January 1980

This document describes how to use the MACRO-11 relocatable assembler to develop PDP-11 assembly language programs. Although no prior knowledge of MACRO-11 is required, the user should be familiar with the PDP-11 processor addressing modes and instruction set. This manual presents detailed descriptions of MACRO-11's features, including source and command string control of assembly and listing functions, directives for conditional assembly and program sectioning, and user-defined and system macro libraries. The chapters on operating procedures previously were found in two separate manuals (the PDP-11 MACRO-11 Language Reference Manual and the IAS/RSX MACRO-11 Reference Manual). This manual should be used in conjunction with a system-specific user's guide as well as a Linker or a Task Builder manual.

PDP-11 MACRO-11

Language Reference Manual

Order No. AA-5075B-TC

SUPERSESSION/UPDATE INFORMATION: This manual supersedes previous editions, Order Numbers AA-5075A-TC, published 1977, and DEC-11-OIMRA-A-B-D, published 1976.

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

digital equipment corporation · maynard. massachusetts

First Printing, August 1977
Revised: January 1980

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

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

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

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

The following are trademarks of Digital Equipment Corporation:

DIGITAL	DECsystem-10	MASSBUS
DEC	DECtape	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EDUSYSTEM	PHA
UNIBUS	FLIP CHIP	RSTS
COMPUTER LABS	FOCAL	RSX
COMTEX	INDAC	TYPESET-8
DDT	LAB-8	TYPESET-11
DECCOMM	DECSYSTEM-20	TMS-11
ASSIST-11	RTS-8	ITPS-10
VAX	VMS	SBI
DECnet	IAS	PDT
DATATRIEVE	TRAX	

CONTENTS

FIGURES

TABLES

TO THE READER

0.1 MANUAL OBJECTIVES AND READER ASSUMPTIONS

This manual is intended to enable users to develop programs coded in the MACRO-11 V4.0 assembly language.

No prior knowledge of the MACRO-11 Relocatable Assembler is assumed, but the reader should be familiar with the PDP-11 processors and related terminology, as presented in the PDP-11 Processor Handbooks. The reader is also encouraged to become familiar with the linking process, as presented in the applicable system manual (see Section 0.3), because linking is necessary for the development of executable programs.

If a terminal is available to the reader, he/she is advised to try some of the examples in the manual or to write a few simple programs that illustrate the concepts covered. Even experienced programmers find that working with a simple program helps them to understand a confusing feature of a new language.

The examples in this manual were done on an RT-11 system. MACRO-11 V4.0 may also be used on IAS/RSX-11M, RSX-11M-PLUS and RSTS systems (see Part IV for information about operating procedures).

It can be assumed that all references to RSX-11M also apply to RSX-11M-PLUS with the exception of those in Chapter 8, which deals with each system individually.

0.2 STRUCTURE OF THE DOCUMENT

This manual has four parts and eight appendices.

Part I introduces MACRO-11.

Chapter 1 lists the key features of MACRO-11.

Chapter 2 identifies the advantages of following programming standards and conventions and describes the format used in coding MACRO-11 source programs.

Part II presents general information essential to programming with the MACRO-11 assembly language.

Chapter 3 lists the character set and describes the symbols, terms, and expressions that form the elements of MACRO-11 instructions.

Chapter 4 describes the output of MACRO-11 and presents concepts essential to the proper relocation and linking of object modules.

Chapter 5 describes how data stored in memory can be accessed and manipulated using the addressing modes recognized by the PDP-11 hardware.

Part III describes the MACRO-11 directives that control the processing of source statements during assembly.

Chapter 6 discusses directives used for generalized MACRO-11 functions.

Chapter 7 discusses directives used in the definition and expansion of macros.

Part IV presents the operating procedures essential to the assembly, linking, and initiating of MACRO-11 programs.

Chapter 8 covers the IAS, RSX-11M and RSX-11M-PLUS systems.

Chapter 9 covers the RSTS/RT-11 systems.

Appendix A lists the ASCII and Radix-50 character sets used in MACRO-11 programs.

Appendix B lists the special characters recognized by MACRO-11, summarizes the syntax of the various addressing modes used in PDP-11 processors, and briefly describes the MACRO-11 directives in alphabetical order.

Appendix C lists alphabetically the permanent symbols that have been defined for use with MACRO-11.

Appendix D lists alphabetically the error codes produced by MACRO-11 to identify various types of errors detected during the assembly process.

Appendix E contains a coding standard that is recommended practice in preparing MACRO-11 programs.

Appendix F discusses several methods of conserving dynamic memory space for users of small systems who may experience difficulty in assembling MACRO-11 programs.

Appendix G is a discussion of position-independent code (PIC).

Appendix H contains an assembly and cross-reference listing.

0.3 ASSOCIATED DOCUMENTS

For descriptions of documents associated with this manual, refer to the applicable documentation directory listed below:

IAS Documentation Directory

RSX-11M-PLUS Documentation Directory

RSX-11M/RSX-11S Documentation Directory

RT-11 Documentation Directory

0.4 DOCUMENT CONVENTIONS

The portions of text that are shaded in red pertain to features that are not available in the 8K version of MACRO-11.

The portions of text that are shaded in gray pertain to features available only to users of the following operating systems (or subsequent versions):

RT-11 Version 4.0

RSTS Version 7.0

The color red is used in command string examples to indicate user type-in.

The symbols defined below are used throughout this manual.

Symbol	Definition
[]	Brackets indicate that the enclosed argument is optional.
...	Ellipsis indicates optional continuation of an argument list in the form of the last specified argument.
UPPER-CASE CHARACTERS	Upper-case characters indicate elements of the language that must be used exactly as shown.
lower-case characters	Lower-case characters indicate elements of the language that are supplied by the programmer.
(n)	In some instances the symbol (n) is used following a number to indicate the radix. For example, 100(8) indicates that 100 is an octal value, while 100(10) indicates a decimal value.

PART I

MACRO-11: ASSEMBLY AND FORMATTING

CHAPTER 1

THE MACRO-11 ASSEMBLER

MACRO-11 provides the following features:

1. Source 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; optional cross-reference listing of symbols
5. Relocatable object modules
6. Global symbols for linking object modules
7. Conditional assembly directives
8. Program sectioning directives
9. User-defined macros and macro libraries
10. Comprehensive system macro library
11. Extensive source and command string control of listing functions.

MACRO-11 assembles one or more ASCII source files containing MACRO-11 statements into a single relocatable binary object file. The output of MACRO-11 consists of a binary object file and a file containing the table of contents, the assembly listing, and the symbol table. An optional cross-reference listing of symbols and macros is available. A sample assembly listing is provided in Appendix H.

1.1 ASSEMBLY PASS 1

During pass 1, MACRO-11 locates and reads all required macros from libraries, builds symbol tables and program section tables for the program, and performs a rudimentary assembly of each source statement.

In the first step of assembly pass 1, MACRO-11 initializes all the impure data areas (areas containing both code and data) that will be used internally for the assembly process. These areas include all dynamic storage and buffer areas used as file storage regions.

MACRO-11 then calls a system subroutine which transfers a command line into memory. This command line contains the specifications of all files to be used during assembly. After scanning the command line for proper syntax, MACRO-11 initializes the specified output files. These files are opened to determine if valid output file specifications have been passed in the command line.

MACRO-11 now initiates a routine which retrieves source lines from the input file. If no input file is open, as is the case at the beginning of assembly, MACRO-11 opens the next input file specified in the command line and starts assembling the source statements. MACRO-11 first determines the length of the instructions, then assembles them according to length as one word, two words, or three words.

At the end of assembly pass 1, MACRO-11 reopens the output files described above. Such information as the object module name, the program version number, and the global symbol directory (GSD) for each program section are output to the object file to be used later in linking the object modules. After writing out the GSD for a given program section, MACRO-11 scans through the symbol tables to find all the global symbols that are bound to that particular program section. MACRO-11 then writes out GSD records to the object file for these symbols. This process is done for each program section.

1.2 ASSEMBLY PASS 2

On pass 2 MACRO-11 writes the object records to the output file while generating both the assembly listing and the symbol table listing for the program. A cross-reference listing may also be generated.

Basically, assembly pass 2 consists of the same steps performed in assembly pass 1, except that all source statements containing MACRO-11-detected errors are flagged with an error code as the assembly listing file is created. The object file that is created as the final consequence of pass 2 contains all the object records, together with relocation records that hold the information necessary for linking the object file.

The information in the object file, when passed to the Task Builder or Linker, enables the global symbols in the object modules to be associated with absolute or virtual memory addresses, thereby forming an executable body of code.

The user may wish to become familiar with the macro object file format and description. This information is presented in the applicable system manual (see Section 0.3 in the Preface).

CHAPTER 2

SOURCE PROGRAM FORMAT

2.1 PROGRAMMING STANDARDS AND CONVENTIONS

Programming standards and conventions allow code written by a person (or group) to be easily understood by other people. These standards also make the program easier to:

Plan
Comprehend
Test
Modify
Convert

The actual standard used must meet local user requirements. A sample coding standard is provided in Appendix E. Used by DIGITAL and its users, this coding example simplifies both communications and the continuing task of software maintenance and improvement.

2.2 STATEMENT FORMAT

A source program is composed of assembly-language statements. Each statement must be completed on one line. Although a line may contain 132 characters (a longer line causes an error (L) in the assembly listing) a line of 80 characters is recommended because of constraints imposed by listing format and terminal line size. Blank lines, although legal, have no significance in the source program.

A MACRO-11 statement may have as many as four fields. These fields are identified by their order within the statement and/or by the separating characters between the fields. The general format of a MACRO-11 statement is:

```
[Label:]  Operator Operand          [;Comment(s)]
```

The label and comment fields are optional. The operator and operand fields are interdependent; in other words, when both fields are present in a source statement, each field is evaluated by MACRO-11 in the context of the other.

A statement may contain an operator and no operand, but the reverse is not true. A statement containing an operand with no operator is illegal and is interpreted by MACRO-11 during assembly as an implicit .WORD directive (see Section 6.3.2).

MACRO-11 interprets and processes source program statements one by one. Each statement causes MACRO-11 either to perform a specified assembly process or to generate one or more binary instructions or data words.

2.2.1 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 current location counter is used by MACRO-11 to assign memory addresses to the source program statements as they are encountered during the assembly process. Thus, a label is a means of symbolically referring to a specific statement.

When a program section is absolute, the value of the current location counter is absolute; its value references an absolute virtual memory address (such as location 100). Similarly, when a program section is relocatable, the value of the current location counter is relocatable; a relocation bias calculated at link time is added to the apparent value of the current location counter to establish its effective absolute virtual address at execution time. (For a discussion of program sections and their attributes, see Section 6.7.)

If present, a label must be the first field in a source statement and must be terminated by a colon (:). For example, if the value of the current location counter is absolute 100(8), the statement:

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

assigns the value 100(8) to the label ABCD. If the location counter value were relocatable, the final value of ABCD would be 100(8)+K, where K represents the relocation bias of the program section, as calculated by the Task Builder or Linker at link time.

More than one label may appear within a single label field. Each label so specified is assigned the same address value. For example, if the value of the current location counter is 100(8), the multiple labels in the following statement are each assigned the value 100(8):

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

Multiple labels may also appear on successive lines. For example, the statements

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

likewise cause the same value to be assigned to all three labels. This second method of assigning multiple labels is preferred because positioning the fields consistently within the source program makes the program easier to read (see Section 2.3).

A double colon (::) defines the label as a global symbol. For example, the statement

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

establishes the label ABCD as a global symbol. The distinguishing attribute of a global symbol is that it can be referenced from within an object module other than the module in which the symbol is defined (see Section 6.8) or by independently assembled object modules. References to this label in other modules are resolved when the modules are linked as a composite executable image.

The legal characters for defining labels are:

A through Z
0 through 9
. (Period)
$ (Dollar Sign)

NOTE

By convention, the dollar sign ($) and period (.) are reserved for use in defining DIGITAL system software symbols. Therefore these characters should not be used in defining labels in MACRO-11 source programs.

A label may be any length; however, only the first six characters are significant and, therefore, must be unique among all the labels in the source program. An error code (M) is generated in the assembly listing if the first six characters in two or more labels are the same (see Appendix D).

A symbol used as a label must not be redefined within the source program. If the symbol is redefined, a label with a multiple definition results, causing MACRO-11 to generate an error code (M) in the assembly listing (see Appendix D). Furthermore, any statement in the source program which references a multi-defined label generates an error code (D) in the assembly listing (see Appendix D).

2.2.2 Operator Field

The operator field specifies the action to be performed. It may consist of an instruction mnemonic (op code), an assembler directive, or a macro call. Chapters 6 and 7 describe these three types of operators.

When the operator is an instruction mnemonic, a machine instruction is generated and MACRO-11 evaluates the addresses of the operands which follow. When the operator is a directive MACRO-11 performs certain control actions or processing operations during the assembly of the source program. When the operator is a macro call, MACRO-11 inserts the code generated by the macro expansion.

Leading and trailing spaces or tabs in the operator field have no significance; such characters serve only to separate the operator field from the preceding and following fields.

An operator is terminated by a space, tab, or any non-RAD50 character*,
as in the following examples:

```
MOV A,B                 ;THE SPACE TERMINATES THE OPERATOR
                        ;MOV.

MOV     A,B             ;THE TAB TERMINATES THE OPERATOR MOV.

MOV@A,B                 ;THE @ CHARACTER TERMINATES THE
                        ;OPERATOR MOV.
```

Although the statements above are all equivalent in function, the second statement is the recommended form because it conforms to MACRO-11 coding conventions.

---

* Appendix A.2 contains a table of Radix-50 characters.

2.2.3 Operand Field

When the operator is an instruction mnemonic (op code), the operand field contains program variables that are to be evaluated/manipulated by the operator. The operand field may also supply arguments to MACRO-11 directives and macro calls, as described in Chapters 6 and 7, respectively.

Operands may be expressions or symbols, depending on the operator. Multiple expressions used in the operand field of a MACRO-11 statement must be separated by a comma; multiple symbols similarly used may be delimited by any legal separator (a comma, tab, and/or space). An operand should be preceded by an operator field; if it is not, the statement is treated by MACRO-11 as an implicit .WORD directive (see Section 6.3.2).

When the operator field contains an op code, associated operands are always expressions, as shown in the following statement:

```
        MOV     R0,A+2(R1)
```

On the other hand, when the operator field contains a MACRO-11 directive or a macro call, associated operands are normally symbols, as shown in the following statement:

```
        .MACRO  ALPHA  SYM1,SYM2
```

Refer to the description of each MACRO-11 directive (Chapter 7) to determine the type and number of operands required in issuing the directive.

The operand field is terminated by a semicolon when the field is followed by a comment. For example, in the following statement:

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

the tab between MOV and A terminates the operator field and defines the beginning of the operand field; a comma separates the operands A and B; and a semicolon terminates the operand field and defines the beginning of the comment field. When no comment field follows, the operand field is terminated by the end of the source line.

2.2.4 Comment Field

The comment field normally begins in column 33 and extends through the end of the line. This field is optional and may contain any ASCII characters except null, RUBOUT, carriage-return, line-feed, vertical-tab or form-feed. All other characters appearing in the comment field, even special characters reserved for use in MACRO-11, are checked only for ASCII legality and then included in the assembly listing as they appear in the source text.

All comment fields must begin with a semicolon (;). When lengthy comments extend beyond the end of the source line (column 80), the comment may be resumed in a following line. Such a line must contain a leading semicolon, and it is suggested that the body of the comment be continued in the same columnar position in which the comment began. A comment line can also be included as an entirely separate line within the code body.

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

2.3 FORMAT CONTROL

Horizontal 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 string, or unless they are used as the operator field terminator. Thus, the space and tab characters can be used to provide an orderly and readable source program.

DIGITAL's standard source line format is shown below:

Label - begins in column 1

Operator - begins in column 9

Operands - begin in column 17

Comments - begin in column 33.

These formatting conventions are not mandatory; free-field coding is permissible. However, note the increased readability after formatting in the example below:

```
REGTST:BIT#MASK,VALUE;COMPARES BITS IN OPERANDS.

1       9       17              33      (columns)

REGTST: BIT     #MASK,VALUE     ;COMPARES BITS IN OPERANDS.
```

Page formatting and assembly listing considerations are discussed in Chapter 6 in the context of MACRO-11 directives that may be specified to accomplish desired formatting operations. Appendix E contains a sample coding standard.

PART II

PROGRAMMING IN MACRO-11 ASSEMBLY LANGUAGE

CHAPTER 3

SYMBOLS AND EXPRESSIONS

This chapter describes the components of MACRO-11 instructions: the character set, the conventions observed in constructing symbols, and the use of 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 (see Section 6.2, .ENABL LC).

2. The digits 0 through 9.

3. The characters . (period) and $ (dollar sign). These characters are reserved for use as Digital Equipment Corporation system program symbols.

4. The special characters listed in Table 3-1.

Table 3-1
Special Characters Used in MACRO-11

Character	Designation	Function
:	Colon	Label terminator.
::	Double colon	Label terminator; defines the label as a global label.
=	Equal sign	Direct assignment operator and macro keyword indicator.
==	Double equal sign	Direct assignment operator; defines the symbol as a global symbol.
=:	Equal sign colon*	Direct assignment operator; macro keyword indicator; causes error (M) in listing if an attempt is made to change the value of the symbol.

* RT-11 V4.0 only.

(continued on next page)

Table 3-1 (Cont.)
Special Characters Used in MACRO-11

Character	Designation	Function
==:	Double equal sign colon*	Direct assignment operator; defines the symbol as a global symbol; causes error (M) in listing if an attempt is made to change the value of the symbol.
%	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.
.	Period	Current location counter.
,	Comma	Operand field separator.
;	Semicolon	Comment field indicator.
<	Left angle bracket	Initial argument or expression indicator.
>	Right angle bracket	Terminal argument or expression indicator.
+	Plus sign	Arithmetic addition operator or autoincrement indicator.
-	Minus sign	Arithmetic subtraction operator or autodecrement indicator.
*	Asterisk	Arithmetic multiplication operator.
/	Slash	Arithmetic division operator.
&	Ampersand	Logical AND operator.
!	Exclamation point	Logical inclusive OR operator.
"	Double quote	Double ASCII character indicator.

* RT-11 V4.0 only.

(continued on next page)

Table 3-1 (Cont.)
Special Characters Used in MACRO-11

Character	Designation	Function
'	Single quote	Single ASCII character indicator; or concatenation indicator.
^	Up arrow or circumflex	Universal unary operator or argument indicator.
\	Backslash	Macro call numeric argument indicator.

3.1.1 Separating and Delimiting Characters

Legal separating characters and legal argument delimiters are defined in Tables 3-2 and 3-3 respectively.

Table 3-2
Legal Separating Characters

Character	Definition	Usage
Space	One or more spaces and/or tabs	A space is a legal separator between instruction fields and between symbolic arguments within the operand field. Spaces within expressions are ignored (see Section 3.9).
	Comma	A comma is a legal separator between symbolic arguments within the operand field. Multiple expressions used in the operand field must be separated by a comma.

3.1.2 Illegal Characters

A character is illegal for one of two reasons:

1. If a character is not an element of the recognized MACRO-11 character set, it is replaced in the listing by a question mark, and an error code (I) is printed in the assembly listing (see Appendix D). The exception to this is an embedded null which, when detected, terminates the scan of the current line.

2. If a legal MACRO-11 character is used in a source statement with illegal or questionable syntax, an error code (Q) is printed in the assembly listing.

Table 3-3
Legal Argument Delimiters

Character	Definition	Usage
<...>	Paired angle brackets	Paired angle brackets may be used anywhere in a program to enclose an expression for treatment as a single term. Paired angle brackets are also used to enclose a macro argument, particularly when that argument contains separating characters (see Section 7.3).
^x...x	Up-arrow (unary operator) construction, where the up-arrow is followed by an argument that is bracketed by any paired printing characters (x).	This construction is equivalent in function to the paired angle brackets described above and is generally used only where the argument itself contains angle brackets.

3.1.3 Unary and Binary Operators

Legal MACRO-11 unary operators are described in Table 3-4. Unary operators are used in connection with single terms (arguments or operands) to indicate an action to be performed on that term during assembly. Because a term preceded by a unary operator is considered to contain that operator, a term so specified can be used alone or as an element of an expression.

Table 3-4
Legal Unary Operators

Unary Operator	Explanation	Example	Effect
+	Plus sign	+A	Produces the positive value of A.
-	Minus sign	-A	Produces the negative (2's complement) value of A.

(continued on next page)

Table 3-4 (Cont.)
Legal Unary Operators

Unary Operator	Explanation	Example	Effect
^	Up-arrow, universal unary operator. (This usage is described in detail in Section 6.4.)	^C24	Produces the 1's complement value of 24(8).
		^D127	Interprets 127 as a decimal number.
		^F3.0	Interprets 3.0 as a 1-word, floating-point number.
		^O34	Interprets 34 as an octal number.
		^B11000111	Interprets 11000111 as a binary number.
		^RABC	Evaluates ABC in Radix-50 form.

