
        .PSECT  ENT,ABS
.=1000
A:      CLR     X                 ;ASSEMBLED AS CLR BASE OF
                                  ;RELOCATABLE SECTION + 10
        JMP     Y                 ;ASSEMBLED AS JMP BASE OF
                                  ;RELOCATABLE SECTION + 6
        .PSECT  SEN,REL
        MOV     R0,R1
        JMP A                     ;ASSEMBLED AS JMP 1000
Y:      HALT
X:      WORD    0
        .END
```

## Code or Data Sharing

Named relocatable program sections with the attribute OVR operate as FORTRAN labeled COMMON; that is, sections of the same name with the attribute OVR from different assemblies are all loaded at the same location by LINK All other program sections (those with the attribute CON) are concatenated.

Note that there is no conflict between internal symbolic names and program section names; that is, it is legal to use the same symbolic name for both purposes. In fact, considering FORTRAN again, this is necessary to accommodate the FORTRAN statement:

```
     COMMON /X/A,B,C,X
```

where the symbol X represents the base of this program section and also the fourth element of this program section.

Program section names should not duplicate .GLOBL names. In FORTRAN language, COMMON block names and SUBROUTINE names should not be the same.

## 6.9.2 .ASECT and .CSECT Directives

DOS/BATCH assembly language programs use the .PSECT directive exclusively, as it affords all the capabilities of the .ASECT and .CSECT directives defined for other PDP-11 assemblers. The Macro Assembler will accept .ASECT and .CSECT but assembles them as if they were .PSECT's with the default attributes listed below. Also, compatibility exists between non-DOS/BATCH MACRO-11 programs and LINK, because LINK recognizes .ASECT and .CSECT directives that appear in such programs. LINK accepts these directives from non-DOS/BATCH programs, and assigns default values as shown in Table 6-3.

Table 6-3

Non-DOS/BATCH Program Section Defaults

| Attribute | Default Value | | |
|---|---|---|---|
| | .ASECT | .CSECT (named) | .CSECT |
| Name | ABS | name | Blank |
| Access | RW | RW | RW |
| Type | I | I | I |
| Scope | GBL | GBL | LCL |
| Relocation | ABS | REL | REL |
| Allocation | OVR | OVR | CON |
| Memory | LOW | LOW | LOW |

The allowable syntactical forms of .ASECT and .CSECT are:

```
            .ASECT
            .CSECT
            .CSECT   symbol
```

Note that

```
          .CSECT  JIM
```

is identical to

```
          .PSECT  JIM,GBL,OVR
```

## 6.10 SYMBOL CONTROL: .GLOBL

The Assembler produces a relocatable object module and a listing file containing the assembly listing and symbol table. LINK joins separately assembled object modules into a single load module. Object modules are relocated as a function of the specified base of the load module. The object modules (where there are more than one) are linked via global symbols, such that a global symbol in one module (either defined by direct assignment or as a label) can be referenced from another module.

A global symbol may be specified in a .GLOBL directive.

In addition, symbols referenced but not defined within a module are assumed to be global references. The .GLOBL directive is provided to reference (and provide linkage to) symbols not otherwise referenced within a module. For example, one might include a .GLOBL directive to cause linkage to a library. When defining a global definition, the .GLOBL A,B,C directive is equivalent to

```
          A==value (or A::value)
          B==value (or B::value)
          C==value (or C::value)
```

The form of the .GLOBL directive is:

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

| | | |
|---|---|---|
| where: | sym1,sym2,... | are legal symbolic names, separated by commas or spaces where more than one symbol is specified. |

Symbols appearing in a .GLOBL directive are either defined within the current program or are external symbols, in which case they are defined in another program which is to be linked with the current program by LINK prior to execution.

A .GLOBL directive line may contain a label in the label field and comments in the comment field.

At the end of assembly pass 1, MACRO-11 has determined whether a given global symbol is defined within the program or is expected to be an external symbol. All internal symbols to a given program, then, must be defined by the end of pass 1 or they will be assumed to be global references (see .ENABL, .DSABL of globals in section 6.1.6).

```
; DEFINE A SUBROUTINE WITH 2 ENTRY POINTS WHICH CALLS AN
;       EXTERNAL SUBROUTINE
        .PSECT                          ;DECLARE THE PROGRAM SECTION
        .GLOBL  A,C                     ;DEFINE A,C AS GLOBALS
A::     MOV     @(R5)+,R0               ;ENTRY A DEFINED
        MOV     #X,R1
X:      JSR     PC,C                    ;CALL EXTERNAL SUBROUTINE C
        RTS     R5                      ;EXIT
B::     MOV     +(R5)+,R1               ;DEFINE ENTRY B
        CLR     R1
        BR      X
```

In the example above, A and B are entry symbols (B is defined as global via double colon convention), C is an external symbol and X is an internal symbol.

References to external symbols can appear in the operand field of an instruction or assembler directive in the form of a direct reference, i.e.:

```
        CLR     EXT
        .WORD   EXT
        CLR     @EXT
```

or a direct reference plus or minus a constant, i.e.:

```
A=6
        CLR     EXT+A
        .WORD   EXT-2
        CLR     @EXT+A
```

An external symbol cannot be used in the evaluation of a direct assignment expression. A global symbol defined within the program can be used in the evaluation of a direct assignment statement.

## 6.11 CONDITIONAL ASSEMBLY DIRECTIVES

Conditional assembly directives provide the programmer with the capability to conditionally include or ignore blocks or source code in the assembly process. This technique is used to allow several variations of a program to be generated from the source program.

The general form of a conditional block is as follows:

```
        .IF     cond,argument(s) ;START CONDITIONAL BLOCK
                 .               ;RANGE OF CONDITIONAL
                 .
                 .               ;BLOCK
        .ENDC                    ;END CONDITIONAL BLOCK
```

| | | |
|---|---|---|
| where | cond | is a condition which must be met if the block is to be included in the assembly. These conditions are defined below. |
| | argument(s) | are a function of the condition to be tested. |

| | |
|---|---|
| range | is the body of code which is included in the assembly or ignored depending upon whether the condition is met. |

The following are the allowable conditions:

| Conditions | | | |
|---|---|---|---|
| POSITIVE | COMPLEMENT | ARGUMENTS | ASSEMBLE BLOCK IF |
| EQ | NE | expression | expression=0 (or 0) |
| GT | LE | expression | expression>: (or <0) |
| LT | GE | expression | expression<0 (or >0) |
| DF | NDF | symbolic argument | symbol is defined (or undefined) |
| B | NB | macro-type argument | argument is blank (or nonblank) |
| IDN | DIF | two macro-type arguments separated by a comma | arguments identical (or different) |
| Z | NZ | expression | same as EQ/NE |
| G | L | expression | same as GT/LE |

***NOTE***

A macro-type argument is enclosed in angle brackets or within an up-arrow construction (as described in Section 7.3.1). For example:

```
<A,B,C>
↑/124/
```

For example:

```
.IF EQ  ALPHA+1          ;ASSEMBLE IF ALPHA+1=0
          .
          .
          .
.ENDC
```

Within the conditions DF and NDF the following two operators are allowed to group symbolic arguments:

| | |
|---|---|
| & | logical AND operator |
| ! | logical inclusive OR operator |

For example:

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

assembles if both SYM1 and SYM2 are defined.

### 6.11.1 Subconditionals

Subconditionals may be placed within conditional blocks to indicate:

1. Assembly of an alternate body of code when the condition of the block indicates that the code within the block is not to be assembled.
2. Assembly of a non-contiguous body of code within the conditional block depending upon the result of the conditional test to enter the block.
3. Unconditional assembly of a body of code within a conditional block.

There are three subconditional directives, as follows:

| Subconditional Directives | Function |
|---|---|
| .IFF | The code following this statement up to the next subconditional or end of the conditional block is included in the program providing the value of the condition tested upon entering the conditional block was false. |
| .IFT | The code following this statement up to the next subconditional or end of the conditional block is included in the program providing the value of the condition tested upon entering the conditional block was true. |
| .IFTF | The code following this statement up to the next subconditional or the end of the conditional block is included in the program regardless of the value of the condition tested upon entering the conditional block. |

The implied argument of the subconditionals is the value of the condition upon entering the conditional block. Subconditionals are used within outer level conditional blocks. Subconditionals are ignored within nested, unsatisfied conditional blocks.

For example:

```
        .IF DF   SYM              ;ASSEMBLE BLOCK IF SYM IS DEFINED
        .IFF
          .                       ;ASSEMBLE THE FOLLOWING CODE ONLY IF
          .                       ;SYM IS UNDEFINED.
          .
        .IFT                      ;ASSEMBLE THE FOLLOWING CODE ONLY IF
          .                       ;SYM IS DEFINED.
          .
          .
        .IFTF                     ;ASSEMBLE THE FOLLOWING CODE
          .                       ;UNCONDITIONALLY.
          .
          .
        .ENDC


        .IF DF   X                ;ASSEMBLY TESTS FALSE
        .IF DF   Y                ;TESTS FALSE
        .IFF                      ;NESTED CONDITIONAL
          .                       ;IGNORED
          .
          .
        .IFT                      ;NOT SEEN
          .
          .
          .
        .ENDC
        .ENDC
```

However,

```
        .IF DF   X                ;TESTS TRUE
        .IF DF   Y                ;TESTS FALSE
        .IFF                      ;IS ASSEMBLED
          .
          .
          .
        .IFT                      ;NOT ASSEMBLED
          .
          .
          .
        .ENDC
        .ENDC
```

## 6.11.2 Immediate Conditionals

An immediate conditional directive is a means of writing a 1-line conditional block. In this form, no .ENDC statement is required and the condition is completely expressed on the line containing the conditional directive. Immediate conditions are of the form:

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

| | | |
|---|---|---|
| where: | cond | is one of the legal conditions defined for conditional blocks in section 6.11. |
| | arg | is the argument associated with the conditional specified, that is, either an expression, symbol, or macro-type argument, as described in section 6.11. |
| | statement | is the statement to be assembled if the condition is met. |

For example:

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

this statement generates the code