Unary operators can be used adjacent to each other or in constructions involving multiple terms, as shown below:

```
-^D50    (Equivalent to -<^D50>)
^C^O12   (Equivalent to ^C<^O12>)
```

Legal MACRO-11 binary operators are described in Table 3-5. In contrast to unary operators, binary operators specify actions to be performed on multiple items or terms within an expression.

Table 3-5
Legal Binary Operators

Binary Operator	Explanation	Example
+	Addition	A+B
-	Subtraction	A-B
*	Multiplication	A*B (signed 16-bit product returned)
/	Division	A/B (signed 16-bit quotient returned)
&	Logical AND	A&B
!	Logical inclusive OR	A!B

All binary operators have equal priority. Terms enclosed by angle brackets are evaluated first, and remaining operations are performed from left to right, as shown in the examples below:

```
.WORD   1+2*3           ;EQUALS 11(8).
.WORD   1+<2*3>         ;EQUALS 7(8).
```

3.2 MACRO-11 SYMBOLS

MACRO-11 maintains a symbol table for each of the three symbol types that may be defined in a MACRO-11 source program: the Permanent Symbol Table (PST), the User Symbol Table (UST), and the Macro Symbol Table (MST). The PST contains all the permanent symbols defined within (and thus automatically recognized by) MACRO-11 and is part of the MACRO-11 image. The UST (for user-defined symbols) and MST (for macro symbols) are constructed as the source program is assembled.

3.2.1 Permanent Symbols

Permanent symbols consist of the instruction mnemonics (see Appendix C) and MACRO-11 directives (see Chapters 6 and 7 and Appendix B). These symbols are a permanent part of the MACRO-11 image and need not be defined before being used in the operator field of a MACRO-11 source statement (see Section 2.2.2).

3.2.2 User-Defined and Macro Symbols

User-defined symbols are those symbols that are equated to a specific value through a direct assignment statement (see Section 3.3), appear as labels (see Section 2.2.1), or act as dummy arguments (see Section 7.1.1). These symbols are added to the User Symbol Table as they are encountered during assembly.

Macro symbols are those symbols used as macro names (see Section 7.1). They are added to the Macro Symbol Table as they are encountered during assembly.

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

1. Symbols can be composed of alphanumeric characters, dollar signs ($), and periods (.) only (see Note below).

2. The first character of a symbol must not be a number (except in the case of local symbols; see Section 3.5).

3. The first six characters of a symbol must be unique. A symbol can be written with more than six legal characters, but the seventh and subsequent characters are checked only for ASCII legality and are not otherwise evaluated or recognized by MACRO-11.

4. Spaces, tabs, and illegal characters must not be embedded within a symbol. The legal MACRO-11 character set is defined in Section 3.1.

NOTE

The dollar sign ($) and period (.) characters are reserved for use in defining Digital Equipment Corporation system software symbols. For example, READ$ is a file-processing system macro. The user is cautioned not to employ these characters in constructing user-defined symbols or macro symbols in order to avoid possible conflicts with existing or future Digital Equipment Corporation system software symbols.

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 described above; permanent, user-defined, or macro. To determine the value of an operator-field symbol, MACRO-11 searches the symbol tables in the following order:

1. Macro Symbol Table
2. Permanent Symbol Table
3. User-Defined Symbol Table

This search order allows permanent symbols to be used as macro symbols. But the user must keep in mind the sequence in which the search for symbols is performed in order to avoid incorrect interpretation of the symbol's use.

When a symbol appears in the operand field, the search order is:

1. User-Defined Symbol Table
2. Permanent Symbol Table

Depending on their use in the source program, user-defined symbols have either a local (internal) attribute or a global (external) attribute.

Normally, MACRO-11 treats all user-defined symbols as local, that is, their definition is limited to the module in which they appear. However, symbols can be explicitly declared to be global symbols through one of three methods:

1. Use of the .GLOBL directive (see Section 6.8).
2. Use of the double colon (::) in defining a label (see Section 2.2.1).
3. Use of the double equal sign (==) or double equal colon sign (==:) in a direct assignment statement (see Section 3.3)*.

All symbols within a module that remain undefined at the end of assembly are treated as default global references.

* RT-11 V4.0 only.

NOTE

Undefined symbols at the end of assembly are assigned a value of 0 and placed into the user-defined symbol table as undefined default global references. If the .DSABL GBL directive is in effect, however, (see Section 6.2) the statement containing the undefined symbol is flagged with an error code (U) in the assembly listing (see Appendix D).

Global symbols provide linkages between independently-assembled object modules within the task image. A global symbol defined as a label, for example, may serve as an entry-point address to another section of code within the image. Such symbols are referenced from other source modules in order to transfer control throughout execution. These global symbols are resolved at link time, ensuring that the resulting image is a logically coherent and complete body of code.

3.3 DIRECT ASSIGNMENT STATEMENTS

The General Format for a direct assignment statement is:

symbol=expression

or

symbol==expression

where: expression - can have only one level of forward reference (see 5. below).

- cannot contain an undefined global reference.

The Colon Format* for a direct assignment statement is:

symbol=:expression

or

symbol==:expression

where: expression - can have only one level of forward reference (see 5. below).

- cannot contain an undefined global reference.

All the direct assignment statements above allow the user to equate a symbol with a specific value. After the symbol has been defined it is entered into the User-Defined Symbol Table. If the general format is used (= or ==) the value of the symbol may be changed in subsequent direct assignment statements. If, however, the colon format is used (=: or ==:) any attempt to change the value of the symbol will generate an error (M) in the assembly listing.

A direct assignment statement embodying either the double equal (==) sign or the double equal colon (==:) sign, as shown above, defines the symbol as global (see Section 6.8).

---

* RT-11 V4.0 only.

The following examples illustrate the coding of direct assignment statements.

Example 1:

```
A=10                            ;DIRECT ASSIGNMENT

B==30                           ;GLOBAL ASSIGNMENT

A=15                            ;LEGAL REASSIGNMENT

L=:5                            ;EQUAL COLON ASSIGNMENT*

M==:A+2                         ;DOUBLE EQUAL COLON ASSIGNMENT*
                                ;M BECOMES EQUAL TO 17

L=4                             ;ILLEGAL REASSIGNMENT
                                ;M ERROR IS GENERATED
```

Example 2:

```
C:
D=.
E:      MOV     #1,ABLE         ;THE SYMBOL D IS EQUATED TO ., AND
                                ;THE LABELS C AND E ARE ASSIGNED A
                                ;VALUE THAT IS EQUAL TO THE LOCATION
                                ;OF THE MOV INSTRUCTION.
```

The code in the second example above would not usually be used and is shown only to illustrate the performance of MACRO-11 in such situations. See Section 3.6 for a description of the period (.) as the current location counter symbol.

The following conventions apply to the coding of direct assignment statements:

1. An equal sign (=), double equal sign (==), equal colon sign* (=:) or double equal colon sign* (==:) must separate the symbol from the expression defining the symbol's value. Spaces preceding and/or following the direct assignment operators, although permissible, have no significance in the resulting value.

2. The symbol being assigned in a direct assignment statement is placed in the label field.

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

4. A direct assignment statement may be followed only by a comment field.

5. Only one level of forward referencing is allowed. The following example would cause an error code (U) in the assembly listing on the line containing the illegal forward reference:

```
X=Y     (Illegal forward reference)

Y=Z     (Legal forward reference)

Z=1
```

* RT-11 V4.0 only.

Although one level of forward referencing is allowed for local symbols, no forward referencing is allowed for global symbols. In other words, the expression being assigned to a global symbol can contain only previously defined symbols. A forward reference in a direct assignment statement defining a global symbol will cause an error code (A) to be generated in the assembly listing.

3.4 REGISTER SYMBOLS

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

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

where % indicates a reference to a register rather than a location. The digit specifying the register can be replaced by any legal, absolute term that can be evaluated during the first assembly pass.

The register definitions listed below are the normal default values and remain valid for all register references within the source program.

```
R0=%0                           ;REGISTER 0 DEFINITION.
R1=%1                           ;REGISTER 1 DEFINITION.
R2=%2                           ;REGISTER 2 DEFINITION.
R3=%3                           ;REGISTER 3 DEFINITION.
R4=%4                           ;REGISTER 4 DEFINITION.
R5=%5                           ;REGISTER 5 DEFINITION.
SP=%6                           ;STACK POINTER DEFINITION.
PC=%7                           ;PROGRAM COUNTER DEFINITION.
```

Registers 6 and 7 are given special names because of their unique system functions. The symbolic default names assigned to the registers, as listed above, are the conventional names used in all DIGITAL-supplied PDP-11 system programs. For this reason, you are advised to follow these conventions.

A register symbol may be defined in a direct assignment statement appearing in the program. The defining expression of a register symbol must be a legal, absolute value between 0 and 7, inclusive, or an error code (R) will appear in the assembly listing. Although you can reassign the standard register symbols through the use of the .DSABL REG directive (see Section 6.2), this practice is not recommended. An attempt to redefine a default register symbol without first specifying the .DSABL REG directive to override the normal register definitions causes that assignment statement to be flagged with an error code (R) in the assembly listing. All non-standard register symbols must be defined before they are referenced in the source program.

The % character may be used with any legal term or expression to specify a register. For example, the statement

```
CLR     %3+1
```

is equivalent in function to the statement

```
CLR     %4
```

and clears the contents of register 4.

In contrast, the statement

```
CLR     4
```

clears the contents of virtual memory location 4.

3.5 LOCAL SYMBOLS

Local symbols are specially formatted symbols used as labels within a block of coding that has been delimited as a local symbol block. Local symbols are of the form n$, where n is a decimal integer from 1 to 65535, inclusive. Examples of local symbols are:

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

A local symbol block is delimited in one of three ways:

1. The range of a local symbol block usually consists of those statements between two normally-constructed symbolic labels (see Figure 3-1). Note that a statement of the form:

   ALPHA=expression

   is a direct assignment statement (see Section 3.3) but does not create a label and thus does not delimit the range of a local symbol block.

2. The range of a local symbol block is normally terminated upon encountering a .PSECT, .CSECT, .ASECT, or .RESTORE directive in the source program (see Figure 3-1).

3. The range of a local symbol block is delimited through MACRO-11 directives, as follows:

   Starting delimiter: .ENABL LSB (see Section 6.2)

   Ending delimiter: .ENABL LSB

   or

   one of the following:

   Symbolic label (see Section 2.2.1)
   .PSECT (see Section 6.7.1)
   .CSECT (see Section 6.7.2)
   .ASECT (see Section 6.7.2)
   .RESTORE (see Section 6.7.4)

   encountered after a .DSABL LSB (see Section 6.2).

Local symbols provide a convenient means of generating labels for branch instructions and other such references within a local symbol blocks. Using local symbols reduces the possibility of symbols with multiple definitions appearing within a user program. In addition, the use of local symbols differentiates entry-point labels from local labels, since local symbols cannot be referenced from outside their respective local symbol blocks. Thus, local symbols of the same name can appear in other local symbol blocks without conflict. Local symbols do not appear in cross-reference listings and require less symbol table space than other types of symbols. Their use is recommended.

When defining local symbols, use the range from 1$ to 63$ first, then the range from 128$ to 65535$. Local symbols within the range 64$ through 127$, inclusive, can be generated automatically as a feature of MACRO-11. Such local symbols are useful in the expansion of macros during assembly (see Section 7.3.5).

Be sure to avoid multiple definitions of local symbols within the same local symbol block. For example, if the local symbol 10$ is defined two or more times within the same local symbol block, each symbol represents a different address value. Such a multi-defined symbol causes an error code (P) to be generated in the assembly listing.

For examples of local symbols and local symbol blocks as they appear in a source program, see Figure 3-1.

```
 1                      ;+
 2                      ; Simple illustration of local symbols; the second block is delimited
 3                      ; by the label XCTPAS.
 4                      ;-
 5
 6 000000  012700  XCTPRG: MOV     #IMPURE,R0      ;Point to impure area
           000000G
 7 000004  005020  1$:     CLR     (R0)+           ;Clear a word
 8 000006  020027          CMP     R0,#IMPURT      ;Test if at top of area
           000000G
 9 000012  001374          BNE     1$              ;Iterate if not
10                                                 ;Fall in to perform pass initialization
11
12 000014  012700  XCTPAS: MOV     #IMPPAS,R0      ;Point to pass storage area
           000000G
13 000020  005020  1$:     CLR     (R0)+           ;Clear the area
14 000022  020027          CMP     R0,#IMPPAT      ;Test if at top of area
           000000G
15 000026  001374          BNE     1$              ;Iterate of not
16 000030  000207          RTS     PC              ;Return if so
```

Figure 3-1 Assembly Listing Showing Local Symbol Block

3.6 CURRENT LOCATION COUNTER

The period (.) is the symbol for the current location counter. When used in the operand field of an instruction, the period represents the address of the first word of the instruction, as shown in the first example below. When used in the operand field of a MACRO-11 directive, it represents the address of the current byte or word, as shown in the second example below.

```
A:      MOV     #.,R0           ;THE PERIOD (.) REFERS TO THE ADDRESS
                                ;OF THE MOV INSTRUCTION.
```

(The function of the # symbol is explained in Section 5.9.)

```
SAL=0
        .WORD   177535,.+4,SAL  ;THE OPERAND .+4 IN THE .WORD
                                ;DIRECTIVE REPRESENTS A VALUE
                                ;THAT IS STORED AS THE SECOND
                                ;OF THREE WORDS DURING
                                ;ASSEMBLY.
```

Assume that the current value of the location counter is 500. During assembly, MACRO-11 reserves storage in response to the .WORD directive (see Section 6.3.2), beginning with location 500. The operands accompanying the .WORD directive determine the values so stored. The value 177535 is thus stored in location 500. The value represented by .+4 is stored in location 502; this value is derived as the current value of the location counter (which is now 502), plus the absolute value 4, thereby depositing the value 506 in location 502. Finally, the value of SAL, previously equated to 0, is deposited in location 504.

Figure 3-2 illustrates the result of the example.

LOCATION	CONTENTS
500	177535
502	506
504	0

Figure 3-2 Sample Assembly Results

At the beginning of each assembly pass, MACRO-11 resets the location counter. Normally, consecutive memory locations are assigned to each byte of object data generated. However, the value of the location counter can be changed through a direct assignment statement of the following form:

```
.=expression
```

The current location counter symbol (.) is either absolute or relocatable, depending on the attribute of the current program section.

The attribute of the current location counter can be changed only through the program sectioning directives (.PSECT, .ASECT, .CSECT and .RESTORE), as described in Section 6.7. Therefore, assigning to the counter an expression having an attribute different than that of the current program section will generates an error code (A) in the assembly listing.

Furthermore, an expression assigned to the counter may not contain a forward reference (a reference to a symbol that is not previously defined). The user must also be sure that the expression assigned will not force the counter into another program section, even if both sections involved have the same relocatability. Either of these conditions generates an error code (A) in the assembly listing.

The following coding illustrates the use of the current location counter:

```
        .ASECT
.=500                                   ;SET LOCATION COUNTER TO
                                        ;ABSOLUTE 500(OCTAL).
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 THE 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 THE
                                        ;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 areas may be reserved in the program by advancing the location counter. For example, if the current value of the location counter is 1000, each of the following statements:

```
.=.+40
```

or

```
.BLKB 40
```

or

```
.BLKW 20
```

reserves 40(8) bytes of storage space in the source program. The .BLKB and .BLKW directives, however, are the preferred ways to reserve storage space (see Section 6.5.3).

3.7 NUMBERS

MACRO-11 assumes that all numbers in the source program are to be interpreted in octal radix, unless otherwise specified. An exception to this assumption is that operands associated with Floating Point Processor instructions and Floating Point Data directives are treated as decimal (see Section 6.4.2). This default radix can be altered with the .RADIX directive (see Section 6.4.1.1). Also, individual numbers can be designated as decimal, binary, or octal numbers through temporary radix control operators (see Section 6.4.1.2).

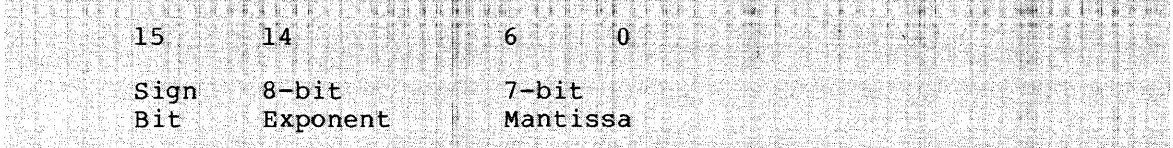
For every statement in the source program that contains a digit that is not in the current radix, an error code (N) is generated in the assembly listing. However, MACRO-11 continues with the scan of the statement and evaluates each such number encountered as a decimal value.

Negative numbers must be preceded by a minus sign; MACRO-11 translates such numbers into two's complement form. Positive numbers may (but need not) be preceded by a plus sign.

A number containing more than 16 significant bits (greater than 177777(8)), is truncated from the left and flagged with an error code (T) in the assembly listing.

Numbers are always considered to be absolute values; therefore, they are never relocatable.

Single-word floating-point numbers may be generated with the ^F operator (see Section 6.4.2.2) and are stored in the following format:



Refer to the appropriate PDP-11 Processor Handbook for details of the floating-point number format.

3.8 TERMS

A term is a component of an expression and 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 in Section 3.2. Symbols are evaluated as follows:

   a. A period (.) specified in an expression causes the value of the current location counter to be used.

   b. A defined symbol is located in the User-Defined Symbol Table (UST) and its value is used.

   c. A permanent symbol's basic value is used, with zero substituted for the addressing modes. (Appendix C lists all op codes and their values.)

   d. An undefined symbol is assigned a value of zero and inserted in the User-Defined Symbol Table as an undefined default global reference. If the .DSABL GBL directive (see Section 6.2) is in effect, the automatic global reference default function of MACRO-11 is inhibited, and the statement containing the undefined symbol is flagged with an error code (U) in the assembly listing.

3. A single quote followed by a single ASCII character, or a double quote followed by two ASCII characters. This type of expression construction is explained in detail in Section 6.3.3.

4. An expression enclosed in angle brackets (<>). Any expression so enclosed is evaluated and reduced to a single term before the remainder of the expression in which it appears is evaluated. Angle brackets, for example, may be used to alter the left-to-right evaluation of expressions (as in A*B+C versus A*<B+C>), or to apply a unary operator to an entire expression (as in -<A+B>).

5. A unary operator followed by a symbol or number.

3.9 EXPRESSIONS

Expressions are combinations of terms joined together by binary operators (see Table 3-5). Expressions reduce to a 16-bit value. The evaluation of an expression includes the determination of its attributes. A resultant expression value may be any one of four types (as described later in this section): relocatable, absolute, external, or complex relocatable.

Expressions are evaluated from left to right with no operator hierarchy rules, except that unary operators take precedence over binary operators. A term preceded by a unary operator is considered to contain that 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 or illegal operator terminates the expression analysis, causing error codes (A) and/or (Q), to be generated in the assembly listing, depending on the context of the expression itself. For example, the expression:

```
A + B 177777
```

is evaluated as

```
A + B
```

because the first non-blank character following the symbol B is not a legal binary operator, an expression separator (a comma), or an operand field terminator (a semicolon or the end of the source line).

NOTE

Spaces within expressions can serve as delimiters only between symbols. In other words, the expressions

```
A + B
```

and

```
A+B
```

are the same, but the symbols

```
B17
```

and

```
B 17
```

are not (B 17 is not a single symbol).

At assembly time the value of an external (global) expression is equal to the value of the absolute part of that expression. For example, the expression EXTERN+A, where "EXTERN" is an external symbol, has a value at assembly time that is equal to the value of the internal (local) symbol A. This expression, however, when evaluated at link time takes on the resolved value of the symbol EXTERN, plus the value of symbol A.

Expressions, when evaluated by MACRO-11, are one of four types: relocatable, absolute, external, or complex relocatable. The following distinctions are important:

1. An expression is **relocatable** if its value is fixed relative to the base address of the program section in which it appears; it will have an offset value added at link time. Terms that contain labels defined in relocatable program sections will have a relocatable value; similarly, a period (.) in a relocatable program section, representing the value of the current location counter, will also have a relocatable value.

2. An expression is **absolute** if its value is fixed. An expression whose terms are numbers and ASCII conversion characters will reduce to an absolute value. A relocatable expression or term minus a relocatable term, where both elements being evaluated belong to the same program section, is an absolute expression. This is because every term in a program section has the same relocation bias. When one term is subtracted from another, the resulting bias is zero. MACRO-11 can then treat the expression as absolute and reduce it to a single term upon completion of the expression scan. Terms that contain labels defined in an absolute section will also have an absolute value.

3. An expression is **external** (or global) if it contains a single global reference (plus or minus an absolute expression value) that is not defined within the current program. Thus, an external expression is only partially defined following assembly and must be resolved at link time.

4. An expression is **complex relocatable** if any one of the following conditions applies:

   - It contains a global reference and a relocatable symbol.

   - It contains more than one global reference.

   - It contains relocatable terms belonging to different program sections.

   - The value resulting from the expression has more than one level of relocation. For example, if the relocatable symbols TAG1 and TAG2, associated with the same program section, are specified in the expression TAG1+TAG2, two levels of relocation will be introduced, since each symbol is evaluated in terms of the relocation bias in effect for the program section.

   - An operation other than addition is specified on an undefined global symbol.

   - An operation other than addition, subtraction, negation, or complementation is specified for a relocatable value.

The evaluation of relocatable, external, and complex relocatable expressions is completed at link time.

CHAPTER 4

RELOCATION AND LINKING

The output of MACRO-11 is an object module that must be processed or linked before it can be loaded and executed. Essentially, linking fixes (makes absolute) the values of relocatable or external symbols in the object module, thus transforming the object module, or several object modules, into an executable image.

To allow the value of an expression to be fixed at link time, MACRO-11 outputs certain instructions in the object file, together with other required parameters. For relocatable expressions in the object module, the base of the associated relocatable program section is added to the value of the relocatable expression provided by MACRO-11. For external expression values, the value of the external term in the expression (since the external symbol must be defined in one of the other object modules being linked together) is determined and then added to the absolute portion of the external expression, as provided by MACRO-11.

All instructions that require modification at link time are flagged in the assembly listing, as illustrated in the example below. The apostrophe (') following the octal expansion of the instruction indicates that simple relocation is required; the letter G indicates that the value of an external symbol must be added to the absolute portion of an expression; and the letter C indicates that complex relocation analysis at link time is required in order to fix the value of the expression.

EXAMPLE:

```
005065  CLR     RELOC(R5)           ;ASSUMING THAT THE VALUE OF THE
000040'                             ;SYMBOL "RELOC", 40, IS RELOCATABLE
                                    ;THE RELOCATION BIAS
                                    ;WILL BE ADDED TO THIS VALUE.