```
        BEQ     ALPHA
```

if the symbol FOO is defined.

A label must not be placed in the label field of the .IIF statement. Any necessary labels may be placed on the previous line:

```
LABEL:
        .IIF    DF F00,BEQ ALPHA

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

## 6.11.3 PAL-11R Conditional Assembly Directives

In order to maintain compatibility with programs developed under PAL-11R, the following conditionals remain permissible under MACRO-11. It is advisable that future programs be developed using the format for MACRO-11 conditional assembly directives.

| Directive | Arguments | Assemble Block if |
|---|---|---|
| .IFZ or .IFEQ | expression | expression=0 |
| .IFNZ or.IFNE | expression | expression not equal 0 |
| .IFL or .IFLT | expression | expression< 0 |
| .IFG or .IFGT | expression | expression> 0 |
| .IFLE | expression | expression is < or =0 |
| .IFDF | logical expression | expression is true (defined) |
| .IFNDF | logical expression | expression is false (undefined) |

The rules governing the usage of these directives are now the same as for the MACRO-11 conditional assembly directives previously described. Conditional assembly blocks must end with the .ENDC directive and are limited to a nesting depth of 16(10) levels (instead of the 127(10) levels allowed under PAL-11R).

# CHAPTER 7

# MACRO DIRECTIVES

## 7.1 MACRO DEFINITION

It is often convenient in assembly language programming to generate a recurring coding sequence with a single statement. In order to do this, the desired coding sequence is first defined with dummy arguments as a macro. Once a macro has been defined, a single statement calling the macro by name with a list of real arguments (replacing the corresponding dummy arguments in the definition) generates the correct sequence or expansion.

### 7.1.1 .MACRO

The first statement of a macro definition must be a .MACRO directive. The .MACRO directive is of the form:

```
        .MACRO name, dummy argument list
```

where:

| | |
|---|---|
| name | is the name of the macro. This name is any legal symbol. The name chosen may be used as a label elsewhere in the program. |
| ' | represents any legal separator (generally a comma or space). |
| dummy argument list | zero, one, or more legal symbols which may appear anywhere in the body of the macro definition, even as a label. These symbols can be used elsewhere in the user program with no conflicts of definition. Where more than one dummy argument is used, they are separated by any legal separator (generally a comma). |

A comment may follow the dummy argument list in a statement containing a .MACRO directive. For example:

```
        .MACRO ABS,A,B        ;DEFINE MACRO ABS WITH TWO ARGUMENTS
```

A label must not appear on a .MACRO statement. Labels are sometimes used on macro calls, but serve no function when attached to .MACRO statements.

### 7.1.2 .ENDM

The final statement of every macro definition must be an .ENDM directive of the form:

```
        .ENDM name
```

where:

| | |
|---|---|
| name | is an optional argument, being the name of the macro terminated by the statement. |

For example:

```
        .ENDM     (terminates the current macro definition)

        .ENDM ABS (terminates the definition of the macro ABS)
```

If specified, the symbolic name in the .ENDM statement must correspond to that in the matching .MACRO statement. Otherwise the statement is flagged and processing continues. Specification of the macro name in the .ENDM statement permits the Assembler to detect missing .ENDM statements or improperly nested macro definitions.

The .ENDM statement may contain a comment field, but must not contain a label.

An example of a macro definition is shown below:

```
       .MACRO  TYPMSG MESSGE      ;TYPE A MESSAGE
       JSR     R5,TYPMSG
       .WORD   MESSGE
       .ENDM
```

## 7.1.3 .MEXIT

In order to implement alternate exit points from a macro (particularly nested macros), the .MEXIT directive is provided. .MEXIT terminates the current macro as though an .ENDM directive were encountered. Use of .MEXIT bypasses the complications of conditional nesting and alternate paths. For example:

```
       .MACRO  ALTR  N,A,B
         .
         .
         .
       .IF EQ,N                   ;START CONDITIONAL BLOCK
         .
         .
         .
       .MEXIT                     ;EXIT FROM MACRO DURING CONDITIONAL
                                  ;BLOCK
       .ENDC                      ;END CONDITIONAL BLOCK
         .
         .
         .
       .ENDM                      ;NORMAL END OF MACRO
```

In an assembly where N=0, the .MEXIT directive terminates the macro expansion.

```
        .MACRO  IDT  SYM           ;ASSUME THAT THE SYMBOL ID TAKES
        .IDENT  /SYM/              ;ON A UNIQUE 2-DIGIT VALUE FOR
        .ENDM                      ;EACH POSSIBLE CONDITIONAL ASSEMBLY
        .MACRO  OUT ARG            ;OF THE PROGRAM
        IDT     005A'ARG               .
        .ENDM                          .
           .                           .
           .
           .                       ;WHERE 005A IS THE UPDATE
        OUT     \ID                ;VERSION OF THE PROGRAM
                                   ;AND ARG INDICATES THE
                                   ;CONDITIONAL ASSEMBLY VERSION.
```

The above macro call expands to

```
        .IDENT  /005AXX/
```

where XX is the conditional value of ID.