005065  CLR     EXTERN(R5)          ;THE VALUE OF THE SYMBOL "EXTERN" IS
000000G                             ;ASSEMBLED AS ZERO AND IS
                                    ;RESOLVED AT LINK TIME.

005065  CLR     EXTERN+6(R5)        ;THE VALUE OF THE SYMBOL "EXTERN"
000006G                             ;IS RESOLVED AT LINK TIME
                                    ;AND ADDED TO
                                    ;THE ABSOLUTE PORTION (+6) OF
                                    ;THE EXPRESSION.

005065  CLR     =<EXTERN+RELOC>(R5) ;THIS EXPRESSION IS COMPLEX
000000C                             ;RELOCATABLE BECAUSE IT REQUIRES
                                    ;THE NEGATION OF AN EXPRESSION
                                    ;THAT CONTAINS A GLOBAL "EXTERN"
                                    ;REFERENCE AND A RELOCATABLE TERM.
```

For a complete description of object records output by MACRO-11, refer to the applicable system manual (see Section 0.3 in the Preface).

CHAPTER 5

ADDRESSING MODES

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

> "whenever the processor implicitly uses the program counter (PC) to fetch a word from memory, the program counter is automatically incremented by 2 after the fetch operation is completed."

The PC always contains the address of the next word to be fetched. This word will be either the address of the next instruction to be executed, or the second or third word of the current instruction.

Table 5-1 lists the address modes, and Table 5-2 lists the symbols used in this chapter to describe the address modes. Each mode of address in the chapter is illustrated using either the single operand instruction CLR or the double operand instruction MOV.

Table 5-1
Addressing Modes

Mode	Form	Section Reference
Register mode*	R	5.1
Register deferred mode*	@R or (ER)	5.2
Autoincrement mode*	(ER)+	5.3
Autoincrement deferred mode*	@(ER)+	5.4
Autodecrement mode*	-(ER)	5.5
Autodecrement deferred mode*	@-(ER)	5.6
Index mode**	E(ER)	5.7
Index deferred mode**	@E(ER)	5.8
Immediate mode**	#E	5.9
Absolute mode**	@#E	5.10
Relative mode**	E	5.11
Relative deferred mode**	@E	5.12
Branch	Address	5.13

* Does not increase the length of an instruction.

** Adds one word to the instruction length for each occurrence of an operand of this form.

Table 5-2
Symbols Used in Chapter 5

Symbol	Explanation
E	Any expression, as defined in Chapter 3.
R	A register expression; in other words, any expression containing a term preceded by a percent sign (%) or a symbol previously equated to such a term, as shown below: R0=%0 ;GENERAL REGISTER 0. R1=R0+1 ;GENERAL REGISTER 1. R2=1+%1 ;GENERAL REGISTER 2. This symbol may also represent any of the normal default register definitions (see Section 3.4).
ER	A register expression or an absolute expression in the range 0 to 7, inclusive.

5.1 REGISTER MODE

Format:

```
R
```

The register itself (R) contains the operand to be manipulated by the instruction.

Example:

```
CLR     R3              ;CLEARS REGISTER 3.
```

5.2 REGISTER DEFERRED MODE

Format:

```
@R or (ER)
```

The register (R) contains the address of the operand to be manipulated by the instruction.

Examples:

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

5.3 AUTOINCREMENT MODE

Format:

```
(ER)+
```

The contents of the register (ER) are incremented immediately after being used as the address of the operand (see Note below).

Examples:

```
CLR     (R0)+           ;EACH INSTRUCTION CLEARS
CLR     (R4)+           ;THE WORD AT THE ADDRESS
CLR     (R2)+           ;CONTAINED IN THE SPECIFIED
                        ;REGISTER AND INCREMENTS
                        ;THAT REGISTER'S CONTENTS
                        ;BY TWO.
```

NOTE

Certain special instruction/address mode combinations, which are rarely or never used, do not operate the same on all PDP-11 processors, as described below.

In the autoincrement mode, both the JMP and JSR instructions autoincrement the register before its use on the PDP-11/40 but not on the PDP-11/45 or 11/10.

In double operand instructions having the addressing form Rn,(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 example, as executed on the PDP-11/40, Register 0 originally contains 100(8):

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

MOV     R0,-(R0)        ;THE QUANTITY 76 IS MOVED
                        ;TO LOCATION 100.
```

The use of these forms should be avoided, since they are not compatible with the entire family of PDP-11 processors.

An error code (Z) is printed in the assembly listing with each instruction which is not compatible among all members of the PDP-11 family.

5.4 AUTOINCREMENT DEFERRED MODE

Format:

```
@(ER)+
```

The register (ER) contains a pointer to the address of the operand. The contents of the register are incremented after being used as pointer.

Example:

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

5.5 AUTODECREMENT MODE

Format:

```
-(ER)
```

The contents of the register (ER) are decremented before being used as the address of the operand (see Note in Section 5.3).

Examples:

```
CLR     -(R0)           ;DECREMENT THE CONTENTS OF THE SPECI-
                        ;FIED REGISTER (0, 3, OR 2) BY TWO
CLR     -(R3)           ;BEFORE USING ITS CONTENTS
CLR     -(R2)           ;AS THE ADDRESS OF THE WORD TO BE
                        ;CLEARED.
```

5.6 AUTODECREMENT DEFERRED MODE

Format:

```
@-(ER)
```

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

Example:

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

5.7 INDEX MODE

Format:

```
E(ER)
```

An expression (E), plus the contents of a register (ER), yields the effective address of the operand. In other words, the value E is the offset of the instruction, and the contents of register ER form the base. (The value of the expression (E) is stored as the second or third word of the instruction.)

Examples:

```
CLR     X+2(R1)         ;THE EFFECTIVE ADDRESS OF THE WORD
                        ;TO BE CLEARED IS X+2, PLUS THE
                        ;CONTENTS OF REGISTER 1.
MOV     R0,-2(R3)       ;THE EFFECTIVE ADDRESS OF THE
                        ;DESTINATION LOCATION IS -2, PLUS
                        ;THE CONTENTS OF REGISTER 3.
```

5.8 INDEX DEFERRED MODE

Format:

```
@E(ER)
```

An expression (E), plus the contents of a register (ER), yields a pointer to the address of the operand. As in index mode above, the value E is the offset of the instruction, and the contents of register ER form the base. (The value of the expression (E) is stored as the second or third word of the instruction.)

Example:

```
CLR     @114(R4)        ;IF REGISTER 4 CONTAINS 100, THIS
                        ;VALUE, PLUS THE OFFSET 114, YIELDS
                        ;THE POINTER 214.  IF LOCATION 214
                        ;CONTAINS THE ADDRESS 2000, LOCATION
                        ;2000 WOULD BE CLEARED.
```

NOTE

The expression @(ER) may be used, but it will be assembled as if it were written @0(ER), and a word will be used to store the 0.

5.9 IMMEDIATE MODE

Format:

```
#E
```

Immediate mode allows the operand itself (E) to be stored as the second or third word of the instruction. The number sign (#) is an addressing mode indicator. Appearing in the operand field this character specifies the immediate addressing mode, indicating to MACRO-11 that the operand itself immediately follows the instruction word. This mode is assembled as an autoincrement of the PC.

Examples:

```
MOV     #100,R0         ;MOVE THE VALUE 100 INTO REGISTER 0.
MOV     #X,R0           ;MOVE THE VALUE OF SYMBOL X INTO
                        ;REGISTER 0.
```

The operation of this mode can be shown through the first example, MOV #100,R0, which assembles as two words:

```
Location 20:  0 1 2 7 0 0

Location 22:  0 0 0 1 0 0

Location 24:  Next instruction
```

The source operand (the value 100) is assembled immediately following the instruction word. Upon execution of the instruction, the processor fetches the first word (MOV) and increments the PC by 2 so that it points to the second word, location 22, which contains the source operand.

After the next fetch and increment cycle, the source operand (100) is moved into register 0, leaving the PC pointing to location 24 (the next instruction).

5.10 ABSOLUTE MODE

Format:

```
@#E
```

Absolute mode is the equivalent of immediate mode deferred. The address expression @#E specifies an absolute address which is stored as the second or third word of the instruction. In other words, the value immediately following the instruction word is taken as the absolute address of the operand. Absolute mode is assembled as an autoincrement deferred of the PC.

Examples:

```
MOV     @#100,R0        ;MOVE THE CONTENTS OF ABSOLUTE
                        ;LOCATION 100 INTO REGISTER R0.
CLR     @#X             ;CLEAR THE CONTENTS OF THE LOCATION
                        ;WHOSE ADDRESS IS SPECIFIED BY
                        ;THE SYMBOL X.
```

The operation of this mode can be shown through the first example, MOV @#100,R0, which assembles as two words:

```
Location 20:  0 1 3 7 0 0

Location 22:  0 0 0 1 0 0

Location 24:  Next instruction
```

The absolute address 100 is assembled immediately following the instruction word. Upon execution of the instruction, the processor fetches the first word (MOV) and increments the PC by 2 so that it points to the second word, location 22, which contains the absolute address of the source operand. After the next fetch and increment cycle, the contents of absolute address 100 (the source operand) are moved into register 0, leaving the PC pointing to location 24 (the next instruction).

5.11 RELATIVE MODE

Format:

E

Relative mode is the normal mode for memory references within your program. It is assembled as index mode, using the PC as the index register. The offset for the address calculation is assembled as the second or third word of the instruction. This value is added to the contents of the PC to yield the address of the source operand.

Examples:

```
CLR     100             ;CLEAR ABSOLUTE LOCATION 100
MOV     R0,Y            ;MOVE THE CONTENTS OF REGISTER 0
                        ;TO LOCATION Y
```

The operation of relative mode can be shown with the statement MOV 100,R3, which assembles as two words:

```
Location 20:  0 1 6 7 0 3

Location 22:  0 0 0 0 5 4

Location 24:  Next instruction
```

The offset, the constant 54, is assembled immediately following the instruction word. Upon execution of the instruction, the processor fetches the first word (MOV) and increments the PC by 2 so that it points to the second word, location 22, containing the value 54. After the next fetch and increment cycle, the processor calculates the effective address of the source operand by taking the contents of location 22 (the offset) and adding it to the current value of the PC, which now points to location 24 (the next instruction). Thus, the source operand address is the result of the calculation OFFSET+PC = 54+24 = 100(8), causing the contents of location 100 to be moved into register 3.

The index mode statement:

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

is equivalent to the relative mode statement:

```
MOV     100,R3.
```

100-.-4 is the offset for the index mode statement. The current location counter (.) holds the address of the first word of the instruction (20, in this case) and the PC has to move down 4 bytes to reach location 24 (the next instruction). So, the offset could be written as 100-20-4 or 54(8).

Therefore, for the index mode, the offset (54(8)) added to the PC(24(8)) yields the effective address (54 + 24 = 100 (8)) of the operand.

Thus, both statements move the contents of location 100 into register 3.

NOTE

The addressing form @#E differs from 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 (see Section 5.10). 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 .ENABL AMA function in Section 6.2, which causes all relative mode addresses to be assembled as absolute mode addresses.

5.12 RELATIVE DEFERRED MODE

Format:

```
@E
```

The relative deferred mode is similar in operation to the relative mode above, except that the expression E is used as a pointer to the address of the operand. In other words, the operand following the instruction word is added to the contents of the PC to yield a pointer to the address of the operand.

Example:

```
MOV     @X,R0           ;RELATIVE TO THE CURRENT VALUE OF
                        ;THE PC, MOVE THE CONTENTS OF THE
                        ;LOCATION WHOSE ADDRESS IS POINTED
                        ;TO BY LOCATION X INTO REGISTER 0.
```

5.13 BRANCH INSTRUCTION ADDRESSING

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

1. Extends the sign of the offset through bits 8-15.
2. Multiplies the result by 2, creating a byte offset rather than a word offset.
3. Adds the result to the current value of the PC to form the effective branch address.

MACRO-11 performs the reverse operation to form the word offset from the specified address.

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

When the offset is added to the PC, the PC is moved to the next word (PC=.+2). Hence the -2 in the following calculation.

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

The following conditions generate an error code (A) in the assembly listing:

1. Branching from one program section to another
2. Branching to a location that is defined as an external (global) symbol
3. Specifying a branch address that is out of range, meaning that the branch offset is a value that does not lie within the range -128(10) to +127(10).

5.14 USING TRAP INSTRUCTIONS

Since the EMT and TRAP instructions do not use the low-order byte of the instruction word, information is transferred to the trap handlers in the low-order byte. If the EMT or TRAP instruction is followed by an expression, the value of the expression is stored in the low-order byte of the word. Expressions greater than 377(8) are truncated to eight bits, and an error code (T) is generated in the assembly listing.

For more information on traps see the PDP-11 Processor Handbook and the applicable system manual (see Section 0.3 in the Preface).

PART III

MACRO-11 DIRECTIVES

CHAPTER 6

GENERAL ASSEMBLER DIRECTIVES

A MACRO-11 directive is placed in the operator field of a source line. Only one directive is allowed per source line. Each directive may have a blank operand field or one or more operands. Legal operands differ with each directive.

General assembler directives are divided into the following categories:

1. Listing control
2. Function control
3. Data storage
4. Radix and numeric control
5. Location counter control
6. Terminators
7. Program boundaries
8. Program sectioning
9. Symbol control
10. Conditional assembly
11. PAL-11R conditional assembly.

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

Table 6-1
Directives in Chapter Six

Directive	Function	Section Reference
.ASCII	Stores delimited string as a sequence of the 8-bit ASCII code of their characters.	6.3.4
.ASCIZ	Same as .ASCII except the string is followed by a zero byte.	6.3.5

(continued on next page)

Table 6-1 (Cont.)
Directives in Chapter Six

Directive	Function	Section Reference
.ASECT	Similar to .PSECT.	6.7.2
.BLKB	Allocates bytes of data storage.	6.5.3
.BLKW	Allocates words of data storage.	6.5.3
.BYTE	Stores successive bytes of data.	6.3.1
.CSECT	Similar to .PSECT.	6.7.2
.DSABL	Disables specified assembler functions.	6.2
.ENABL	Enables specified assembler functions.	6.2
.END	Indicates end of source input.	6.6.1
.ENDC	Indicates end of conditional assembly block.	6.9.1
.EOT	End of tape, ignored under RSX-11M, RSX-11M-PLUS, RT-11, RSTS and IAS systems.	6.6.2
.EVEN	Ensures that current value of the location counter is even.	6.5.1
.FLT2	Generates 2 words of storage for each floating-point number argument.	6.4.2.1
.FLT4	Generates 4 words of storage for each floating-point number argument.	6.4.2.1
.GLOBL	Defines listed symbols as global.	6.8
.IDENT	Provides additional means of labeling an object module.	6.1.5
.IF	Assembles block if specified conditions are met.	6.9.1
.IFF	Assembles block if condition tests false.	6.9.2
.IFT	Assembles block if condition tests true.	6.9.2
.IFTF	Assembles block regardless of whether condition tests true or false.	6.9.2
.IIF	Permits writing a one line conditional assembly block.	6.9.3

(continued on next page)

Table 6-1 (Cont.)
Directives in Chapter Six

Directive	Function	Section Reference
.LIMIT	Allocates two words for storage. At link time puts the lowest address of the load image in one of the saved words and the address of the first free word following the image in the other.	6.5.4
.LIST	Increments listing count or lists certain types of code.	6.1.1
.NLIST	Decrements listing count or suppresses certain types of code.	6.1.1
.ODD	Ensures that the current value of the location counter is odd.	6.5.2
.PACKED	Generates packed decimal data, two digits per byte.	6.3.8
.PAGE	Starts a new listing page.	6.1.6
.PSECT	Declares names for program sections and establishes their attributes.	6.7.1
.RAD50	Generates data in Radix-50 packed format.	6.3.6
.RADIX	Changes radices throughout or in portions of the source program.	6.4.1.1
.RESTORE	Retrieves a previously .SAVEd program section.	6.7.4
.SAVE	Places the current program section on top of the program section context stack.	6.7.3
.SBTTL	Produces a table of contents immediately preceding the assembly listing and puts subheadings on each page in the listing.	6.1.4
.TITLE	Assigns a name to the object module and puts headings on each page of the assembly listing.	6.1.3
.WORD	Generates successive words of data in the object module.	6.3.2

6.1 LISTING CONTROL DIRECTIVES

Listing control directives control the content, format, and pagination of all line printer (see Figure 6-1) and teleprinter (see Figure 6-2) assembly listing output. On the first line of each page, MACRO-11 prints the following (from left to right):

1. Title of the object module, as established through the .TITLE directive (see Section 6.1.2).
2. Assembler version identification.
3. Date.
4. Time of day.
5. Page number.

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

In the line printer format (Figure 6-1) binary extensions for statements generating more than one word are listed horizontally.

In the teleprinter format (Figure 6-2) binary extensions for statements generating more than one word are listed vertically. There is no explicit truncation of output to 80 characters by the assembler.

```
 1                                      ;+
 2                                      ; GETSYM
 3                                      ; Scan off a RAD50 symbol. Leave with scan pointer set at next non-blank
 4                                      ; char past end of symbol. Symbol buffer clear and Z set if no symbol
 5                                      ; seen; In this case scan pointer is unaltered.
 6                                      ;-
 7
 8 000126  010146                       GETSYM::MOV     R1,-(SP)        ;Save work register
 9 000130  016767  000000G 000000G              MOV     CHRPNT,SYMBEG   ;Save scan pointer in case of rescan
10 000136  012701  000004G              MOV     #SYMBOL+4,R1    ;Point at end of symbol buffer
11 000142  005041                       CLR     -(R1)           ;Now clear it
12 000144  005041                       CLR     -(R1)
13 000146  136527  000000' 000040      BITB    CTTBL(R5),#CT.ALP ;Test first char for alphabetic
14 000154  001436                       BEQ     4$              ;Exit if not with Z set
15 000156  116500  000262'          1$:  MOVB    CTTBL2(R5),R0   ;Map to RAD50
16 000162  003431                       BLE     3$              ;Exit if not
17 000164  006300                       ASL     R0              ;Make word index
18 000166  016011  000462'              MOV     R50TB1(R0),(R1) ;Load the high char
19 000172                               GETCHR                  ;Get another char
20 000176  116500  000262'              MOVB    CTTBL2(R5),R0   ;Handle it as above
21 000202  003421                       BLE     3$
22 000204  006300                       ASL     R0
23 000206  066011  000602'              ADD     R50TB2(R0),(R1)
24 000212                               GETCHR                  ;Now get low order char
25 000216  116500  000262'              MOVB    CTTBL2(R5),R0   ;Map and test it
26 000222  003411                       BLE     3$
27 000224  060021                       ADD     R0,(R1)+        ;Just add in the low char, advance pointer
28 000226                          2$:  GETCHR                  ;Get following char
29 000232  020127  000004G              CMP     R1,#SYMBOL+4    ;Test if at end of symbol buffer
30 000236  001347                       BNE     1$              ;Go again if no
31 000240  105765  000262'              TSTB    CTTBL2(R5)      ;Flush to end of symbol if yes
32 000244  003370                       BGT     2$
33 000246                          3$:  SETNB                   ;Now scan to a non blank char
34 000252  012601                  4$:  MOV     (SP)+,R1        ;Restore work register
35 000254  016700  000000G              MOV     SYMBOL,R0       ;Set Z if no symbol found
36 000260  000207                       RETURN                  ;Exit
37
```

Figure 6-1 Example of Line Printer Assembly Listing

```
38                                      ;+
39                                      ; Table CTTBL2
40                                      ; Index with 7 bit ASCII value to get corresponding RAD50 value
41                                      ; If EQ 0 then space, if LT 0 then not RAD50; Other bits reserved.
42                                      ;-
43
44                                              .NLIST  BEX
45 000262      200     200     200     CTTBL2: .BYTE   200,200,200,200,200,200,200,200 ;
46 000272      200     200     200             .BYTE   200,200,200,200,200,200,200,200 ;
47 000302      200     200     200             .BYTE   200,200,200,200,200,200,200,200 ;
48 000312      200     200     200             .BYTE   200,200,200,200,200,200,200,200 ;
49 000322      000     200     200             .BYTE   000,200,200,200,033,200,200,200 ;    $
50 000332      200     200     200             .BYTE   200,200,200,200,200,200,034,200 ;      .
51 000342      036     037     040             .BYTE   036,037,040,041,042,043,044,045 ;01234567
52 000352      046     047     200             .BYTE   046,047,200,200,200,200,200,200 ;89
53 000362      200     001     002             .BYTE   200,001,002,003,004,005,006,007 ; ABCDEFG
54 000372      010     011     012             .BYTE   010,011,012,013,014,015,016,017 ;HIJKLMNO
55 000402      020     021     022             .BYTE   020,021,022,023,024,025,026,027 ;PQRSTUVW
56 000412      030     031     032             .BYTE   030,031,032,200,200,200,200,200 ;XYZ
57 000422      200     001     002             .BYTE   200,001,002,003,004,005,006,007 ; abcdefs
```

Figure 6-1 (Cont.) Example of Line Printer Assembly Listing

```
 1                      ;+
 2                      ; GETSYM
 3                      ; Scan off a RAD50 symbol. Leave with scan pointer set at next non-blank
 4                      ; char past end of symbol. Symbol buffer clear and Z set if no symbol
 5                      ; seen; In this case scan pointer is unaltered.
 6                      ;-
 7
 8 000126  010146  GETSYM::MOV     R1,-(SP)          ;Save work register
 9 000130  016767          MOV     CHRPNT,SYMBEG     ;Save scan pointer in case of rescan
           000000G
           000000G
10 000136  012701          MOV     #SYMBOL+4,R1      ;Point at end of symbol buffer
           000004G
11 000142  005041          CLR     -(R1)             ;Now clear it
12 000144  005041          CLR     -(R1)
13 000146  136527          BITB    CTTBL(R5),#CT.ALP ;Test first char for alphabetic
           000000'
           000040
14 000154  001436          BEQ     4$                ;Exit if not with Z set
15 000156  116500  1$:     MOVB    CTTBL2(R5),R0     ;Map to RAD50
           000262'
16 000162  003431          BLE     3$                ;Exit if not
17 000164  006300          ASL     R0                ;Make word index
18 000166  016011          MOV     R50TB1(R0),(R1)   ;Load the high char
           000462'
19 000172                  GETCHR                    ;Get another char
20 000176  116500          MOVB    CTTBL2(R5),R0     ;Handle it as above
           000262'
21 000202  003421          BLE     3$
22 000204  006300          ASL     R0
23 000206  066011          ADD     R50TB2(R0),(R1)
           000602'
```

Figure 6-2 Example of Teleprinter Assembly Listing

```
24 000212                       GETCHR                          ;Now get low order char
25 000216  116500               MOVB    CTTBL2(R5),R0           ;Map and test it
           000262'
26 000222  003411               BLE     3$
27 000224  060021               ADD     R0,(R1)+                ;Just add in the low char, advance pointer
28 000226          2$:          GETCHR                          ;Get following char
29 000232  020127               CMP     R1,$SYMBOL+4            ;Test if at end of symbol buffer
           000004G
30 000236  001347               BNE     1$                      ;Go again if no
31 000240  105765               TSTB    CTTBL2(R5)              ;Flush to end of symbol if yes
           000262'
32 000244  003370               BGT     2$
33 000246          3$:          SETNB                           ;Now scan to a non blank char
34 000252  012601  4$:          MOV     (SP)+,R1                ;Restore work register
35 000254  016700               MOV     SYMBOL,R0               ;Set Z if no symbol found
           000000G
36 000260  000207               RETURN                          ;Exit
37
38                      ;+
39                      ; Table CTTBL2
40                      ; Index with 7 bit ASCII value to get corresponding RAD50 value
41                      ; If EQ 0 then space, if LT 0 then not RAD50; Other bits reserved.
42                      ;-
43
44                              .NLIST  BEX
```

Figure 6-2 (Cont.) Example of Teleprinter Assembly Listing

6.1.1 .LIST and .NLIST Directives

Formats:

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

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

As indicated above, the listing control directives may be used without arguments, in which case the listing directives alter the listing level count. The listing level count is initialized to zero. At each occurrence of a .LIST directive, the listing level count is incremented; at each occurrence of an .NLIST directive, the listing level count is decremented. When the level count is negative, the listing is suppressed (unless the line contains an error). Conversely, when the level count is greater than zero, the listing is generated regardless of the context of the line. Finally, when the count is zero, the line is either listed or suppressed, depending on the listing controls currently in effect for the program. The following macro definition employs the .LIST and .NLIST directives to selectively list portions of the macro body when the macro is expanded:

```
        .MACRO  LTEST           ;LIST TEST
; A-THIS LINE SHOULD LIST       ;LISTING LEVEL COUNT IS 0.
        .NLIST                  ;LISTING LEVEL COUNT IS -1.
; B-THIS LINE SHOULD NOT LIST
        .NLIST                  ;LISTING LEVEL COUNT IS -2.
; C-THIS LINE SHOULD NOT LIST
        .LIST                   ;LISTING LEVEL COUNT IS -1.
; D-THIS LINE SHOULD NOT LIST
        .LIST                   ;LISTING LEVEL COUNT IS 0.
; E-THIS LINE SHOULD LIST       ;LISTING LEVEL COUNT IS 0.
; F-THIS LINE SHOULD LIST       ;LISTING LEVEL COUNT IS 0.
; G-THIS LINE SHOULD LIST       ;LISTING LEVEL COUNT IS 0.
        .ENDM
          .
          .
          .
        .LIST   ME              ;LIST MACRO EXPANSION.
        LTEST                   ;CALL THE MACRO
; A-THIS LINE SHOULD LIST       ;LISTING LEVEL COUNT IS 0.
; E-THIS LINE SHOULD LIST       ;LISTING LEVEL COUNT IS 0.
; F-THIS LINE SHOULD LIST       ;LISTING LEVEL COUNT IS 0.
; G-THIS LINE SHOULD LIST       ;LISTING LEVEL COUNT IS 0.
```

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

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

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

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

        .NLIST  ME              ;DO NOT LIST MACRO EXPANSIONS.
        XX
X=.
```

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

Table 6-2
Symbolic Arguments of Listing Control Directives

Argument	Default	Function
SEQ*	List	Controls the listing of the sequential numbers assigned to the source lines. If this number field is suppressed through an .NLIST SEQ directive, MACRO-11 generates a tab, effectively allocating blank space for the field. Thus, the positional relationships of the other fields in the listing remain undisturbed. During the assembly process, MACRO-11 examines each source line for possible error conditions. For any line in error, the error code is printed preceding the number field. MACRO-11 does not assign line numbers to files that have had such numbers assigned by other programs (an editor program, for instance).

(continued on next page)

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

Table 6-2 (Cont.)
Symbolic Arguments of Listing Control Directives

Argument	Default	Function
LOC*	List	Controls the listing of the current location counter field. Normally, this field is not suppressed. However, if it is suppressed through the .NLIST LOC directive, MACRO-11 does not generate a tab, nor does it allocate space for the field, as is the case with the SEQ field described above. Thus, the suppression of the current location counter (LOC) field effectively left-justifies all subsequent fields (while preserving positional relationships) to the position normally occupied by the counter's field.
BIN*	List	Controls the listing of generated binary code. If this field is suppressed through an .NLIST BIN directive, left-justification of the source code field occurs in the same manner described above for the LOC field.
BEX	List	Controls the listing of binary extensions (the locations and binary contents beyond those that will fit on the source statement line). This is a subset of the BIN argument.
SRC*	List	Controls the listing of source lines.
COM	List	Controls the listing of comments. This is a subset of the SRC argument. The .NLIST COM directive reduces listing time and space when comments are not desired.
MD	List	Controls the listing of macro definitions and repeat range expansions.
MC	List	Controls the listing of macro calls and repeat range expansions.
ME	No list	Controls the listing of macro expansions.

(continued on next page)

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

Table 6-2 (Cont.)
Symbolic Arguments of Listing Control Directives

Argument	Default	Function
MEB	No list	Controls the listing of macro expansion binary code. A .LIST MEB directive lists only those macro expansion statements that generate binary code. This is a subset of the ME argument.
CND	List	Controls the listing of unsatisfied conditional coding and associated .IF and .ENDC directives in the source program. A .NLIST CND directive lists only satisfied conditional coding.
LD	No list	Controls the listing of all listing directives having no arguments, in other words, the directives that alter the listing level count.
TOC	List	Controls the listing of the table of contents during assembly pass 1 (see Section 6.1.3 describing the .SBTTL directive). This argument does not affect the printing of the full assembly listing during assembly pass 2.
SYM	List	Controls the listing of the symbol table resulting from the assembly of the source program.
TTM**	List	Controls the listing output format. The default can be set by the system manager. If the system manager does not set a default, it is set to line printer format. Figure 6-1 illustrates the line printer output format. Figure 6-2 illustrates the teleprinter output format.

** The default for RSX-11M and RT-11 programs is no list.

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

The listing control options can also be specified at assembly time through switches included in the command string to MACRO-11 (see Section 8.1.3 and/or the appropriate system manual). The use of these switches overrides all corresponding listing control (.LIST or .NLIST) directives specified in the source program.

Figure 6-3 shows a listing, produced in line printer format, reflecting the use of the .LIST and .NLIST directives in the source program and the effects such directives have on the assembly listing output.

```
LISTING CONTROL EXAMPLE MACRO Y04.00  24-OCT-79 13:33:56 PAGE 1

 1                                          .TITLE  LISTING CONTROL EXAMPLE
 2
 3                                          .LIST   ME              ;List macro expansions
 4
 5                                  ;+
 6                                  ; Listing control test macro
 7                                  ;-
 8
 9                                          .MACRO  LSTMAC  ARG
10                                          .NLIST  ARG
11                                          .WORD   1,2,3,4         ;This is a test comment
12                                          .LIST   ARG
13                                          .ENDM
14
15 000000                                  LSTMAC  LOC             ;Location counter test
                                    .NLIST  LOC
   000001  000002  000003           .WORD   1,2,3,4         ;This is a test comment
   000004
                                           .LIST   LOC
16
17 000010                                  LSTMAC  BIN             ;Generated binary test
                    .NLIST  BIN
   000010           .WORD   1,2,3,4        ;This is a test comment
                                           .LIST   BIN
18
19 000020                                  LSTMAC  BEX             ;Binary extensions test
                                           .NLIST  BEX
   000020  000001  000002  000003         .WORD   1,2,3,4         ;This is a test comment
                                           .LIST   BEX
20
21 000030                                  LSTMAC  SRC             ;Source lines test

   000030  000001  000002  000003
   000036  000004
```

Figure 6-3 Listing Produced with Listing Control Directives

```
                                        .LIST   SRC
22
23 000040                               LSTMAC  COM             ;Comment lines test
                                        .NLIST  COM
   000040  000001  000002  000003       .WORD   1,2,3,4
   000046  000004
                                        .LIST   COM
24
25 000050                               LSTMAC  <COM,BEX>       ;Comment lines and extended binary test
                                        .NLIST  COM,BEX
   000050  000001  000002  000003       .WORD   1,2,3,4
                                        .LIST   COM,BEX
26
27                      .LIST   TTM             ;Enable narrow listing
28
29 000060               LSTMAC  SEQ             ;Sequence numbers test
                        .NLIST  SEQ
   000060  000001       .WORD   1,2,3,4         ;This is a test comment
   000062  000002
   000064  000003
   000066  000004
                        .LIST   SEQ
30
31 000070               LSTMAC  BEX             ;Binary extensions test
                        .NLIST  BEX
   000070  000001       .WORD   1,2,3,4         ;This is a test comment
                        .LIST   BEX
32
33         000001       .END
```

Figure 6-3 (Cont.) Listing Produced with Listing Control Directives

6.1.2 .TITLE Directive

.TITLE

Format:

```
.TITLE string
```

where: string represents an identifier of 1 or more Radix-50 characters. Appendix A.2 contains a table of Radix-50 characters.

The .TITLE directive assigns a name to the object module. The name assigned is the first six non-blank, Radix-50 characters following the .TITLE directive. All spaces and/or tabs up to the first non-space/non-tab character following the .TITLE directive are ignored by MACRO-11 when evaluating the text string. Any characters beyond the first six are checked for ASCII legality, but they are not used as part of the object module name. For example, the directive:

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

causes the assembled object module to be named PROGRA. This 6-character name bears no relationship to the filename of the object module, as specified in the command string to MACRO-11. The name of an object module (specified in the .TITLE directive) appears in the load map produced at link time. This is also the module name which the Librarian will recognize.

If the .TITLE directive is not specified, MACRO-11 assigns the default name .MAIN. to the object module. If more than one .TITLE directive is specified in the source program, the last .TITLE directive encountered during assembly pass 1 establishes the name for the entire object module.

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

6.1.3 .SBTTL Directive

.SBTTL

Format:

```
.SBTTL string
```

where: string represents an identifier of 1 or more Radix-50 characters. Appendix A.2 contains a table of Radix-50 characters.

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

```
.SBTTL  CONDITIONAL ASSEMBLIES
```

causes the text

```
CONDITIONAL ASSEMBLIES
```

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

During assembly pass 1, a table of contents containing the line sequence number, the page number, and the text accompanying each .SBTTL directive is printed for the assembly listing. The listing of the table of contents is suppressed whenever an .NLIST TOC directive is encountered in the source program (see Table 6-2). An example of a table of contents listing is shown in Figure 6-4.

```
MTTEMT - RT-11 MULTI-TTY EMT SE MACRO Y04.00  10-OCT-79 15:00:26
TABLE OF CONTENTS

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

Figure 6-4 Assembly Listing Table of Contents

NOTE

When using the .SBTTL directive it is not a good idea to use the switch that assembles only pass 1. During assembly pass 1 the pages of the listing are numbered and a table of contents listed. After assembly pass 2, if the switch is used on one or more of the files, the table of contents will be incorrect because a complete re-numbering of the listing is not possible.

6.1.4 .IDENT Directive

.IDENT

Format:

```
.IDENT  /string/
```

where: string represents six or fewer Radix-50 characters which establish the program identification or version number. This number is included in the global symbol directory of the object module; the first four characters are printed in the load map and librarian listing.

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

In addition to the name assigned to the object module with the .TITLE directive (see Section 6.1.3), the .IDENT directive allows the user to label the object module with the program version number.

An example of the .IDENT directive is shown below:

```
.IDENT  /V05A/
```

The character string V05A is converted to Radix-50 representation and included in the global symbol directory of the object module. This character string also appears in the load map produced at link time and the Librarian directory listings.

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

RT-11 allows only one .IDENT string in a program. The linker uses the first .IDENT directive encountered during the first pass to establish the character string that will be identified with all of the object modules.

RSX-11M allows an .IDENT string for each module in the program. The Task Builder uses the first .IDENT directive in each module to establish the character string that will be identified with that module. Like the RT-11 Linker, the RSX-11M Task Builder uses the .IDENT directives encountered on the first pass.

6.1.5 .PAGE Directive/Page Ejection

.PAGE

Format:

```
.PAGE
```

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

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

Page ejection is accomplished in three other ways:

1. After reaching a count of 58 lines in the listing, MACRO-11 automatically performs a page eject to skip over page perforations on line printer paper and to formulate teleprinter output into pages. The page number is not changed.

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

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

6.2 FUNCTION DIRECTIVES: .ENABL AND .DSABL

.ENABL
.DSABL

Formats:

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

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

These directives are included in a source program to invoke or inhibit certain MACRO-11 functions and operations incidental to the assembly process itself. Specifying any argument in an .ENABL/.DSABL directive other than those listed in Table 6-3 causes that directive to be flagged with an error code (A) in the assembly listing.

Table 6-3
Symbolic Arguments of Function Control Directives

Argument	Default	Function
ABS	Disable	Enabling this function produces absolute binary output in FILES-11 format. To convert this output to Formatted Binary format (as required by the Absolute Loader), use the FLX utility.
AMA	Disable	Enabling this function causes all relative addresses (address mode 67) to be assembled as absolute addresses (address mode 37). This function is useful during the debugging phase of program development.
CDR	Disable	Enabling this function causes source columns from 73 to the end of the line, to be treated as a comment. The most common use of this feature is to permit sequence numbers in card columns 73-80.
CRF	Enable	Disabling this function inhibits the generation of cross-reference output. This function only has meaning if cross-reference output generation is specified in the command string.
FPT	Disable	Enabling this function causes floating-point truncation; disabling this function causes floating-point rounding.
LC	Disable	Enabling this function causes MACRO-11 to accept lower-case ASCII input instead of converting it to upper-case. If this function is not enabled, all text is converted to upper-case.
LSB	Disable	This argument permits the enabling or disabling of a local symbol block. Although a local symbol block is normally established by encountering a new symbolic label, a .PSECT directive or a .RESTORE directive in the source program, an .ENABL LSB directive establishes a new local symbol block which is not terminated until (1) another .ENABL LSB is encountered, or (2) another symbolic label, .PSECT directive or .RESTORE directive is encountered following a paired .DSABL LSB directive. The basic function of this directive with regard to .PSECTs is limited to those instances where it is desirable to leave a program section temporarily to store data, followed by a return to the original program section. This temporary dismissal of the current program section may also be accomplished through the .SAVE and .RESTORE directives (see Sections 6.7.3 and 6.7.4).

(continued on next page)

Table 6-3 (Cont.)
Symbolic Arguments of Function Control Directives

Argument	Default	Function
LSB (cont.)	Disable	Attempts to define local symbols in an alternate program section are flagged with an error code (P) in the assembly listing. An example of the .ENABL LSB and .DSABL LSB directives, as typically used in a source program, is shown in Figure 6-5.
PNC	Enable	Disabling this function inhibits binary output until an .ENABL PNC statement is encountered within the same module.
REG	Enable	When specified, the .DSABL REG directive inhibits the normal MACRO-11 default register definitions; if not disabled, the default definitions listed below remain in effect. R0=%0 R1=%1 R2=%2 R3=%3 R4=%4 R5=%5 SP=%6 PC=%7 The .ENABL REG statement may be used as the logical complement of the .DSABL REG directive. The use of these directives, however, is not recommended. For logical consistency, use the normal default register definitions listed above.
GBL*	Enable	When the .ENABL GBL directive is specified, MACRO-11 treats all symbol references that are undefined at the end of assembly pass 1 as default global references; when the .DSABL GBL directive is specified, MACRO-11 treats all such references as undefined symbols. In assembly pass 2, if the .DSABL GBL function is still in effect, these undefined symbols are flagged with an error code (U) in the assembly listing; otherwise, they continue to be regarded by MACRO-11 as global references.

* The default is Disable for RT-11 MACRO programs.

```
.ENABL/.DSABL   MACRO Y04.00  24-OCT-79 18:26:26 PAGE 1

     1                                          .TITLE  .ENABL/.DSABL
     2
     3                                  ;+
     4                                  ; ILLUSTRATE .ENABL/.DSABL LC
     5                                  ;-
     6
     7                                          .ENABL  LC      ;STORE MACRO IN LOWER CASE
     8
     9                                  .MACRO  TEXT    $$$
    10                                          .ASCII  /This $$$ a lower case string/
    11                                  .ENDM
    12
    13                                          .LIST   ME
    14                                          .NLIST  BEX
    15
    16 000000                                   TEXT    is      ;Invoke macro in lower case
       000000     124     150     151          .ASCII  /This is a lower case string/
    17
    18                                          .DSABL  LC      ;Now disable lower case
    19
    20 000033                                   TEXT    WAS     ;RE-INVOKE MACRO IN UPPER CASE
       000033     124     110     111          .ASCII  /THIS WAS A LOWER CASE STRING/
    21
    22          000001                          .END
```

Figure 6-5 Example of .ENABL and .DSABL Directives

6.3 DATA STORAGE DIRECTIVES

A wide range of data and data types can be generated with the directives, ASCII conversion characters, and radix-control operators described in the following sections.