Two macros are necessary since the text delimiting characters in the .IDENT statement would inhibit the concatenation of a dummy argument.

## 7.3.4 Number of Arguments

If more arguments appear in the macro call than in the macro definition, the excess arguments are ignored. If fewer arguments appear in the macro call than in the definition, missing arguments are assumed to be null (consist of no characters). The conditional directives .IF B and .IF NB can be used within the macro to detect unnecessary arguments.

A macro can be defined with no arguments.

## 7.3.5 Automatically Created Symbols

MACRO-11 can create symbols of the form n$ where n is a decimal integer number such that $64<n<127$. Created symbols are always local symbols between 64$ and 127$. (For a description of local symbols, see Section 3.5.) Such local symbols are created by the Assembler in numerical order, i.e.:

```
          64$
          65$
           .
           .
           .
          126$
          127$
```

Created symbols are particularly useful where a label is required in the expanded macro. Such a label must otherwise be explicitly stated as an argument with each macro call or the same label is generated with each expansion (resulting in a multiply-defined label). Unless a label is referenced from outside the macro, there is no reason for the programmer to be concerned with that label.

The range of these local symbols extends between two explicit labels. Each new explicit label causes a new local symbol block to be initialized.

The macro processor creates a local symbol on each call of a macro whose definition contains a dummy argument preceded by the ? (question mark) character. For example:

```
        .MACRO   ALPHA, 3A,?B
        TST      A
        BEQ      B
        ADD      #5,A
B:
        .ENDM
```

Local symbols are generated only where the real argument of the macro call is either null or missing. If a real argument is specified in the macro call, the generation of a local symbol is inhibited and normal replacement is performed. Consider the following expansions of the macro ALPHA above.

Generate a local symbol for missing argument:

```
        ALPHA    %1
        TST      %1
        BEQ      64$
        ADD      #5,%1
64$:
```

do not generate a local symbol:

```
        ALPHA    %2,XYZ
        TST      %2
        BEQ      XYZ
        ADD      #5,%2
XYZ:
```

These Assembler-generated symbols are restricted to the first 16 (decimal) arguments of a macro definition.

## 7.3.6 Concatenation

The apostrophe or single quote character (') operates as a legal separating character in macro definitions. An ' character which 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:

```
        .MACRO   DEF A,B,C
A'B:    .ASCIZ   /C/
        .WORD    ''A'''B
        .ENDM
```

When this macro is called:

```
DEF     X,Y,<MACRO-11>
```

it expands as follows:

```
XY:     .ASCIZ  /MACRO-11/
        .WORD   'X'Y
```

In the macro definition, the scan terminates upon finding the first ' character. Since A is a dummy argument, the ' is removed. The scan resumes with B, notes B as another dummy argument and concatenates the two dummy arguments. The third dummy argument is noted as going into the operand of the .ASCIZ directive. On the next line (this is not a useful example, but one for purely illustrative purposes) the argument to .WORD is seen as follows: The scan begins with a ' character. Since it is neither preceded nor followed by a dummy argument, the ' character remains in the macro definition. The scan then encounters the second ' character which is followed by a dummy argument and is discarded. The scan of the argument A terminated upon encountering the second ' which is also discarded since it follows a dummy argument. The next ' character is neither preceded nor followed by a dummy argument and remains in the macro expansion. The last ' character is followed by another dummy argument and is discarded. (Note that the five ' characters were necessary to generate two ' characters in the macro expansion.)

Within nested macro definitions, multiple single quotes can be used, with one quote removed at each level of macro nesting.

## 7.4 .NARG, .NCHR, AND .NTYPE

These three directives allow the user to obtain the number of arguments in a macro call (.NARG), the number of characters in an argument (.NCHR), or the addressing mode of an argument (.NTYPE). Use of these directives permits selective modifications of a macro depending upon the nature of the arguments passed.

The .NARG directive enables the macro being expanded to determine the number of arguments supplied in the macro call, and is of the form:

```
label:  .NARG   symbol
```

| | | |
|---|---|---|
| where: | label | is an optional statement label |
| | symbol | is any legal symbol whose value is equated to the number of arguments in the macro call currently being expanded. The symbol can be used by itself or in expressions. |

The .NARG directive can occur only within a macro definition.

The .NCHR directive enables a program to determine the number of characters in a character string, and is of the form:

```
label:  .NCHR   symbol, <character string>
```

| | | |
|---|---|---|
| where: | label | is an optional statement label |
| | symbol | is any legal symbol which is equated to the number of characters in the specified character string. |

The symbol is separated from the character string argument by any legal separator.

| | |
|---|---|
| <character string> | is a string of printing characters which should only be enclosed in angle brackets if it contains a legal separator. A semicolon also terminates the character string. |

The .NCHR directive can occur anywhere in a MACRO-11 program.