6.3.1 .BYTE Directive

.BYTE

Format:

```
.BYTE   exp             ;STORES THE BINARY VALUE OF THE
                        ;EXPRESSION "EXP" IN THE NEXT BYTE.

.BYTE   expl,exp2,expn  ;STORES THE BINARY VALUES OF THE LIST
                        ;OF EXPRESSIONS IN SUCCESSIVE BYTES.
```

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

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

Example:

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

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

At link time, it is likely that a relocatable expression will result in a value having more than eight bits, in which case the task builder or linker issues a truncation (T) error for the object module in question. For example, the following statements create such a possibility:

```
        .BYTE   23              ;STORES OCTAL 23 IN NEXT BYTE.
A:
        .BYTE   A               ;RELOCATABLE VALUE A WILL PROBABLY
                                ;CAUSE TRUNCATION ERROR.
```

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

```
.=420
        .BYTE   ,,,             ;ZEROS ARE STORED IN BYTES 420, 421,
                                ;422, AND 423.
```

Note that in the above example, four bytes of storage result from the .BYTE directive. The three commas in the operand field represent an implicit declaration of four null values, each separated from the other by a comma. Hence, four bytes, each containing a value of zero (0), are reserved in the object module.

6.3.2 .WORD Directive

.WORD

Formats:

```
     .WORD   exp             ;STORES THE BINARY EQUIVALENT OF THE
                             ;EXPRESSION EXP IN THE NEXT WORD.

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

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

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

Example:

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

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

```
.=500
        .WORD   ,5,             ;STORES THE VALUES 0, 5, AND 0 IN
                                ;LOCATION 500, 502, AND 504,
                                ;RESPECTIVELY.
```

A statement with a blank operator field (one that contains a symbol other than a macro call, an instruction mnemonic, a MACRO-11 directive, or a semicolon) is interpreted during assembly as an implicit .WORD directive, as shown in the example below:

```
.=440
LABEL:  100,LABEL               ;STORES THE VALUE 100 IN LOCATION 440
                                ;AND THE VALUE 440 IN LOCATION 442.
```

CAUTION

You should not use this technique to generate .WORD directives because it may not be included in future PDP-11 assemblers.

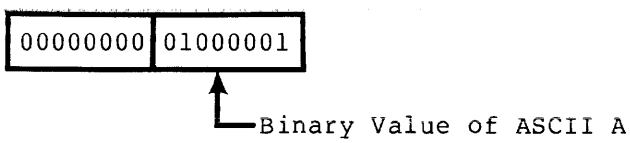
6.3.3 ASCII Conversion Characters

The single quote (') and the double quote (") characters are unary operators that can appear in any MACRO-11 expression. Used in MACRO-11 expressions, these characters cause a 16-bit expression value to be generated.

When the single quote is used, MACRO-11 takes the next character in the expression and converts it from its 7-bit ASCII value to a 16-bit expression value. The high-order byte of the resulting expression value is always zero (0). The 16-bit value is then used as an absolute term within the expression. For example, the statement:

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

moves the following 16-bit expression value into register 0:



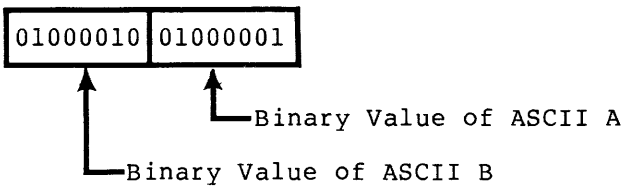
Thus the expression 'A results in a value of 101(8).

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

When the double quote is used, MACRO-11 takes the next two characters in the expression and converts them to a 16-bit binary expression value from their 7-bit ASCII values. This 16-bit value is then used as an absolute term within the expression. For example, the statement:

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

moves the following 16-bit expression value into register 0:



Thus the expression "AB results in a value of 041101(8).

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

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

6.3.4 .ASCII Directive

.ASCII

Format:

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

where: string is a string of printable ASCII characters. The vertical-tab, null, line-feed, RUBOUT, and all other non-printable ASCII characters, except carriage-return and form-feed, cause an error code (I) if used in an .ASCII string. The carriage-return and form-feed characters are flagged with an error code (A) because these characters end the scan of the line, preventing MACRO-11 from detecting the matching delimiter at the end of the character string.

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

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

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

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

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

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

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

```
.ASCII  /HELLO/             ;STORES THE BINARY REPRESENTATION
                            ;OF THE LETTERS HELLO IN FIVE
                            ;CONSECUTIVE BYTES.

.ASCII  /ABC/<15><12>/DEF/  ;STORES THE BINARY REPRESENTATION
                            ;OF THE CHARACTERS A,B,C,CARRIAGE
                            ;RETURN,LINE FEED,D,E,F IN EIGHT
                            ;CONSECUTIVE BYTES.

.ASCII  /A<15>B/            ;STORES THE BINARY REPRESENTATION
                            ;OF THE CHARACTERS A, <, 1, 5, >,
                            ;AND B IN SIX CONSECUTIVE BYTES.
```

NOTE

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

```
.ASCII  ;ABC;/DEF/    ;STORES THE BINARY
                      ;REPRESENTATION OF
                      ;THE  CHARACTERS
                      ;A, B, C, D, E, AND
                      ;F IN SIX
                      ;CONSECUTIVE BYTES;
                      ;NOT RECOMMENDED
                      ;PRACTICE.

.ASCII  /ABC/;DEF;    ;STORES THE BINARY
                      ;REPRESENTATIONS OF
                      ;THE CHARACTERS A,
                      ;B, AND C IN THREE
                      ;CONSECUTIVE BYTES;
                      ;THE CHARACTERS D,
                      ;E, F, AND ; ARE
                      ;TREATED AS A
                      ;COMMENT.
```

```
.ASCII  /ABC/=DEF=      ;STORES THE BINARY
                        ;REPRESENTATION OF
                        ;THE CHARACTERS A,
                        ;B, C, D, E, AND
                        ;F IN SIX
                        ;CONSECUTIVE BYTES;
                        ;NOT RECOMMENDED
                        ;PRACTICE.
```

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

```
.ASCII  =DEF=           ;THE DIRECT
                        ;ASSIGNMENT
                        ;OPERATION
                        ;.ASCII=DEF IS
                        ;PERFORMED, AND A
                        ;SYNTAX ERROR (Q)
                        ;IS GENERATED UPON
                        ;ENCOUNTERING THE
                        ;SECOND = SIGN.
```

6.3.5 .ASCIZ Directive

.ASCIZ

Format:

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

where: string is a string of printable ASCII characters. The vertical-tab, null, line-feed, RUBOUT, and all other non-printable ASCII characters, except carriage-return and form-feed, cause an error code (I) if used in an .ASCIZ string. The carriage-return and form-feed characters are flagged with an error code (A) because they end the scan of the line, preventing MACRO-11 from detecting the matching delimiter.

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

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

```
CR=15
LF=12
HELLO:  .ASCIZ  <CR><LF>/MACRO-11 V01A/<CR><LF>  ;INTRODUCTORY MESSAGE
        .EVEN
        .
        .
        .
        MOV     #HELLO,R1       ;GET ADDRESS OF MESSAGE.
        MOV     #LINBUF,R2      ;GET ADDRESS OF OUTPUT BUFFER.
10$:    MOVB    (R1)+,(R2)+     ;MOVE A BYTE TO OUTPUT BUFFER.
        BNE     10$             ;IF NOT NULL, MOVE ANOTHER BYTE.
        .
        .
        .
```

6.3.6 .RAD50 Directive

.RAD50

Format:

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

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

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

As with the .ASCII directive (described in Section 6.3.4), the vertical-tab, null, line-feed, RUBOUT, and all other non-printing characters, except carriage-return and form-feed, cause an error code (I) if used in a .RAD50 string. The carriage-return and form-feed characters result in an error code (A) because these characters end the scan of the line, preventing MACRO-11 from detecting the matching delimiter.

/ / represent delimiting characters. These delimiters may be any paired printing characters, other than the equal sign (=), the left angle bracket (<), or the semicolon (;) (see Note in Section 6.3.4), provided that the delimiting character is not contained within the text string itself. If the delimiting characters do not match or if an illegal delimiting character is used, the .RAD50 directive is flagged with an error code (A) in the assembly listing.

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

```
.RAD50  /ABC/           ;PACKS ABC INTO ONE WORD.
.RAD50  /AB/            ;PACKS AB (SPACE) INTO ONE WORD.
.RAD50  /ABCD/          ;PACKS ABC INTO FIRST WORD AND
                        ;D (SPACE) (SPACE) INTO SECOND WORD.
.RAD50  /ABCDEF/        ;PACKS ABC INTO FIRST WORD, DEF INTO
                        ;SECOND WORD.
```

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

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

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

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

For example:

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

Refer to Appendix A.2 for a table of Radix-50 equivalents.

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

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

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

6.3.7 Temporary Radix-50 Control Operator

Format:

```
^Rccc
```

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

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

```
MOV  #^RMAC,FILEXT        ;STORE RAD50 MAC AS FILE EXTENSION
```

The number sign (#) is used to indicate immediate data (data to be assembled directly into object code). ^R specifies that the characters MAC are to be converted to Radix-50. This value is then stored in location FILEXT.

6.3.8 .PACKED Directive

.PACKED

Format:

```
.PACKED decimal-string[,symbol]
```

where: decimal-string — represents a decimal number from 0 to 31(10) digits long. Each digit must be in the range 0 to 9. The number may have a sign, but it is not required and is not counted as a digit.

symbol — is assigned a value equivalent to the number of decimal digits in the string.

The .PACKED directive generates packed decimal data, 2 digits per byte. Arithmetic and operational properties of packed decimals are similar to those of numeric strings. Below is an example of the .PACKED directive:

```
        .PACKED -12,PACK          ;PACK GETS VALUE OF 2
        .PACKED +500              ;500 IS PACKED
        .PACKED 0                 ;0 IS PACKED
        .PACKED -0,SUM            ;SUM GETS VALUE OF 1
        .PACKED 1234E6            ;ILLEGAL PACKED DECIMAL NUMBER
                                  ;E6 WILL BE TREATED AS A VARIABLE
                                  ;AND GIVEN A VALUE OF 4
```

6.4 RADIX AND NUMERIC CONTROL FACILITIES

6.4.1 Radix Control and Unary Control Operators

Any numeric or expression value in a MACRO-11 source program is read as an octal value by default. Occasionally, however, an alternate radix would be useful. By using the MACRO-11 facilities described below, a programmer may declare a radix to affect a term or an entire program depending on his needs.

NOTE

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

6.4.1.1 .RADIX Directive

.RADIX

Format:

```
.RADIX  n
```

where: n represents one of the three radices: 2, 8 or 10. Any value other than null or one of the three acceptable radices will cause an error code (A) in the assembly listing. If the argument n is not specified, the octal default radix is assumed. The argument (n) is always read as a decimal value.

Numbers used in a MACRO-11 source program are initially considered to be octal values; however, with the .RADIX directive you can declare alternate radices applicable throughout the source program or within specific portions of the program.

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

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

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

6.4.1.2 Temporary Radix Control Operators

Formats:

```
^D"number"  ("number" is evaluated as a decimal number)
^O"number"  ("number" is evaluated as an octal number)
^B"number"  ("number" is evaluated as a binary number)
```

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

Thus an alternate radix can be declared temporarily to meet a localized requirement in the source program. The temporary radix control operator may be used any time regardless of the radix in effect or other radix declarations within the program. Because the

operator affects only the term immediately following it, it may be used anywhere a numeric value is legal. The term (or expression) associated with the temporary radix control operator will be evaluated during assembly as a 16-bit entity.

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

```
^D123          Decimal radix
^O 47          Octal Radix
^B 00001101    Binary Radix
^O<A+13>       Octal Radix
```

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

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

```
        .RADIX  10
A=10
        .WORD   ^O<A+10>*10
```

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

```
        .WORD   180.
```

MACRO-11 also allows a temporary radix change to decimal by specifying a number, immediately followed by a decimal point (.), as shown below:

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

The above expression forms are equivalent in function to:

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

6.4.2 Numeric Directives and Unary Control Operators

Two storage directives and two numeric control operators are available to simplify the use of the floating-point hardware on the PDP-11. These facilities allow floating-point data to be created in the program, and numeric values to be complemented or treated as floating-point numbers.

A floating-point number is represented by a string of decimal digits. The string (which can be a single digit in length) may contain an optional decimal point and may be followed by an optional exponent indicator in the form of the letter E and a signed decimal integer exponent. The number may not contain embedded blanks, tabs or angle brackets and may not be an expression. Such a string will result in one or more errors (A and/or Q) in the assembly listing.

The list of numeric 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 inferred, the list could be extended indefinitely (3000E-3, .03E2, etc.). A leading plus sign is optional (3.0 is considered to be +3.0). A leading minus sign complements the sign bit. No other operators are allowed (for example, 3.0+N is illegal).

All floating-point numbers are evaluated as 64 bits in the following format:

```
63    62      55   54          0

S     EEEEEEEE    MMM.....MMM

                              Mantissa    (55 bits)
                              Exponent    (8 bits)
                              Sign        (1 bit)
```

MACRO-11 returns a value of the appropriate size and precision via one of the floating-point directives. The values returned may be truncated or rounded (see Section 6.2).

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 bit of the unretained word is added to the low-order bit of the retained word, as shown below. For example, if the number is to be stored in a 2-word field, but more than 32 bits are needed to express its exact value, the highest bit (32) of the unretained field is added to the least significant bit (0) of the retained field (see illustration below). The .ENABL FPT directive is used to enable floating-point truncation; .DSABL FPT is used to return to floating-point rounding (see Table 6-3).

```
Bit          Bit  Bit          Bit
32             0  32             0
 \_____________/    \_____________/
      Retained           Unretained
      field              field
```

lps69,70 All numeric operands associated with Floating Point Processor instructions are automatically evaluated as single-word, decimal, floating-point values unless a temporary radix control operator is specified. For example, to add (floating) the octal constant 41040 to the contents of floating accumulator zero, the following instruction must be used:

```
ADDF    #^O41040,F0
```

where: F0 is assumed to represent floating accumulator zero.

Floating-point numbers are described in greater detail in the applicable PDP-11 Processor Handbook.

.FLT2
.FLT4

6.4.2.1 Floating-Point Storage Directives

Formats:

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

where: arg1,arg2,... represent one or more floating point numbers as described in Section 6.4.2. Multiple arguments must be separated by commas.

.FLT2 causes two words of storage to be generated for each argument, while .FLT4 generates four words of storage for each argument. As in the .WORD directive, the arguments are evaluated and the results stored in the object module.

6.4.2.2 **Temporary Numeric Control Operators: ^C and ^F** - The ^C unary operator allows you to specify an argument that is to be complemented as it is evaluated during assembly. The ^F unary operator allows you to specify an argument that is a 1-word floating-point number.

As with the radix control operators described above, the numeric control operator (^C) can be used anywhere in the source program that an expression value is legal. Such a construction is evaluated by MACRO-11 as a 16-bit binary value before being complemented. For example, the following statement:

```
TAG4:   .WORD   ^C151
```

causes the 1's complement of the value 151 (octal) to be stored as a 16-bit value in the program. The resulting value expressed in octal form is 177626(8).

Because the ^C construction is a unary operator, the operator and its argument are regarded as a term. Thus, more than one unary operator may be applied to a single term. For example, the following construction:

```
^C^D25
```

causes the decimal value 25 to be complemented during assembly. The resulting binary value, when expressed in octal form, reduces to 177746(octal).

The term created through the use of the temporary numeric control operator can be used alone or in combination with other expression elements. For example, the following construction:

```
^C2+6
```

is equivalent in function to:

```
<^C2>+6
```

This expression is evaluated during assembly as the 1's complement of 2, plus the absolute value of 6. When these terms are combined, the resulting expression value generates a carry beyond the most significant bit, leaving 000003(8) as the reduced value.

As shown above, when the temporary numeric control operator and its argument are coded as a term within an expression, angle brackets should be used as delimiters to ensure precise evaluation and readability.

^F, as stated above, is a unary operator for numeric control which allows you to specify an argument that is a 1-word floating-point number. For example, the following statement:

```
A:      MOV   #^F3.7,R0
```

creates a 1-word floating-point number at location A+2 containing the value 3.7 formatted as shown below.

```
BIT 15              14     7              6     0

    S               EEEEEEEE              MMMMMMM

    Sign (1 bit)    Exponent (8 bits)     Mantissa (7 bits)
```

The importance of ordering with respect to unary operators is shown below.

```
^F1.0   = 040200
^F-1.0  = 140200
-^F1.0  = 137600
-^F-1.0 = 037600
```

The value created by the ^F unary operator and its argument is, like ^C and its argument, a term that can be used by itself or in an expression. For example:

```
^C^F6.2
```

is equivalent to:

```
^C<^F6.2>
```

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

6.5 LOCATION COUNTER CONTROL DIRECTIVES

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

Several MACRO-11 statements (listed below) may cause an odd number of bytes to be allocated:

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

In cases that yield an odd address value, the next word-boundaried instruction automatically forces the location counter to an even value, but that instruction is flagged with an error code (B) in the assembly listing.

6.5.1 .EVEN Directive

.EVEN

Format:

```
.EVEN
```

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

The .EVEN directive is used as follows:

```
        .ASCIZ  /THIS IS A TEST/
        .EVEN                   ;ENSURES THAT THE NEXT STATEMENT WILL
                                ;BEGIN ON A WORD BOUNDARY.
        .WORD   XYZ
```

6.5.2 .ODD Directive

.ODD

Format:

```
.ODD
```

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

6.5.3 .BLKB and .BLKW Directives

.BLKB
.BLKW

Formats:

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

where: exp represents the specified number of bytes or words to be reserved in the object program. Any expression that is defined at assembly time and that reduces to an absolute value is legal. If the expression specified in either of these directives is not an absolute value, the statement is flagged with an error code (A) in the assembly listing. These directives should not be used without arguments. However, if no argument is present, a default value of 1 is assumed.

The .BLKB directive reserves byte blocks in the object module; the .BLKW directive reserves word blocks. Figure 6-6 illustrates the use of the .BLKB and .BLKW directives.

```
 1                ;+
 2                ; Illustrate use of .BLKB and .BLKW directives
 3                ;-
 4 000000                 .PSECT  IMPURE,D,GBL,RW
 5
 6 000000         COUNT:  .BLKW   1       ;Character counter
 7
 8 000002         MESSAG: .BLKB   80.     ;Message text buffer
 9
10 000122         CHRSAV: .BLKB           ;Saved character
11
12 000123         FLAG:   .BLKB           ;Flag byte
13
14 000124         MSGPTR: .BLKW           ;Message buffer pointer
```

Figure 6-6 Example of .BLKB and .BLKW Directives

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

```
.=.+expression
```

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

6.5.4 .LIMIT Directive

.LIMIT

Format:

```
.LIMIT
```

To know the upper and lower address boundaries of the image is often desirable. When the .LIMIT directive is specified in the source program, MACRO-11 generates the following instruction:

```
.BLKW   2
```

causing two storage words to be reserved in the object module. Later, at link time, the lowest address in the load image (the initial value of SP) is inserted into the first reserved word, and the address of the first free word following the image is inserted into the second reserved word.

During linking, the size of the image is rounded upward to the nearest 2-word boundary.

6.6 TERMINATING DIRECTIVES

6.6.1 .END Directive

.END

Format:

```
.END    [exp]
```

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

When MACRO-11 encounters a valid occurrence of the .END directive, it terminates the current assembly pass. Any text beyond this point in the current source file, or in additional source files identified in the command line, will be ignored.

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

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

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

6.6.2 .EOT Directive

.EOT

Under RSX-11M, RT-11, RSTS and IAS operating systems, the MACRO-11 .EOT (End of Tape) directive is ignored and simply treated as a directive without effect.

6.7 PROGRAM SECTIONING DIRECTIVES

The MACRO-11 program sectioning directives are used to declare names for program sections (p-sections) and to establish certain program section attributes essential to linking.

6.7.1 .PSECT Directive

.PSECT

Format:

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

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

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

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

Table 6-4
Symbolic Arguments of .PSECT Directive

Argument	Default	Meaning
NAME	Blank	Establishes the program section name, which is specified as one to six Radix-50 characters. If this argument is omitted, a comma must appear in place of the name parameter. The Radix-50 character set is listed in Appendix A.2.
RO/RW	RW	Defines which type of access is permitted to the program section: RO=Read-Only Access RW=Read/Write Access NOTE RSX-11M and RT-11 use only Read/Write access.

(continued on next page)

Table 6-4 (Cont.)
Symbolic Arguments of .PSECT Directive

Argument	Default	Meaning
I/D	I	Defines the contents of the program section: I=Instructions. If a p-section has the I attribute and the program is overlaid, all calls to the p-section are referenced through a body of overlay code stored in the root. If a concatenated p-section has the I attribute, code is concatenated on even bytes. D=Data. If a p-section has the D attribute, all calls to the p-section are referenced directly. If a concatenated p-section has the D attribute, code is concatenated on the next byte regardless of whether the byte is odd or even.
GBL/LCL	LCL	Defines the scope of the program section, as it will be interpreted at link time: NOTE The GBL/LCL arguments apply only in the case of overlays; in building single-segment nonoverlaid programs, the GBL/LCL arguments have no meaning, because the total memory allocation for the program will go into the root segment of the image. LCL=Local. If an object module contains a local program section, then the storage allocation for that module will remain in the segment containing the module. Many modules can contribute (allocate memory) to this same program section; the memory allocation for each contributing module is either concatenated or overlaid within the segment, depending on the allocation argument of the program section (see CON/OVR below).

(continued on next page)

Table 6-4 (Cont.)
Symbolic Arguments of .PSECT Directive

Argument	Default	Meaning
GBL/LCL (cont.)		GBL=Global. If a global program section is used in more than one segment of a program, all references to the p-section are collected across segment boundaries. The program sections are then stored in the segment (of those originally containing the p-sections) that is nearest the root. NOTE RT-11 stores the collected p-sections in the root.
ABS/REL	REL	Defines the relocatability attribute of the program section: ABS=Absolute (non-relocatable). The ABS argument causes the linker or task builder to treat the p-section as an absolute module; therefore, no relocation is required. The program section is assembled and loaded, starting at absolute virtual address 0. The location of data in absolute program sections must fall within the virtual memory limits of the segment containing the program section; otherwise, an error results at link time. For example, the following code, although valid during assembly, may generate an error message (A) if virtual location 100000 is outside the segment's virtual address space: .PSECT ALPHA,ABS .=.+100000 .WORD X REL=Relocatable. The REL argument causes the linker or task builder to treat the p-section as a relocatable module and a relocation bias is added to all location references within the program section making the references absolute.

(continued on next page)

Table 6-4 (Cont.)
Symbolic Arguments of .PSECT Directive

Argument	Default	Meaning
CON/OVR	CON	Defines the allocation requirements of the program section: CON=Concatenated. All references to one program section are concatenated to determine the total memory space needed for the p-section. OVR=Overlaid. All references to one program section are overlaid; the total memory space needed equaling the largest, individual p-section.

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