The .NTYPE directive enables the macro being expanded to determine the addressing mode of any argument, and is of the form:

label: .NTYPE symbol, arg

| | | |
|---|---|---|
| where: | label | is an optional statement label |
| | symbol | is any legal symbol, the value of which is equated to the 6-bit addressing mode of the argument. The symbol is separated from the argument by a legal separator. This symbol can be used by itself or in expressions. |
| | arg | is any legal macro argument (dummy argument) as defined in section 7.3. |

The .NTYPE directive can occur only within a macro definition. An example of .NTYPE usage in a macro definition is shown below:

```
        .MACRO  SAVE ARG
        .NTYPE  SYM,ARG
        .IF     EQ,SYM&70
        MOV     ARG,TEMP            ;REGISTER MODE
        .IFF
        MOV     #ARG,TEMP           ;NON-REGISTER MODE
        .ENDC
        .ENDM
```

## 7.5 .ERROR and .PRINT

The .ERROR directive is used to output messages to the command output device during assembly pass 2. A common use is to provide diagnostic announcements of a rejected or erroneous macro call. The form of the .ERROR directive is as follows:

label: .ERROR expr;text

| | | |
|---|---|---|
| where | label | is an optional statement label |
| | expr | is an optional legal expression whose value is output to the command device when the .ERROR directive is encountered. Where expr is not specified, the text only is output to the command device. |
| | ; | denotes the beginning of the text string to be output. |

# APPENDIX A

# MACRO-11 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, duplicated by 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. duplicated by 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/B (DC0). |
| 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 (TAPE, 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 | 044 | $ | |
| 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 | 062 | 2 | |
| 0 | 063 | 3 | |
| 1 | 064 | 4 | |
| 0 | 065 | 5 | |
| 0 | 066 | 6 | |
| 1 | 067 | 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 | |
| 1 | 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 ^ on some machines.

---------------

** ← appears as _ (underscore) 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, thus,

```
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 is (arithmetic is performed in octal):

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

## E.2 COMMENTS

Comment all coding to convey the global role of an instruction and not simply a literal translation of the instruction into English. In general this will consist of a comment per line of code. If a particularly difficult, obscure, or elegant instruction sequence is used, a paragraph of comments shall immediately precede that section of code.

Preface text describing formats, algorithms, program-local variables, etc. will be delimited by the character sequence ;+ at the start of the text and ;- at the end. The comment will start in column 3.

For example:

```
;+

; The invert routine accepts

; a list of random numbers and

; applies the Kolmogorov algorithm

; to alphabetize them.

;-
```

## E.3 NAMING STANDARDS

### E.3.1 Register Standards

#### E.3.1.1 General Purpose Registers

Only the following names are permitted as register names; and may not be used for any other purpose:

```
R0=%0                               ;REG 0
R1=%1                               ;REG 1
R2=%2                               ;REG 2
R3=%3                               ;REG 3
R4=%4                               ;REG 4
R5=%5                               ;REG 5
SP=%6                               ;STACK POINTER (REG 6)
PC=%7                               ;PROGRAM COUNTER (REG 7)
```

#### E.3.1.2 Hardware Registers

These registers must be named identically with the hardware definition. For example, PS and SWR.

#### E.3.1.3 Device Registers

These are symbolically named identically to the hardware notation. For example, the control status register for the RK disk is RKCS. Only this symbolic names may be used to refer to this register.

### E.3.2 Processor Priority

Testing or altering the processor priority is done using the symbols

PR0, PR1, PR2, ......PR7

which are equated to their corresponding priority bit pattern.

Use of SPL is permitted only by showing cause and then its generation occurs via a macro call.

### E.3.3 Other Symbols

Frequently-used bit patterns such as CR and LF will be made conventional symbolics on an as-needed basis.

### E.3.4 Using the Standard Symbolics

The register standards will be defined within the assembler. All other standard symbols will appear in a file and will be linked prior to program execution.

### E.3.5 Labels

#### E.3.5.1 Global Labels

Global labels should be easily recognized by their format. The following standards apply and completely define symbol standards for DOS/BATCH.

```
<letter>              ::=A/B/C/.../Y/Z
<digit>               ::=0/1/.../8/9
<alpha-num>           ::=<letter>/<digit>
<doll-or-dot>         ::=$/.
<char>                ::=<alpha-num>/<doll-or-dot>
<number>              ::=[1-5]<digit>*
<non-glbl-sym>        ::=<letter>[0-5]<char>
<glbl-lbl>            ::=<doll-or-dot>[0-5]<char>
<glbl-offset>         ::=<letter><doll-or-dot>[1-4]<char>
<glbl-bit-ptrn>       ::=<letter><alpha-num><doll-or-dot>[1-3]<char>
<local-sym>           ::=<number>$**
```

where

| | |
|---|---|
| non-glbl-sym | are non-global symbols. |
| glbl-lbl | are global labels (addresses). |
| glbl-offset | are global offsets (absolute quantities). |
| glbl-bit-ptrn | are global bit patterns. |

*The notation [n-m] indicates the number of repetitions permitted for the immediately following non-terminal.

**number is in the range 0<number<65535.

#### E.3.5.2 Program-local Labels

Self-relative address arithmetic (.+n) is absolutely forbidden in branch instructions, and should be used only where absolutely essential elsewhere. Indeed no implication of adjacency is permitted without showing cause. Non-symbolic absolute references are also forbidden.

Target labels for branches that exist solely for positional reference will use local labels of the form

<num>$:

Use of non-local labels is restricted, within reason, to those cases where reference to the code occurs external to the code. Local-labeling is formatted such that the numbers proceed sequentially down the page and from page to page.

## E.4 PROGRAM MODULES

### E.4.1 General Comments on Programs

In DOS/BATCH, a program provides a single distinct function. No limits exist on size, but the single function limitation should make modules larger than 1K a rarity. Since DOS/BATCH may eventually exploit the virtual memory capacity of the 11/40 and 11/45, programs should make every attempt to maintain a dense reference locus (don't promiscuously branch over page boundaries or over a large absolute address distance).

All code is read-only. Code and data areas are distinct and each contains explanatory text. Read-only data should be segregated from read-write data.

### E.4.2 The Module Preface

Program modules adhere to a strict format. This format adds to the readability and understandability of the module. The following sections are included in each module:

For the Code Section:

1. A .TITLE statement that specifies the name of the module.

2. A .PSECT statement that defines the program section in which the module resides. If a module contains more than one routine, subtitles may be used.

3. A copyright statement, and the disclaimer.

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

4. The version number of the file.
   Note: Items 1-5 must appear on the same page. The PDP-11 version number standard is described in Section 9.0.

5. The name of the principal author and the date on which the module was first created.

6. The name of each modifying author and the date of modification, name and modification dates appear one per line and in chronological order.

7. A brief statement of the function of the module.

8. A list of the definitions of all equated local symbols used in the module. These definitions appear one per line and in alphabetical order.

9. All local Macro definitions, preferably in alphabetical order by name.

10. All local data. The data should indicate

    a. Description of each element (type, size, etc.)
    b. Organization (functional, alpha, adjacent, etc.)
    c. Adjacency requirements

11. A list of the inputs expected by the module. This includes the calling sequence, condition code settings, and global data settings.

12. A list of the outputs produced as a result of entering this module. These include delivered results, condition code settings, but not side effects. (All these outputs are visible to the caller.)

13. A list of all effects (including side effects) produced as a result of entering this module. Effects include alterations in the state of the system not explicitly expected in the calling sequence, or those not visible to the caller.

14. A more detailed definition of the function of the module.

15. The module code.

## E.4.3 Formatting the Module Preface

Rules

1. The first five items appear on the same page and will not have explicit headings.

2. Titles start at the left margin*; descriptive text is indented 1 tab position.

3. Items 7-14 will have headings which start at the left margin, preceded and followed by blank lines. Items which do not

---------------

*The left margin consists of a ; a space then the heading, so the text of the heading begins in column 3.

apply may be omitted.

A template for the module preface follows.

Template.

```
FILE-EXAMPL.S01
        .TITLE
        .PSECT KERNEL
;
; COPYRIGHT 1972, DIGITAL ...
;
; VERSION V001A
;
; JOE PASCUSNIK 1/1/72
;
;
; MODIFICATIONS
;       RICHARD DOE
;
;       FIX SPR 3477 1/21/72
;
;       ADD PAGE CHANGE LOGIC 1/22/72
;
; MODULE FUNCTION
;          :
;          :
; EQUATED SYMBOLS
;          :
;          :
; LOCAL MACROS
;          :
;          :
; LOCAL DATA
;          :
;          :
; INPUTS
;          :
;          :
; OUTPUTS
;          :
; EFFECTS
;          :
; MODULE FUNCTION-DETAILS
;          :
; MODULE CODE
;          :
```

## E.4.3 Modularity

### E.4.3.1 Introduction

No other characteristic has more impact on the ultimate engineering success of a system than does modularity. Modularity for DOS/BATCH consists of the application of the uni-function philosophy described in section 4.1 and a set of calling and return conventions universally adhered to.

### E.4.3.2 Calling Conventions (Inter-Module)

Transfer of Control

Macros will exist for call and return. The actual transfer will be via a JSR PC instruction. For the register save routine, a JSR Rn,SAVE will be permitted.

Register Conventions

On entry, except for result registers, a subroutine, mimimally, saves all registers it intends to alter, and on exit it restores these registers. (State preservation is assumed across calls.)

Argument Passing

Any registers may be used, but their use should follow a coherent pattern. For example, if passing three arguments pass them in R0, R1 and R2 rather than R0, R2, R5. Saving and restoring occurs in one place.

### E.4.3.3 Exiting

All subroutine exits occur through a single RTS PC.

### E.4.3.4 Intra Module Calling Conventions

Designer optional, but consistency favors a calling sequence identical to that of the inter module sequence.

### E.4.3.5 Success/Failure Indication

The C bit will be used to return success/failure indicator, where success equals 0, and failure equals 1. The volatile registers can be used to return values or additional success/failure data.

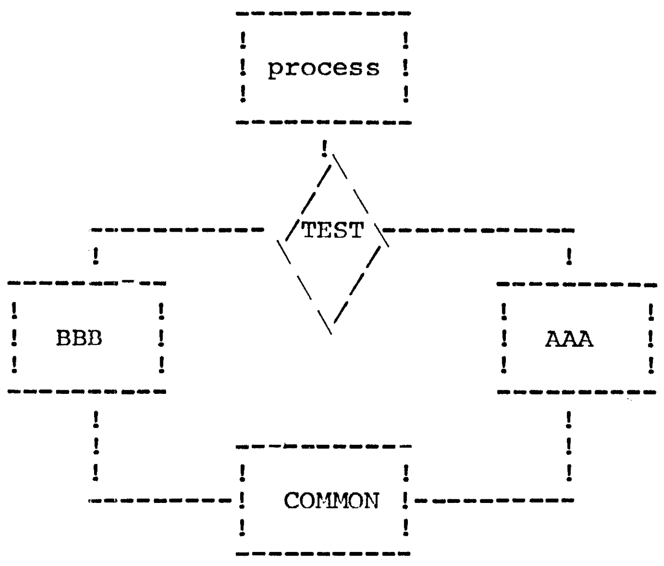
### E.4.3.6 Module Checking Routines

Modules have the responsiblity of verifying the validity of arguments passed to them. The design of a module's calling sequence should aim at minimizing the validity checks by minimizing invalid combinations. Programmers can add test code to perform additional checks during shakedown. All code should aim at discovering an error as close (in terms of instruction executions) to its occurrence as possible.

## E.5.0 FORMATTING STANDARDS

### E.5.1 Program Flow

Programs will be organized on the listing such that they flow down the page, even at the cost of an extra branch or jump.



shall appear on the listing as:

```
        TST
        BNE BBB
AAA:.......
        .......
        .......
        .......
        B   CMN

BBB:.......
        .......
        .......
CMN:.......
        .......
        .......
```

# APPENDIX F

# WRITING POSITION-INDEPENDENT CODE - A TUTORIAL

It is possible to write a source program that can be loaded and run in any section of virtual memory. Such a program is said to consist of position-independent code. The construction of position independent code is dependent upon the proper usage of PDP-11 addressing modes. (Addressing modes are described in detail in Chapter 5. The remainder of this Appendix assumes the reader is familiar with the various addressing modes.)

All addressing modes involving only register references are position-independent. These modes are as follows:

| | |
|---|---|
| R | register mode |
| @R | deferred register mode |
| (R)+ | autoincrement mode |
| @(R)+ | deferred autoincrement mode |
| -(R) | autodecrement mode |
| @-(R) | deferred autodecrement mode |

When using these addressing modes, position-independence is guaranteed providing the contents of the registers have been supplied such that they are not dependent upon a particular core location.

The relative addressing modes are generally position independent. These modes are as follows:

| | |
|---|---|
| A | relative mode |
| @A | relative deferred mode |

Relative modes are not position-independent when A is an absolute address (that is, a non-relocatable address) which is referenced from a relocatable module.

Index modes can be either position-independent or nonposition-independent, according to their use in the program. These modes are:

| | |
|---|---|
| X(R) | index mode |
| @X(R) | index deferred mode |

If the base, X, is position-independent, the reference is also position-independent. For example:

```
        MOV     2(SP),R0          ;POSITION-INDEPENDENT
        N=4
        MOV     N(SP),R0          ;POSITION-INDEPENDENT
        CLR     ADDR(R1)          ;NONPOSITION-INDEPENDENT
```

Caution must be exercised in the use of index modes in position independent code.

Immediate mode can also be either position-independent or not, according to its usage. Immediate mode references are formatted as follows:

```
          #N          immediate mode
```

Where an absolute number or a symbol defined by an absolute direct assignment replaces N, the code is position independent. Where a label replaces N, the code is nonposition-independent. (That is, immediate mode references are position-independent only where N is an absolute value.)

Absolute mode addressing is unlikely to be position-independent and should be avoided when coding position-independently. Absolute mode addressing references are formatted as follows:

```
          @#A         absolute mode
```

Since this mode is used to obtain the contents of a specific core address, it violates the intentions of position-independent code.

Such a reference is position-independent if A is an absolute address.

Position-independent code is used in writing programs such as device drivers and utility routines which are most useful when they can be brought into any available core space. Figure F-1 and Figure F-2 show pieces of device driver code; one of which is position-independent and one of which is not.

```
; DVRINT -- ADDRESS OF DEVICE DRIVER INTERRUPT SERVICE
; VECTOR -- ABSOLUTE ADDRESS OF DEVICE INTERRUPT VECTOR
; DRIVER -- START ADDRESS OF DEVICE DRIVER

        MOV     #DVRINT,VECTOR  ;SET INTERRUPT ADDRESS
MOVB            DRIVER+6,VECTOR+2 ;SET PRIORITY
        CLRB    VECTOR+3        ;CLEAR UPPER STATUS BYTE
```

Figure F-1 Non-Position Independent Code