```
.PSECT  ,GBL
```

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

The .PSECT directive may be used without a name or arguments (see Section 6.7.1.1).

The .PSECT directive allows a user to create program sections (see Section 6.7.1.1) and to share code and data among the sections he has created (see Section 6.7.1.2). In declaring the program sections (also called p-sections), you may declare the attributes of the p-sections. This allows you to control memory allocation and at the same time increases program modularity. (For a discussion of memory allocation, refer to the applicable system manual - see Section 0.3 in the Preface.)

MACRO-11 provides for 256(10) program sections, as listed below:

1. One default absolute program section (. ABS.)
2. One default relocatable program section (. BLK.)*
3. Two-hundred-fifty-four named program sections.

For each program section specified or implied, MACRO-11 maintains the following information:

1. Program section name
2. Contents of the current location counter
3. Maximum location counter value encountered
4. Program section attributes (described in Table 6-4 above).

* In RT-11 this program section is unnamed

6.7.1.1 **Creating Program Sections** - The first statement of a source program is always an implied .PSECT directive; this causes MACRO-11 to begin assembling source statements at relocatable zero of the unnamed program section.

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

```
        .PSECT                  ;DECLARES UNNAMED RELOCATABLE PROGRAM
A:      .WORD   0               ;SECTION ASSEMBLED AT RELOCATABLE
B:      .WORD   0               ;ADDRESSES 0, 2, AND 4.
C:      .WORD   0
        .PSECT  ALPHA           ;DECLARES RELOCATABLE PROGRAM SECTION
X:      .WORD   0               ;NAMED ALPHA ASSEMBLED AT RELOCATABLE
Y:      .WORD   0               ;ADDRESSES 0 AND 2.
        .PSECT                  ;RETURNS TO UNNAMED RELOCATABLE
D:      .WORD   0               ;PROGRAM SECTION AND CONTINUES ASSEM-
                                ;BLY AT RELOCATABLE ADDRESS 6.
```

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

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

and later referenced through the equivalent directive:

```
.PSECT   ALPHA
```

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

By maintaining separate location counters for each program section, MACRO-11 allows you to write statements that are not physically sequential but that can be loaded sequentially following assembly, as shown in the following example.

```
        .PSECT  SEC1,REL,RO     ;START A RELOCATABLE PROGRAM SECTION
A:      .WORD   0               ;NAMED SEC1 ASSEMBLED AT RELOCATABLE
B:      .WORD   0               ;ADDRESSES 0, 2, AND 4.
C:      .WORD   0
ST:     CLR     A               ;ASSEMBLE CODE AT RELOCATABLE
        CLR     B               ;ADDRESSES 6 THROUGH 12.
        CLR     C
        .PSECT  SECA,ABS        ;START AN ABSOLUTE PROGRAM SECTION
                                ;NAMED SECA.  ASSEMBLE CODE AT
        .WORD   .+2,A           ;ABSOLUTE ADDRESSES 0 AND 2.
        .PSECT  SEC1            ;RESUME RELOCATABLE PROGRAM SECTION
        INC     A               ;SEC1.  ASSEMBLE CODE AT RELOCATABLE
        BR      ST              ;ADDRESSES 14 AND 16.
```

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

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

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

the label A is assigned to relocatable address zero of the unnamed program section.

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

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

```
        .PSECT  ENT,ABS
.=.+1000
A:      CLR     X               ;ASSEMBLED AS CLR BASE OF
                                ;RELOCATABLE SECTION + 10(8).
        JMP     Y               ;ASSEMBLED AS JMP BASE OF
                                ;RELOCATABLE SECTION + 6(8).
        .PSECT  SEN,REL
        MOV     R0,R1
        JMP     A               ;ASSEMBLED AS JMP 1000.
Y:      HALT
X:      .WORD   0
```

NOTE

In the preceding example, using a constant in conjunction with the current location counter symbol (.) in the form .=1000 would result in an error, because constants are always absolute and are always associated with the program's .ASECT (. ABS.). If the form .=1000 were used, a program section incompatibility would be detected. See Section 3.6 for a discussion of the current location counter.

Thus, MACRO-11 provides the linker or task builder with the necessary information to resolve the linkages between various program sections. Such information is not necessary, however, when referencing an absolute program section, because all instructions in an absolute program section are associated with an absolute virtual address.

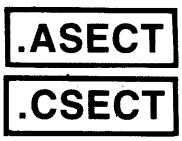
6.7.1.2 **Code or Data Sharing** - Named relocatable program sections with the arguments GBL and OVR operate in the same manner as FORTRAN COMMON, that is, program sections of the same name with the arguments GBL and OVR from different assemblies are all loaded at the same location at link time. All other program sections (those with the argument CON) are concatenated.

A single symbol could name both an internal symbol and a program section. Considering FORTRAN again, using the same symbolic name is necessary to accommodate the following statement:

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

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

6.7.1.3 **Memory Allocation Considerations** - The assembler does not generate an error when a module ends at an odd location. You can, therefore, place odd length data at the end of a module. However, when several modules contain object code contributions to the same program section having the concatenate attribute (see Table 6-4; CON/OVR), odd length modules (except the last) may cause succeeding modules to be linked starting at odd locations, thereby making the linked program unexecutable. To avoid this problem, separate code and data from each other and place them in separately named program sections (see Table 6-4; I/D). The linker or task builder can then begin each program section on an even address. Refer to the applicable system manual for further information on memory allocation of tasks (see Section 0.3 in the Preface).



6.7.2 .ASECT and .CSECT Directives

Formats:

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

where: symbol represents one or more of the arguments in Table 6-4.

IAS and RSX-11M assembly-language programs use the .PSECT and .ASECT directives exclusively, because the .PSECT directive provides all the capabilities of the .CSECT directive defined for other PDP-11 assemblers. MACRO-11 will accept both .ASECT and .CSECT directives, but assembles them as though they were .PSECT directives with the default attributes listed in Table 6-5. Compatibility exists between other MACRO-11 programs and the IAS/RSX-11M Task Builders, because the Task Builders also treat the .ASECT and .CSECT directives like .PSECT directives with the default values listed in Table 6-5.

Table 6-5
Program Section Default Values

Attribute	Default Value		
	.ASECT	.CSECT (named)	.CSECT (unnamed)
Name	. ABS.	name	. BLK.*
ACCESS	RW	RW	RW
Type	I	I	I
Scope	GBL	GBL	LCL
Relocation	ABS	REL	REL
Allocation	OVR	OVR	CON

Note that the statement:

```
.CSECT  JIM
```

is identical to the statement:

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

because the .CSECT default values GBL and OVR are assumed for the named program section.

6.7.3 .SAVE Directive

.SAVE

Format:

```
.SAVE
```

.SAVE stores the current program section context on the top of the program section context stack, while leaving the current program section context in effect. If the stack is full when .SAVE is issued, an error (A) occurs. The stack can handle 16 .SAVEs. The program section context includes the values of the current location counter and the maximum value assigned to the location counter in the current program section.

See Figure 6-7 for an example of .SAVE

* In RT-11 this program section has no default name.

6.7.4 .RESTORE Directive

.RESTORE

Format:

```
.RESTORE
```

The .RESTORE directive retrieves the program section from the top of the program section context stack. If the stack is empty when .RESTORE is issued, an error (A) occurs. When .RESTORE retrieves a program section, it restores the current location counter to the value it had when the program section was saved.

See Figure 6-7 for an example of .RESTORE.

```
.MAIN. MACRO Y04.00  18-JUL-79 14:53:42 PAGE 1
EXAMPLE OF .SAVE/.RESTORE USAGE

 1                                      .SBTTL  Example of .SAVE/.RESTORE usage
 2
 3                                ;+
 4                                ; Macro DS
 5                                ; Define local impure storage
 6                                ;-
 7
 8                                      .MACRO  DS      NAME,SIZE
 9                                      .SAVE                   ;Save the current PSECT
10                                      .PSECT  IMPURE,D,GBL    ;Store the data in the impure PSECT
11                                NAME: .BLKW   SIZE            ;Set aside the space
12                                      .RESTORE                ;Reenter the current PSECT
13                                      .ENDM
14
15
16                                ;+
17                                ; SCANSY
18                                ; Scan the hash table for valid entries
19                                ;-
20
21 000000  016701  000000'        SCANSY: MOV     SYMBAS,R1       ;Get base of table
22 000004  010167  000002'                MOV     R1,CURSYM       ;Initialize pointer to table
23 000010  066701  000004'                ADD     SYMSIZ,R1       ;Point past the table
24 000014  010167  000006'                MOV     R1,SYMTOP       ;Save end address
25                                ;
26                                ; Rest of routine...
27                                ;
28 000020  000207                       RETURN                  ;Table is scanned, exit
29
30                                ;+
31                                ; Local data
32                                ;-
33
34 000022                         DS      SYMBAS                  ;Base address of symbol table
35 000022                         DS      CURSYM                  ;Current symbol pointer during scan
36 000022                         DS      SYMSIZ                  ;Size of table, bytes
37 000022                         DS      SYMTOP                  ;Set to end address of table
38
39                                ;+
40                                ; SSORT
41                                ; Perform shell sort on symbol table prior to listing
42                                ;-
43
44 000022  016701  000006'        SSORT:  MOV     SYMTOP,R1       ;Get end of table
45
46                                ; Additional code ...
47
48         000001                       .END
```

Figure 6-7 Example of .SAVE and .RESTORE Directives

6.8 SYMBOL CONTROL DIRECTIVE

.GLOBL

Format:

```
.GLOBL  syml,sym2,...symn
```

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

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

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

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

Because object modules are linked by global symbols, these symbols are vital to a program. The following paragraph, describing the processing of a program from assembly to linking, explains the global's role.

In assembling a source program, MACRO-11 produces a relocatable object module and a listing file containing the assembly listing and symbol table. The linker or task builder joins separately assembled object modules into a single executable image. During linking, object modules are relocated relative to the base of the module and linked by global symbols. Because these symbols will be referenced by other program modules, they must be singled out as global symbols in the defining modules. As shown above, the .GLOBL directive, global assignment operator, or global label operator will define a symbol as global.

All internal symbols appearing within a given program must be defined at the end of assembly pass 1 or they will be assumed to be default global references. Refer to Section 6.2 for a description of enabling/disabling of global references.

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

```
;
; DEFINE A SUBROUTINE WITH 2 ENTRY POINTS WHICH CALLS AN
; EXTERNAL SUBROUTINE
;
        .PSECT                  ;DECLARE THE UNNAMED PROGRAM SECTION.
        .GLOBL  A               ;DEFINE A AS A GLOBAL SYMBOL.
A:      MOV     @(R5)+,R0       ;DEFINE ENTRY POINT A.
        MOV     #X,R1
X:      JSR     PC,C            ;CALL EXTERNAL SUBROUTINE C.
        RTS     R5              ;EXIT.
B::     MOV     (R5)+,R1        ;DEFINE ENTRY POINT B.
        CLR     R2
        BR      X
```

External symbols can appear in the operand field of an instruction or MACRO-11 directive as a direct reference, as shown in the examples below:

```
CLR     EXT
.WORD   EXT
CLR     @EXT
```

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

```
CLR     EXT+A
.WORD   EXT-2
CLR     @EXT+A(R1)
```

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

6.9 CONDITIONAL ASSEMBLY DIRECTIVES

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

6.9.1 Conditional Assembly Block Directives

.IF
.ENDC

Format:

```
.IF     cond,argument(s) ;START CONDITIONAL ASSEMBLY BLOCK.
  .
  .
  .
range                    ;RANGE OF CONDITIONAL ASSEMBLY BLOCK.
  .
  .
  .
.ENDC                    ;END OF CONDITIONAL ASSEMBLY BLOCK.
```

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

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

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

range	represents the body of code that is either included in the assembly, or excluded, depending upon whether the condition is met.
.ENDC	terminates the conditional assembly block. This directive must be present to end the conditional assembly block.

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

Table 6-6
Legal Condition Tests for Conditional Assembly Directives

Conditions			
Positive	Complement	Arguments	Assemble Block If:
EQ	NE	Expression	Expression is equal to 0 (or not equal to 0).
GT	LE	Expression	Expression is greater than 0 (or less than or equal to 0).
LT	GE	Expression	Expression is less than 0 (or greater than or equal to 0).
DF	NDF	Symbolic argument	Symbol is defined (or not defined).
B	NB	Macro-type argument	Argument is blank (or non-blank).
IDN	DIF	Two macro-type arguments	Arguments are identical (or different).
Z	NZ	Expression	Same as EQ/NE.
G	L	Expression	Same as GT/LT.

NOTE

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

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

An example of a conditional assembly directive follows:

```
.IF EQ  ALPHA+1          ;ASSEMBLE BLOCK IF ALPHA+1=0.
  .
  .
  .
.ENDC
```

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

&	Logical AND operator
!	Logical inclusive OR operator

For example, the conditional assembly statement:

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

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

Nested conditional directives take the form:

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

For example, the following conditional directives:

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

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

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

MACRO-11 permits a nesting depth of 16(10) conditional assembly levels. Any statement that attempts to exceed this nesting level depth is flagged with an error code (O) in the assembly listing.

.IFF
.IFT
.IFTF

6.9.2 Subconditional Assembly Block Directives

Formats:

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

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

1. The assembly of an alternate body of code when the condition of the block tests false.

2. The assembly of a non-contiguous body of code within the conditional assembly block, depending upon the result of the conditional test in entering the block.

3. The unconditional assembly of a body of code within a conditional assembly block.

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

Table 6-7
Subconditional Assembly Block Directives

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

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

EXAMPLE 1: Assume that symbol SYM is defined.

```
        .IF DF  SYM             ;TESTS TRUE, SYM IS DEFINED. ASSEMBLE
          .                     ;THE FOLLOWING CODE.
          .
          .
        .IFF                    ;TESTS FALSE. SYM IS DEFINED. DO NOT
          .                     ;ASSEMBLE THE FOLLOWING CODE.
          .
          .
        .IFT                    ;TESTS TRUE. SYM IS DEFINED. ASSEM-
          .                     ;BLE THE FOLLOWING CODE.
          .
          .
        .IFTF                   ;ASSEMBLE FOLLOWING CODE UNCONDITION-
          .                     ;ALLY.
          .
          .
        .IFT                    ;TESTS TRUE. SYM IS DEFINED. ASSEM-
          .                     ;BLE REMAINDER OF CONDITIONAL ASSEM-
          .                     ;BLY BLOCK.
          .
        .ENDC
```

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

```
        .IF DF  X               ;TESTS TRUE, SYMBOL X IS DEFINED.
        .IF DF  Y               ;TESTS FALSE, SYMBOL Y IS NOT DEFINED.
        .IFF                    ;TESTS TRUE, SYMBOL Y IS NOT DEFINED,
          .                     ;ASSEMBLE THE FOLLOWING CODE.
          .
          .
        .IFT                    ;TESTS FALSE, SYMBOL Y IS NOT DEFINED.
          .                     ;DO NOT ASSEMBLE THE FOLLOWING CODE.
          .
          .
        .ENDC
        .ENDC
```

EXAMPLE 3: Assume that symbol A is defined and that symbol B is not defined.

```
        .IF DF  A               ;TESTS TRUE.  A IS DEFINED.
                                ;ASSEMBLE THE FOLLOWING CODE.
        MOV     A,R1
          .
          .
          .
        .IFF                    ;TESTS FALSE.  A IS DEFINED.  DO NOT
                                ;ASSEMBLE THE FOLLOWING CODE.
        MOV     R1,R0
          .
          .
          .
        .IF NDF B               ;NESTED CONDITIONAL DIRECTIVE IS NOT
          .                     ;EVALUATED.
          .
          .
        .ENDC
        .ENDC
```

EXAMPLE 4: Assume that symbol X is not defined and that symbol Y is defined.

```
        .IF DF  X               ;TESTS FALSE. SYMBOL X IS NOT DEFINED.
                                ;DO NOT ASSEMBLE THE FOLLOWING CODE.
        .IF DF  Y               ;NESTED CONDITIONAL DIRECTIVE IS NOT
          .                     ;EVALUATED.
          .
          .
        .IFF                    ;NESTED SUBCONDITIONAL DIRECTIVE IS
          .                     ;NOT EVALUATED.
          .
          .
        .IFT                    ;NESTED SUBCONDITIONAL DIRECTIVE IS
          .                     ;NOT EVALUATED.
          .
          .
        .ENDC
        .ENDC
```

6.9.3 Immediate Conditional Assembly Directive

Format:

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

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

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

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

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

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

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

For example, the immediate conditional statement:

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

generates the code

```
BEQ     ALPHA
```

if the symbol FOO is defined within the source program.

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

6.9.4 PAL-11R Conditional Assembly Directives

In order to maintain compatibility with programs developed under PAL-11R, using the following conditionals remains permissible under MACRO-11. It is advisable, however, to develop future programs 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	Symbolic argument	Symbol is defined
.IFNDF	Symbolic argument	Symbol is undefined

The rules governing these directives are the same as those for the MACRO-11 conditional assembly directives previously described.

CHAPTER 7

MACRO DIRECTIVES

7.1 DEFINING MACROS

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

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

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

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

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

7.1.1 .MACRO Directive

.MACRO

FORMAT:

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

where:

- label — represents an optional statement label.
- name — represents the user-assigned symbolic name of the macro. This name may be any legal symbol and may be used as a label elsewhere in the program.
- ' — represents any legal separator (comma, space, and/or tab).

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

A comment may follow the dummy argument list in a .MACRO directive, as shown below:

```
.MACRO  ABS  A,B        ;DEFINES MACRO ABS WITH TWO ARGUMENTS.
```

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

NOTE

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

This NOTE also applies to .IRP, .IRPC, and .REPT.

7.1.2 .ENDM Directive

.ENDM

FORMAT:

```
.ENDM   [name]
```

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

Example:

```
.ENDM                   ;TERMINATES THE CURRENT
                        ;MACRO DEFINITION.