```
        MOV     PC,R1           ;GET DRIVER START
        ADD     #DRIVER-.,R1
        MOV     #VECTOR,R2      ;...& VECTOR ADDRESSES
        CLR     @R2             ;SET INTERRUPT ADDRESS
        MOVB    5(R1),@R2       ;...AS START ADDRESS+OFFSET
        ADD     R1,(R2)+
        CLR     @R2             ;SET PRIORITY
        MOVB    6(R1),@R2
```

Figure F-2 Position Independent Code

In both examples it is assumed that the program calling the device driver has correctly initialized its interrupt vector (VECTOR) within absolute memory locations 0-377. The interrupt entry point offset is in byte DRIVER+5. (The contents of the Driver Table shows at DRIVER+5: .BYTE DVRINT,DRIVER.) The priority level is at byte DRIVER+6.

In the first example, the interrupt address is directly inserted into the absolute address of VECTOR. Neither of these addressing modes is position-independent.

The instruction to initialize the driver priority level uses an offset from the beginning of the driver code to the priority value and places that value into the absolute address VECTOR+2 (which is not position-independent). The final operation clearing the absolute address VECTOR+3 is also not position-independent.

In the position-independent code, operations are performed in registers wherever possible. The process of initializing registers is carefully planned to be position-independent. For example: the first two instructions obtain the starting address of the driver. The current PC value is loaded into R1, and the offset from the start of the driver to the current location is added to that value. Each of these operations is position-independent. The immediate mode value of VECTOR is loaded into R2; which places the absolute address of the transfer vector into a register for later use. The transfer vector is then cleared, and the offset for the driver starting address is loaded into the vector. The starting address of the driver is then added into the vector, giving the desired entry point to the driver. (This is equivalent to the first statement in Figure F-1.) Since R2 has been updated to point to VECTOR+2, that location is then cleared and the priority level inserted into the appropriate byte.

The position-independent code demonstrates a principle of PDP-11 coding practice, which was discussed earlier; that is, the programmer is advised to work primarily with register addressing modes wherever possible, relying on the setup mechanism to determine position-independence.

The MACRO-11 Assembler provides the user with a way of checking the position-independence of the code. In an assembly listing, MACRO-11 inserts a ' character following the contents of any word which requires the Task Builder to perform a operation. In some cases this character indicates a nonposition-independent instruction, in other cases, it merely draws the user's attention to the use of a symbol which may or may not be position-independent. The cases which cause a ' character in the assembly listing are as follows:

1. Absolute mode symbolic references are flagged with an ' character when the reference is not position-independent. References are not flagged when they are position-independent (i.e., absolute). For example:

   ```
   MOV  @#ADDR,R1       ;PIC ONLY IF ADDR IS ABSOLUTE.
   ```

2. Index mode and index deferred mode references are flagged with an ' character when the base is a symbolic label address (relocatable rather than an absolute value). For example:

   ```
   MOV  ADDR(R1),R5     ;NON-PIC IF ADDR IS RELOCATABLE.
   MOV  @ADDR(R1),R5    ;NON-PIC IF ADDR IS RELOCATABLE.
   ```

3. Relative mode and relative deferred mode are flagged with an ' character when the address specified is a global symbol. For example:

   ```
   MOV  GLB1,R1         ;PIC WHEN GLB1 IS A GLOBAL SYMBOL.
   MOV  @GLB1,R1        ;PIC WHEN GLB1 IS A GLOBAL SYMBOL.
   ```

If the symbol is absolute, the reference is flagged and is not position-independent.

4. Immediate mode references to symbolic labels are always flagged with an ' character.

```
MOV   #3,R0            ;ALWAYS POSITION-INDEPENDENT.
MOV   #ADDR,R1         ;NON-PIC WHEN ADDR IS RELOCATABLE.
```

Examples of assembly listings contining the ' character are shown below:

```
1  011744         ENDP2:                                  ;END OF PASS 2
2                          .IF NDF XCREF
3  011744 016702           MOV     CREFPNT,R2             ;ANY CREF IN PROGRESS?
          000142'
4  011750 001402           BEQ     8$                     ;  NO
5  011752                  CALL    CRFDMP                 ;YES, DUMP AND CLOSE BUFFER
6  011756         8$:
7                          .ENDC
8  011756 005767           TST     BLKTYP                 ;ANY OBJECT OUTPUT?
          000542'
9  011762 001423           BEQ     1$                     ;  NO
10  11764                  CALL    OBJDMP                 ;YES, DUMP IT
11  11770 012767           MOV     #BLKT06,BLKTYP         ;SET END
          000006
          000542'
12  11776                  CALL    RLDDMP                 ;DUMP IT
13                         .IF NDF XEDABS
14  12002 032767           BIT     #ED.ABS,EDMASK         ;ABS OUTPUT?
          000002
          000124'
15  12010 001010           BNE     1$                     ;  NO
16  12012 016700           MOV     OBJPNT,R0
          000536'
17  12016 016720           MOV     ENDVEC+6,(R0)+         ;SET END VECTOR
          000244'
18  12022 010067           MOV     R0,OBJPNT
          000536'
19  12026                  CALL    OBJDMP
20                         .ENDC
21  12032 105767 1$:       TSTB    LLTBL+2                ;ANY LISTING OUTPUT?
          000546'
22  12036 001474           BEQ     15$                    ;  NO
23  12040 032767           BIT     #LC.SYM,LCMASK         ;SYMBOL TABLE SUPPRESSION?
          040000
          000110'
```