.ENDM   ABS             ;TERMINATES THE CURRENT
                        ;MACRO DEFINITION NAMED ABS.
```

If specified, the macro name in the .ENDM statement must match the name specified in the corresponding .MACRO directive. Otherwise, the statement is flagged with an error code (A) in the assembly listing. In either case, the current macro definition is terminated. Specifying the macro name in the .ENDM statement thus permits MACRO-11 to detect missing .ENDM statements or improperly nested macro definitions.

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

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

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

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

7.1.3 .MEXIT Directive

.MEXIT

FORMAT:

```
.MEXIT
```

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

```
.MACRO  ALTR N,A,B
   .
   .
   .
.IF EQ  N               ;START CONDITIONAL ASSEMBLY BLOCK.
   .
   .
   .
.MEXIT                  ;TERMINATE MACRO EXPANSION.
.ENDC                   ;END CONDITIONAL ASSEMBLY BLOCK.
   .
   .
   .
.ENDM                   ;NORMAL END OF MACRO.
```

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

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

7.1.4 MACRO Definition Formatting

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

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

7.2 CALLING MACROS

FORMAT:

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

where:

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

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

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

```
ABS:    MOV     (R0),R1         ;ABS IS DEFINED AS A LABEL.
          .
          .
          .
        BR      ABS             ;ABS IS CONSIDERED TO BE A LABEL.
          .
          .
          .
        ABS     #4,ENT,LAR      ;ABS IS A MACRO CALL.
```

7.3 ARGUMENTS IN MACRO DEFINITIONS AND MACRO CALLS

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

Macro definition arguments (dummy) and macro call arguments (real) normally maintain a strict positional relationship. That is, the first real argument in a macro call corresponds with the first dummy argument in a macro definition. Only the use of keyword arguments in a macro call can override this correspondence (see Section 7.3.6).

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

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

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

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

causes the entire expression

```
MOV     X,Y
```

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

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

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

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

Because of the use of the up-arrow (^) shown above, care must be taken when passing an argument beginning with a unary operator (^O, ^D, ^B, ^R, ^F ...). These arguments must be enclosed in angle brackets (as shown below) or MACRO-11 will read the character following the up-arrow as a delimiter.

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

The following macro call:

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

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

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

7.3.1 Macro Nesting

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

To pass an argument containing legal argument delimiters to nested macros, enclose the argument in the macro definition within angle brackets, as shown in the coding sequence below. This extra set of

angle brackets for each level of nesting is required in the macro definition, not in the macro call.

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

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

A call to the LEVEL1 macro, as shown below, for example:

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

causes the following macro expansion to occur:

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

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

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

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

7.3.2 Special Characters in Macro Arguments

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

```
.MACRO  PUSH ARG
MOV     ARG,-(SP)
.ENDM
   .
   .
   .
PUSH    X+3(%2)
```

causes the following code to be generated:

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

7.3.3 Passing Numeric Arguments as Symbols

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

```
        .MACRO  INC A,B
        CON     A,\B            ;B IS TREATED AS A NUMBER IN CURRENT
B=B+1                           ;PROGRAM RADIX.
        .ENDM
        .MACRO  CON A,B
A'B:    .WORD   4               ;A'B IS DESCRIBED IN SECTION 7.3.7.
        .ENDM
          .
          .
          .
C=0     INC     X,C
```

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

```
X0:     .WORD   4
```

In this expanded code, the label X0: results from the concatenation of two real arguments. The single quote (') character in the label A'B: concatenates the real arguments X and 0 as they are passed during the expansion of the macro. This type of argument construction is described in more detail in Section 7.3.7.

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

```
X1:     .WORD   4
```

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

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

```
        .MACRO  IDT  SYM         ;ASSUME THAT THE SYMBOL ID TAKES
        .IDENT  /V05A'SYM/       ;ON A UNIQUE 2-DIGIT VALUE.
        .ENDM                    ;WHERE V05A IS THE UPDATE
            .                    ;VERSION OF THE PROGRAM.
            .
            .
        IDT     \ID
```

The above macro call would then expand to:

```
        .IDENT  /V05A6/
```

where 6 is the numeric value of the symbol ID.

7.3.4 Number of Arguments in Macro Calls

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

7.3.5 Creating Local Symbols Automatically

A label is often required in an expanded macro. In the conventional macro facilities thus far described, a label must be explicitly specified as an argument with each macro call. The user must be careful in issuing subsequent calls to the same macro in order to avoid duplicating labels. This concern can be eliminated through a feature of MACRO-11 that creates a unique symbol where a label is required in an expanded macro.

As noted in Section 3.5, MACRO-11 can automatically create local symbols of the form n$, where n is a decimal integer within the range 64 through 127, inclusive. Such local symbols are created by MACRO-11 in numerical order, as shown below:

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

This automatic generation is invoked on each call of a macro whose definition contains a dummy argument preceded by the question mark (?) character, as shown in the macro definition below:

```
        .MACRO  ALPHA, A,?B     ;CONTAINS DUMMY ARGUMENT B PRECEDED BY
                                ;QUESTION MARK.
        TST     A
        BEQ     B
        ADD     #5,A
B:
        .ENDM
```

A local symbol is created automatically by MACRO-11 only when a real argument of the macro call is either null or missing, as shown in Example 1 below. If the real argument is specified in the macro call, however, MACRO-11 inhibits the generation of a local symbol and normal argument replacement occurs, as shown in Example 2 below. (Examples 1 and 2 are both expansions of the Alpha macro defined above.)

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

```
        ALPHA   R1              ;SECOND ARGUMENT IS MISSING.
        TST     R1
        BEQ     64$             ;LOCAL SYMBOL IS CREATED.
        ADD     #5,R1
64$:
```

EXAMPLE 2: Do Not Create a Local Symbol:

```
        ALPHA   R2,XYZ          ;SECOND ARGUMENT XYZ IS SPECIFIED.
        TST     R2
        BEQ     XYZ             ;NORMAL ARGUMENT REPLACEMENT OCCURS.
        ADD     #5,R2
XYZ:
```

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

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

When a macro has several arguments earmarked for automatic local symbol generation, substituting a specific label for one such argument risks assembly errors because MACRO-11 constructs its argument substitution list at the point of macro invocation. Therefore, the appearance of a label, the .ENABL LSB directive, or the .PSECT directive, in the macro expansion will create a new local symbol block. The new local symbol block could leave local symbol references in the previous block and their symbol definitions in the new one, causing error codes in the assembly listing (see Appendix D). Furthermore, a later macro expansion that creates local symbols in the new block may duplicate one of the symbols in question, causing an additional error code (P) in the assembly listing.

7.3.6 Keyword Arguments

FORMAT:

```
name=string
```

where: name represents the dummy argument,

string represents the real symbolic argument.

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

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

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

```
 1                              .LIST   ME
 2                      ;
 3                      ; DEFINE A MACRO HAVING KEYWORDS IN DUMMY ARGUMENT LIST
 4                      ;
 5
 6                              .MACRO  TEST CONTRL=1,BLOCK,ADDRES=TEMP
 7                              .WORD   CONTRL
 8                              .WORD   BLOCK
 9                              .WORD   ADDRES
10                              .ENDM
11
12
13                      ;
14                      ; NOW INVOKE SEVERAL TIMES
15                      ;
16
17 000000                      TEST    A,B,C
   000000  000000G              .WORD   A
   000002  000000G              .WORD   B
   000004  000000G              .WORD   C
18
19 000006                      TEST    ADDRES=20,BLOCK=30,CONTRL=40
   000006  000040               .WORD   40
   000010  000030               .WORD   30
   000012  000020               .WORD   20
20
21 000014                      TEST    BLOCK=5
   000014  000001               .WORD   1
   000016  000005               .WORD   5
   000020  000000G              .WORD   TEMP
22
23 000022                      TEST    CONTRL=5,ADDRES=VARIAB
   000022  000005               .WORD   5
   000024  000000               .WORD
   000026  000000G              .WORD   VARIAB
24
25 000030                      TEST
   000030  000001               .WORD   1
   000032  000000               .WORD
   000034  000000G              .WORD   TEMP
26
27 000036                      TEST    ADDRES=JACK!JILL
   000036  000001               .WORD   1
   000040  000000               .WORD
   000042  000000C              .WORD   JACK!JILL
28
29
30          000001             .END
```

7.3.7 Concatenation of Macro Arguments

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

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

when the macro DEF is called through the statement:

```
        DEF     X,Y,<MACRO-11>
```

it is expanded, as follows:

```
XY:     .ASCIZ  /MACRO-11/
        .BYTE   'X,'Y
```

In expanding the first line, the scan for the first argument terminates upon finding the first apostrophe (') character. Since A is a dummy argument, the apostrophe (') is removed. The scan then resumes with B; B is also noted as another dummy argument. The two real arguments X and Y are then concatenated to form the label XY:. The third dummy argument is noted in the operand field of the .ASCIZ directive, causing the real argument MACRO-11 to be substituted in this field.

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

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

MACRO-11 has three directives that allow the user to determine certain attributes of macro arguments: .NARG, .NCHR, and .NTYPE. The use of these directives permits selective modifications of a macro expansion, depending on the nature of the arguments being passed. These directives are described below.

7.4.1 .NARG Directive

.NARG

FORMAT:

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

where: label represents an optional statement label.

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

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

An example of the .NARG directive is shown in Figure 7-1.

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

Figure 7-1 Example of .NARG Directive

7.4.2 .NCHR Directive

.NCHR

FORMAT:

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

where:

label — represents an optional statement label.

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

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

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

The .NCHR directive, which can appear anywhere in a MACRO-11 program, is used to determine the number of characters in a specified character string. This directive is useful in calculating the length of macro arguments.

An example of the .NCHR directive is shown in Figure 7-2.

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

Figure 7-2 Example of .NCHR Directive

7.4.3 .NTYPE Directive

.NTYPE

FORMAT:

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

where:

label — represents an optional statement label.

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

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

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

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

An example of the use of an .NTYPE directive in a macro definition is shown in Figure 7-3.

```
 1                          .TITLE  NTYPE
 2
 3                          .ENABL  LC
 4                          .LIST   ME
 5
 6                  ;+
 7                  ; Illustrate the .NTYPE directive
 8                  ;-
 9
10                  .MACRO  SAVE    ARG
11                          .NTYPE  $$$,ARG
12                  .IF EQ  $$$&70
13                          MOV     ARG,-(SP)       ;Save in register mode
14                          .IFF
15                          MOV     #ARG,-(SP)      ;Save in non-register mode
16                  .ENDC
17                  .ENDM
18
19 000000                  SAVE    R1
           000001           .NTYPE  $$$,R1
                    .IF EQ  $$$&70
   000000  010146           MOV     R1,-(SP)        ;Save in register mode
                            .IFF
                            MOV     #R1,-(SP)       ;Save in non-register mode
                    .ENDC
20
21 000002                  SAVE    TEMP
           000067           .NTYPE  $$$,TEMP
                    .IF EQ  $$$&70
                            MOV     TEMP,-(SP)      ;Save in register mode
                            .IFF
   000002  012746           MOV     #TEMP,-(SP)     ;Save in non-register mode
           000006'
                    .ENDC
22
23 000006  000000  TEMP:    .WORD   0
24
25          000001          .END
```

Figure 7-3 Example of .NTYPE Directive in Macro Definition

For additional information concerning addressing modes, refer to Chapter 5 and Appendix B.2.

7.5 .ERROR AND .PRINT DIRECTIVES

.ERROR

FORMAT:

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

where:

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

  denotes the beginning of the text string.
- text — represents the message associated with the .ERROR directive.

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

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

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

For example, the following directive:

```
        .ERROR  A    ;INVALID MACRO ARGUMENT
```

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

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

P   512 005642  000076          .ERROR  A      ;INVALID MACRO ARGUMENT
```

.PRINT

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

7.6 INDEFINITE REPEAT BLOCK DIRECTIVES: .IRP AND .IRPC

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

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

7.6.1 .IRP Directive

FORMAT:

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

where: label represents an optional statement label.

NOTE

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

This NOTE also applies to .IRPC and .REPT.

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

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

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

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

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

The .IRP directive is used to replace a dummy argument with successive real arguments specified in an argument string. This replacement process occurs during the expansion of an indefinite repeat block range.

An example of the use of the .IRP directive is shown in Figure 7-4.

7.6.2 .IRPC Directive

.IRPC

FORMAT:

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

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

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

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

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

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

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

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

An example of the use of the .IRPC directive is shown in Figure 7-4.

```
 1                              .TITLE  IRPTST
 2
 3                              .LIST   ME
 4                      ;+
 5                      ; Illustrate the .IRP and .IRPC directives
 6                      ; by creating a pair of RAD50 tables
 7                      ;-
 8
 9 000000               REGS:   .IRP    REG,<PC,SP,R5,R4,R3,R2,R1,R0>
10                              .RAD50  /REG/
11                              .ENDR
   000000  062170               .RAD50  /PC/
   000002  074500               .RAD50  /SP/
   000004  072770               .RAD50  /R5/
   000006  072720               .RAD50  /R4/
   000010  072650               .RAD50  /R3/
   000012  072600               .RAD50  /R2/
   000014  072530               .RAD50  /R1/
   000016  072460               .RAD50  /R0/
12
13 000020               REGS2:  .IRPC   NUM,<76543210>
14                              .RAD50  /R'NUM/
15                              .ENDR
   000020  073110               .RAD50  /R7/
   000022  073040               .RAD50  /R6/
   000024  072770               .RAD50  /R5/
   000026  072720               .RAD50  /R4/
   000030  072650               .RAD50  /R3/
   000032  072600               .RAD50  /R2/
   000034  072530               .RAD50  /R1/
   000036  072460               .RAD50  /R0/
16
17         000001               .END
```

Figure 7-4 Example of .IRP and .IRPC Directives

.REPT
.ENDR

7.7 REPEAT BLOCK DIRECTIVE: .REPT, .ENDR

FORMAT:

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

where:

label	represents an optional statement label (see Note in Section 7.6.1).
exp	represents any legal expression. This value controls the number of times the block of code is to be assembled within the program. When the expression value is less than or equal to zero (0), the repeat block is not assembled. If this expression is not an absolute value, the .REPT statement is flagged with an error code (A) in the assembly listing.
range	represents the block of code to be repeated. The repeat block may contain macro definitions, indefinite repeat blocks, other repeat blocks and/or the .MEXIT directive (see Section 7.1.3).
.ENDM or .ENDR	indicates the end of the repeat block range.

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

7.8 MACRO LIBRARY DIRECTIVE: .MCALL

.MCALL

FORMAT:

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

where:

arg1, arg2,... argn	represent the symbolic names of the macro definitions required in the assembly of the source program. The names must be separated by any legal separator (comma, space, and/or tab).

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

The /ML switch (see Section 8.1.3) under RSX-11M and the /LIBRARY qualifier (see Section 8.2.2) under IAS and RT-11, used with an input file specification, indicate to MACRO-11 that the file is a macro library. When a macro call is encountered in the source program, MACRO-11 first searches the user macro library for the named macro definitions, and, if necessary, continues the search with the system macro library.

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

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

The command strings to MACRO-11, through which file specifications are supplied, are described in detail in the applicable system manual (see Section 0.3 in the Preface).

MACRO DIRECTIVES

PART IV
OPERATING PROCEDURES

CHAPTER 8

IAS/RSX-11M/RSX-11M-PLUS OPERATING PROCEDURES

8.1 RSX-11M OPERATING PROCEDURES

The following sections describe MACRO-11 operating procedures that apply only to the RSX-11M system.

8.1.1 Initiating MACRO-11 Under RSX-11M/RSX-11M-PLUS

Following the entry of CTRL/C (^C) from an operator's console, the Monitor Console Routine (MCR) indicates its readiness to accept a command by prompting with:

```
MCR>
```

In response, any one of the five methods described below may be employed to initiate MACRO-11.

8.1.1.1 Method 1 - Direct MACRO-11 Call

FORMAT:

```
MCR>MAC
MAC>macll-cmd-string
```

The monitor console routine (MCR) accepts MAC as input, causing MACRO-11 to be activated. Since an assembly command string is not present with the MCR line, MACRO-11 then solicits input with the prompting sequence MAC> and waits for command string input. The command string input, macll-cmd-string, is any legal, syntactically correct MACRO-11 command string of the form described in Section 8.1.2. After the assembly of the indicated files has been completed, MACRO-11 again solicits command string input with the MAC> prompting sequence. This process is repeated until a CTRL/Z (^Z) is entered.

8.1.1.2 Method 2 - Using RUN Facility

FORMAT:

```
MCR>RUN ...MAC[/UIC=[g,m]]
MAC>macll-cmd-string
```

Method 2 is identical to method 1 except for the use of the MCR RUN command which requires the entire task name, including the 3-dot prefix. When the optional /UIC is used, the default User Identification Code (UIC) is changed for one execution. As in method 1, MACRO-11 solicits command string input (see Section 8.1.2) after the assembly of the indicated files is completed.

8.1.1.3 Method 3 - Single Assembly

FORMAT:

```
MCR>MAC macll-cmd-string
```

In method 3, because the command string input is included in the MCR command line, no prompting is necessary, and MACRO-11 exits after assembling the indicated files.

8.1.1.4 Method 4 - Install, Run Immediately, and Remove On Exit

FORMAT:

```
MCR>RUN $MAC[/UIC=[g,m]]
MAC>macll-cmd-string
```

Method 4 is used when the MACRO-11 assembler is not permanently installed in the system. The Monitor Console Routine (MCR) installs MAC from the system program directory and requests it under the specified User Identification Code (UIC). As in methods 1 and 2, MACRO-11 solicits command string input (see Section 8.1.2). After exiting, MACRO-11 is automatically removed from the system.

NOTE

MACRO-11 can be terminated by entering a CTRL/Z (^Z) at any time a request for command string input is present.

8.1.1.5 Method 5 - Using Indirect Filename Facility

FORMATS:

```
MCR>MAC
MAC>@filespec

     or

MCR>RUN ...MAC[/UIC=[g,m]]
MAC>@filespec

     or

MCR>MAC @filespec

     or

MAC>RUN $MAC[/UIC=[g,m]]
MAC>@filespec
```

These forms use the indirect file facility of RSX-11M, substitutes "@filespec" for the "macll-cmd-string" input used in methods 1 through 4. The file specified as "@filespec" contains MACRO-11 command strings. After this file is opened, command lines are read from the file until the end-of-file is detected. Only three nested levels of indirect files are permitted in MACRO-11.

8.1.2 RSX-11M Command String

FORMAT:

```
MAC>object,listing=srcl,src2,...,srcn
```

where:

object	represents the binary object (output) file.
listing	represents the assembly listing (output) file containing the table of contents, the assembly listing, and the symbol table.
=	separates output file specifications from input file specifications.
srcl, src2,... srcn	represent the ASCII source (input) files containing the MACRO-11 source program or the user-supplied macro library files to be assembled.

Only two output file specifications in the command string will be recognized by MACRO-11; any more than two such files will be ignored. No limit is set on the number of source input files. A command line must not be more than 132 characters long; however, a hyphen terminating the line allows the command to be continued on the following line. (An 8K assembler allows 80 characters per line and no continuation lines.)

A null specification in either of the output file specification fields signifies that the associated output file is not desired. A null specification in the input file field, however, is an error condition, resulting in the error message "MAC -- ILLEGAL FILENAME" on the command output device (see Section 8.6). The absence of both the device name (dev:) and the name of the file (filename.type) from a file specification is the equivalent of a null specification.

NOTE

When no listing file is specified, any errors encountered in the source program are printed on the terminal from which MACRO-11 was initiated. When the /NL switch is used in the listing file specification without an argument, the errors and symbol table are output to the specified listing file.

Each file specification contains the following information (in accordance with the standard RSX-11M conventions for file specifications):

```
filespec /switch:value ...
```

where: filespec is the standard RSX-11M file specification as described in Section 8.5.

/switch represents an ASCII name identifying a switch option. A switch option may be specified in three forms, as shown below, depending on the function desired:

/SW	Invokes the specified switch action.
/NOSW	Negates the specified switch action.
/-SW	Also negates the specified switch action.

:value ...

represents any number of the following values: ASCII character strings, octal numbers, or decimal numbers. The default assumption for a numeric value is octal. Decimal values must be followed by a decimal point (.).

Any numeric value preceded by a number sign (#) is regarded as an explicit octal declaration; this option is provided for documentation purposes and ready identification of octal values.

Also, any numeric value can be preceded by a plus sign (+) or a minus (-) sign. The positive specification is the default assumption. If an explicit octal declaration is specified (#), the sign indicator, if included, must precede the number sign.

Switch values must always be preceded by a colon (:).

The switch specifications are interpreted in the context of the program to which they apply. The MACRO-11 switch options are described in Table 8-2.

A syntactical error detected in the command string causes MACRO-11 to output the following error message to the command output device (see Section 8.6):

```
MAC -- COMMAND SYNTAX ERROR
```

followed by a copy of the entire command string.

Table 8-1 lists the default values for each file specification.

Table 8-1
File Specification Default Values

File	Device	Default Value		
		Directory	Filename	Type
Object File	System device.	Current	None	.OBJ
Listing File	Device used for object file.	Directory used in object file	None	.LST
Source 1 File	System device.	Current	None	.MAC
Source 2 to Source n File	Device used for source 1 or last source file specified.	Directory used for source 1 or last source file specified.	None	.MAC
User Macro Library	System device, if macro file is specified first; if not, device used by last source file is used.	Current if macro file is specified first; if not, directory of last source file is used.	None	.MLB
System Macro Library	System device.	[1,1]	RSXMAC	.SML
Indirect Command File	System device.	Current	None	.CMD

8.1.3 RSX-11M File Specification Switches

At assembly time, you may want to override certain MACRO-11 directives appearing in the source program, or you may want to direct how certain individual files are to be handled during assembly. You can do either of these by using switch options with each file (see Section 8.1.2). The available switches for use in MACRO-11 file specifications under RSX-11M are listed in Table 8-2.

Table 8-2
MACRO-11 File Specification Switches for RSX-11M

Switch	Function
/LI:arg /NL:arg	Listing control switches; these options accept ASCII switch values (arg) that are equivalent in function and name to the arguments of the .LIST and .NLIST directives specified in the source program (see Section 6.1.1). This switch overrides the arguments of the directives and remains in effect for the entire assembly process.
/EN:arg /DS:arg	Function control switches; these options accept ASCII switch values (arg) that are equivalent in function and name to the arguments of the .ENABL and .DSABL directives specified in the source program (see Section 6.2). This switch overrides the arguments of the directives and remains in effect for the entire assembly process.
/ML (see Note)	The /ML switch, which takes no accompanying switch values, indicates to MACRO-11 that an input file is a macro library file. Library files hold the definitions of externally defined macros. As noted in Section 7.8, an externally defined macro must be identified in an .MCALL directive before it can be retrieved and assembled with the user program. When MACRO-11 encounters an .MCALL directive, a search begins for the definitions of the macros listed. The search order is important because a macro might have two different definitions, in library files LIB1 and LIB2. If you need the definition, for example, in LIB1, then you must place LIB1 **after** LIB2 in the command line, because MACRO-11 searches the last file specified in the command line first, then moves backwards through the files given until all have been searched. If a macro's definition is not found in any of the files named by the user, MACRO-11 automatically searches the system macro library; if the definition is still not found an error code (U) is generated in the assembly listing.

(continued on next page)

Table 8-2 (Cont.)
MACRO-11 File Specification Switches for RSX-11M

Switch	Function
/PA:1 (see Note)	Assemble the associated file during assembly pass 1 only.
/PA:2 (see Note)	Assemble the associated file during assembly pass 2 only.
/SP (see Note)	Spool listing output (default value).
/NOSP	Do not spool output.
/CR:[arg]	Produce a cross-reference listing. See Section 8.4.

NOTE

The /ML, /PA and /SP switches do not interact with or override MACRO-11 directives. Rather, they have meaning only in the command line itself.

Switches for the object file are limited to /EN and /DS; when specified, they apply throughout the entire command string. Switch options for the listing file are limited to /LI, /NL, /SP, /CR, and /NOSP. Switches for input files are limited to /ML, /PA, /EN, and /DS; the options /ML and /PA apply only to the file immediately preceding the option, whereas the /EN and /DS options, as noted above, are applicable to all the files in the command string.

When using switches with a file specification, be careful not to use the same switch more than once, because the values accompanying the latest use of the switch will override the values accompanying any earlier uses of that switch. For example, in the following command string element:

```
/LI:SRC/LI:MEB
```

the switch specification /LI:MEB will override the previous /LI switch, /LI:SRC. If both switch values are desired, they can be specified in the form shown below:

```
/LI:SRC:MEB
```

Examples:

1. MAC>OBJFIL,LSTFIL/NL:BEX:COM/LI:ME=SRCFIL

   This command string suppresses the listing of binary extensions and the source comments and lists the macro expansions. Furthermore, it causes all listing directives in the source program having the arguments BEX, COM, and ME to be overridden. In this example, the object output is sent to the file named OBJFIL.OBJ, and the listing and symbol table output is sent to the file named LSTFIL.LST.

2. MAC>OBJFIL,LISTM/NL:TOC=SRCFIL

   This command string suppresses the assembly listing's table of contents. When the /NL switch is present in the file specification without an argument (general no-list mode) all listing output except the symbol table is suppressed.

8.2 OPERATING PROCEDURES APPLICABLE ONLY TO THE RSX-11M-PLUS SYSTEM

8.2.1 Initiating MACRO-11 Under RSX-11M-PLUS

RSX-11M-PLUS indicates its readiness to accept a command by prompting with:

```
DCL>
```

In response to this prompt the command strings below may be entered.

```
DCL> MACRO [qualifiers]
FILE?  filespec [qualifiers][,filespec,filespec...]
```

or

```
DCL>MACRO [qualifiers] filespec[qualifiers][,filespec,filespec...]
```

where: qualifiers — are commands that affect either the entire command string (command qualifiers) or only the filespec (parameter qualifiers). Command qualifiers are those that follow the MACRO command and parameter qualifiers are those that follow the filespec. For a description of the command and parameter qualifiers see Tables 8-3 and 8-4 respectively.

filespec — is the standard file specification given in Section 8.5. (The default file specifications are given in Table 8-1.)

Table 8-3
RSX-11M-PLUS Command Qualifiers

Qualifier	Function
/LIST [:filespec]	Produces an assembly listing file according to filespec (see Section 8.5). If filespec is not specified, the listing has the file name of the last source file and the file type .LST. This file is put in your User File Directory and also printed on the line printer. The default is /NOLIST.
/NOLIST [:filespec]	Suppresses an assembly listing of filespec (see Section 8.5). This is the default condition.

(continued on next page)

Table 8-3 (Cont.)
RSX-11M-PLUS Command Qualifiers

Qualifier	Function
/OBJECT [:filespec]	Produces an object file according to filespec (see Section 8.5). If filespec is not specified, the object file has the file name of the last source file and the file type .OBJ. The default is /OBJECT.
/NOOBJECT	Suppresses the generation of an object file.
/CROSS_REFERENCE	Generates a cross-reference listing (see Section 8.4). When this qualifier is used /LIST is specified by implication. If you wish to specify a filespec for the /LIST qualifier you must append /LIST:filespec.
/NOCROSS_REFERENCE	Suppresses the cross-reference listing. This is the default condition.
/SWITCHES:(/sw1:ar1:ar2 .../sw4:arn)	Overrides the .LIST,.NLIST,ENABL and .DSABL directives included in the source program. There are four switches (sw1...sw4) that can be used with /SWITCHES: Switch — Function /LI:ar, /NL:ar — These are the listing control switches; they accept the ASCII switch values (ar) which are equivalent in name and function to the arguments of the .LIST and .NLIST directives (see Section 6.1.1). /EN:ar, /DS:ar — These are the function control switches; they accept the ASCII switch values (ar) which are equivalent in name and function to the arguments of the .ENABL and .DSABL directives (see Section 6.2).

Table 8-4
RSX-11M-PLUS Parameter Qualifiers

Qualifier	Function
/PASS:1	Assembles the associated file during assembly pass 1 only.
/PASS:2	Assembles the associated file during assembly pass 2 only.
/LIBRARY	Specifies that an input file is a macro library file. Library files hold the definitions of externally defined macros. As noted in Section 7.8, an externally defined macro must be identified in an .MCALL directive before it can be retrieved and assembled with the user program. When MACRO-11 encounters an .MCALL directive, a search begins for the definitions of the macros listed. The search order is important because a macro might have two different definitions, in library files LIB1 and LIB2. If you need the definition, for example, in LIB1, then you must place LIB1 after LIB2 in the command line, because MACRO-11 searches the last file specified in the command line first, then moves backwards through the files given until all have been searched. If a macro's definition is not found in any of the files named by the user, MACRO-11 automatically searches the system macro library; if the definition is still not found, an error code (U) is generated in the assembly listing.

8.2.2 RSX-11M-PLUS Command String Examples

1. >MACRO
   FILE? FILENAM

   or

   >MACRO FILNAM

   Both of these examples assembles the source file FILENAM.MAC into a relocatable object module named FILNAM.OBJ.

2. >MACRO/OBJECT:TESTA FILNAM

   This example produces an object file with the name TESTA.OBJ.

3. >MACRO FILNAM/LIBRARY,TESTA,SPAN3,SHELL

   Assembles a (concatenated) object file named SHELL.OBJ from the source files FILNAM.MLB, TESTA.MAC, SPAN3.MAC and SHELL.MAC.

4. >MACRO/LIST/SWITCHES:(/NL:TTM:COM) FILNAM

   This command string produces an object module and an assembly listing. Any .LIST TTM or .LIST COM directive in the source file is ignored. The listing produced by this command has no comments included and is printed in wide format.

8.3 IAS MACRO-11 OPERATING PROCEDURES

The following sections describe those MACRO-11 operating procedures that apply exclusively to the IAS system.

8.3.1 Initiating MACRO-11 Under IAS

The MACRO command is used under IAS to begin MACRO-11 assembler operations. The command causes MACRO-11 to assemble one or more ASCII source files containing MACRO-11 statements into a relocatable binary object file. The assembler will also produce an assembly listing, followed by a symbol table listing. A cross-reference listing can also be produced, by means of the /CROSSREFERENCE qualifier (see 8.3.2, below).

A MACRO-11 program can be input either directly from the terminal (interactive mode) or from a batch file (batch mode). For interactive mode use the MACRO command which can be issued whenever the IAS Program Development System (PDS) is at command level, a condition signified by the appearance of the prompt:

```
PDS>
```

For batch mode use the $MACRO command.

When the assembly is completed, MACRO-11 terminates operations and returns control to PDS. (Refer to the IAS User's Guide for further information about interactive and batch mode operations.)

8.3.2 IAS Command String

FORMATS:

Interactive Mode

```
                                  input
PDS> MACRO qualifiers             filespec /LIBRARY +...

     or

PDS> MACRO qualifiers

       input
FILES? filespec /LIBRARY  +...
```

Batch Mode

```
                                  input
$MACRO qualifiers                 filespec /LIBRARY +...
```

where:

input filespec	is the specification of an input file (see Section 8.5) that contains MACRO-11 source program code. When the program consists of multiple files, a plus sign (+) must be used to separate each file specification from the next. The "wild card" form of a file specification is not allowed.
/LIBRARY	specifies that an input file is a macro library file. Library files hold the definitions of externally defined macros. As noted in Section 7.8, an externally defined macro must be identified in an .MCALL directive before it can be retrieved and assembled with the user program. When MACRO-11 encounters an .MCALL directive, a search begins for the definitions of the macros listed. The search order is important because a macro might have two different definitions in library files LIB1 and LIB2. For example, if you need the definition in LIB1, then you must place LIB1 **after** LIB2 in the command line because MACRO-11 searches the last file specified in the command line first, then moves backwards through the files given until all have been searched. If a macro's definition is not found in any of the files named by the user, MACRO-11 automatically searches the system macro library; if the definition is still not found, an error code (U) is generated in the assembly listing.
qualifiers	specifies one or more of the following:
/OBJECT[:output filespec]	produces an object file as specified by filespec (see Section 8.5). The default is a file with the same filename as the last named source file and an .OBJ extension. /OBJECT is always the default condition.
/NOOBJECT	Does not produce an object file.
/LIST[:output filespec]	produces an assembly listing file according to filespec (see Section 8.5). If filespec is not specified, the listing is printed on the line printer. The default in interactive mode is /NOLIST and in batch mode is /LIST.
/NOLIST	Does not produce a listing file. The default in interactive mode is /NOLIST and in batch mode is /LIST.

NOTE

When no listing file is specified, any errors encountered in the source program are displayed at the terminal from which MACRO-11 was initiated.

/CROSSREFERENCE[:arg1...arg4]

produces a cross-reference listing. Arg1 through arg4 are as described in Section 8.4. This qualifier may be abbreviated to /C.

A MACRO-11 command string can be specified using any one of the three formats shown above for the interactive and batch modes. In interactive mode, if the input file specification (filespec) does not begin on the same line as the MACRO command and its qualifiers, PDS prints the following prompting message:

FILES?

then waits for the user to specify the input file(s).

In batch mode, the $MACRO command and its arguments must appear on the same line unless the PDS line continuation symbol (-) is used.

8.3.3 IAS Indirect Command Files

FORMAT:

@filespec

where:

@ specifies that the name that follows is an indirect file.

filespec is the file specification (see Section 8.5) of a file that contains a command string. The default extension for the file name is .CMD.

The indirect command file facility of PDS can be used with MACRO-11 command strings. This is accomplished by creating an ASCII file that contains the desired command strings (or portions thereof) in the forms shown in Section 8.3.2. When an indirect command file reference is used in a MACRO-11 command string, the contents of the specified file are taken as all or part of the command string.

An indirect command file reference must always be the rightmost entry in the command (see Section 8.3.4 for examples).

8.3.4 IAS Command String Examples

The following examples show typical PDS MACRO-11 command strings.

1. PDS> MACRO /NOLIST
   FILES? A+BOOT.MAC;3

   In this example, the source files A.MAC and BOOT.MAC;3 will be assembled to produce an object file called BOOT.OBJ. No listing will be produced.

2. Where the indirect command file TEST.CMD contains the command string:

   ```
   MACRO/OBJECT:MYFILE  A+B
   ```

   The command:

   ```
   PDS>@TEST
   ```

   causes MACRO-11 to assemble the two files A.MAC and B.MAC into an object file called MYFILE.OBJ.

3. Where the indirect command file IND02.CMD contains the command string segment:

   ```
   ATEST/LIBRARY+BTEST+SRT1.021
   ```

   The command:

   ```
   PDS>MACRO/LIST:DK1:FOO   @IND02
   ```

   causes MACRO-11 to assemble the files BTEST.MAC and SRT1.021 using the macro library file ATEST.MAC to produce an object file named SRT1.OBJ. A listing file named FOO.LST is placed on disk unit 1.

4. $MACRO/LIST:DK0:MICR/NOOBJECT -
   LIB1/LIBRARY+MICR.MAC;002

   In this example, the library file is assembled with the file MICR.MAC;002. The program listing file named MICR.LST is placed on disk unit 0.

8.4 CROSS-REFERENCE PROCESSOR (CREF)

The CREF processor is used to produce a listing that includes cross-references to symbols that appear in the source program. The cross-reference listing is appended to the assembly listing. Such cross-references are helpful in debugging and in reading long programs.

A cross-reference listing can include up to four sections:

1. User-defined symbols
2. Macro symbols
3. Register symbols
4. Permanent symbols

To generate a cross-reference listing, specify the /CR switch in the MACRO-11 command string. Optional arguments can also be specified. The form of the switch is:

```
          SYM
    /CR : MAC
          REG
          PST
```

where:

SYM	specifies user-defined symbols (default)
MAC	specifies macro symbols (default)
REG	specifies register symbols
PST	specifies permanent symbols.

If you wish to generate listings for user-defined and macro symbols only, use /CR. No argument is necessary.

However, if an argument is specified, only that type of cross-reference listing is generated. For example:

```
/CR:SYM
```

produces a cross-reference listing of user-defined symbols only. No listing of macro symbols is generated. Thus, to produce all four types of cross-reference listings, you must specify all four arguments (the order in which they are specified is not significant). Use a colon to separate arguments. For example:

```
/CR:REG:SYM:MAC:PST
```

The CREF processor is more fully described in the Utilities Reference Manual supplied with your system.

Figure 8-1 illustrates a complete cross-reference listing. In the listing references are made in the form: page-line. To make the listing more informative the CREF Processor uses the following signs:

Sign	Meaning
=	somewhere in the source program the symbol listed is defined by a direct assignment statement.
*	destructive reference; at the line referenced by the processor the value of the symbol is changed (its previous contents destroyed).
#	at the line referenced by the processor the symbol listed is defined by a direct assignment statement, a colon sign (:) or a double colon sign (::).

```
R50UNP  MACRO Y04.00  29-AUG-79 16:06:39 PAGE S-1
CROSS REFERENCE TABLE (CREF V01-08 )

R50UNP   2-16#
SYMBOL   2-17     2-25



R50UNP  MACRO Y04.00 29-AUG-79 16:06:39 PAGE R-1
CROSS REFERENCE TABLE (CREF V01-08 )

R0       2-23*    2-32*    2-33*    2-43     2-45     2-48*    2-49*
         2-50*    2-51*    2-52
R1       2-18*    2-23
R2       2-52*
R3       2-19*    2-21*    2-33
R4       2-16     2-17*    2-18     2-25     2-27*
SP       2-16*    2-27



R50UNP  MACRO Y04.00  29-AUG-79 16:06:39 PAGE C-1
CROSS REFERENCE TABLE (CREF V01-08 )

         0-0
. ABS.   0-0
PUREI    2-14
```

Figure 8-1 Sample CREF Listing

8.5 IAS/RSX-11M FILE SPECIFICATION

FORMAT:

dev:[g,m]name.ext;ver

where:

dev: is the name of the physical device where the desired file resides. A device name consists of two characters followed by a 1- or 2-digit device unit number (octal) and a colon (for example, DP1:, DK0:, DT3:). The default device under RSX-11M and RSX-11M-PLUS is as specified in Table 8-1. The default device under IAS is established initially by the system manager for each user and can be changed through the SET command.

[g,m] is the User File Directory (UFD) code. This code consists of a group number (octal), a comma (,), and an owner (member) number (octal) all enclosed in brackets ([]). An example of a UFD code is: [200,30].

The default UFD is equivalent to the User Identification Code (UIC) given at log-in time. Under IAS, the UFD can be changed through the SET DEFAULT command.

name is the filename and consists of one through nine alphanumeric characters. There is no default for a filename.

.ext — is a 1- to 3-alphanumeric character filename extension or type that is preceded by a period (.). An extension is normally used to identify the nature of the file. Default values depend on the context of the file specification and are as follows:

.CMD = Indirect command (input) file
.LST = A listing (print format) file
.MAC = MACRO-11 source module (input file)
.OBJ = MACRO-11 object module (output file)
.CRF = Intermediate CREF input file created by MACRO-11.

;ver — is an octal number between 1 and 77777 that is used to differentiate between versions of the same file. This number must be prefixed by a semicolon (;).

For input files, the default value is the highest version number of the file that exists.

For output files, the default value is the highest version number of the file that exists increased by 1. If no version number exists, the value 1 is used.

This is the general form for a file specification in IAS/RSX-11M systems. Detailed information is provided in the applicable system user's guide or operating procedures manual (see Section 0.3 in the Preface).

8.6 MACRO-11 ERROR MESSAGES UNDER IAS/RSX-11M

MACRO-11 outputs an error message to the command output device when one of the error conditions described below is detected. These error messages reflect operational problems and should not be confused with the error codes (see Appendix D) produced by MACRO-11 during assembly.

All the error messages listed in Table 8-5, with the exception of the "MAC -- COMMAND I/O ERROR" message, result in the termination of the current assembly; MACRO-11 then attempts to restart by reading another command line. In the case of a command I/O error, however, MACRO-11 exits, since it is unable to obtain additional command line input.

Table 8-5
Operational Error Messages; IAS/RSX-11M

Error Message	Meaning
MAC -- COMMAND FILE OPEN FAILURE	Either the file from which MACRO-11 is reading a command could not be opened initially or between assemblies; or the indirect command file specified as "@filename" in the MACRO-11 command line could not be opened. See "OPEN FAILURE ON INPUT FILE" for meaning.

(continued on next page)

Table 8-5 (Cont.)
Operational Error Messages; IAS/RSX-11M

Error Message	Meaning
MAC -- COMMAND I/O ERROR	An error was returned by the file system during MACRO-11's attempt to read a command line. This is an unconditionally fatal error, causing MACRO-11 to exit. No MACRO-11 restart is attempted when this message appears.
MAC -- COMMAND SYNTAX ERROR	An error was detected in the syntax of the MACRO-11 command line.
MAC -- ILLEGAL FILENAME	Neither the device name nor the filename was present in the input file specification (the input file specification was null), or a wild card convention (asterisk) was employed in an input or output file specification. Wildcard options (*) are not permitted in MACRO-11 file specifications.
MAC -- ILLEGAL SWITCH	An illegal switch was specified for a file, an illegal value was specified with a switch, or an invalid use of a switch was detected by MACRO-11.
MAC -- INDIRECT COMMAND SYNTAX ERROR	The name of the indirect command file (@filename) specified in the MACRO-11 command line is syntactically incorrect.
MAC -- INDIRECT FILE DEPTH EXCEEDED	An attempt to exceed the maximum allowable number of nested indirect command files has occurred. (Only three levels of indirect command files are permitted in MACRO-11.)

(continued on next page)

Table 8-5 (Cont.)
Operational Error Messages; IAS/RSX-11M

Error Message	Meaning
MAC -- INSUFFICIENT DYNAMIC MEMORY	There is not enough physical memory available for MACRO-11 to page its symbol table. Reinstall MACRO-11 in a larger partition, or see Appendix F.3.
MAC -- INVALID FORMAT IN MACRO LIBRARY	The library file has been corrupted, or it was not produced by the Librarian Utility Program (LBR).
MAC -- I/O ERROR ON INPUT FILE	In reading a record from a source input file or macro library file, the file system detected an error; for example, a line containing more than 132(10) characters was encountered. This message may also indicate that a device problem exists or that either a source file or a macro library file has been corrupted with incorrect data.
MAC -- I/O ERROR ON MACRO LIBRARY FILE	Same meaning as I/O ERROR ON INPUT FILE, except that the file is a macro library file and not a source input file.
MAC -- I/O ERROR ON OUTPUT FILE	In writing a record to the object output file or the listing output file, an error was detected by the file system. This message may also indicate that a device problem exists or that the device is full.
MAC -- I/O ERROR ON WORK FILE	A read or write error occurred on the work file used to store the symbol table. This error is most likely caused by a problem on the device or by attempting to write to a device that is full.

(continued on next page)

Table 8-5 (Cont.)
Operational Error Messages; IAS/RSX-11M

Error Message	Meaning
MAC -- OPEN FAILURE ON INPUT FILE	1. Specified device does not exist. 2. The volume is not mounted. 3. A problem exists with the device. 4. Specified directory file does not exist. 5. Specified file does not exist. 6. User does not have access to the file directory or to the file itself.
MAC -- OPEN FAILURE ON OUTPUT FILE	1. Specified device does not exist. 2. The volume is not mounted. 3. A problem exists with the device. 4. Specified directory file does not exist. 5. User does not have access to the file directory. 6. The volume is full or the device is write protected. 7. There is insufficient space for File Control Blocks.
MAC -- 64K STORAGE LIMIT EXCEEDED	64K words of work file memory are available to MACRO-11. This message indicates that the assembler has generated so many symbols (about 13,000 to 14,000) that it has run out of space. Either the source program is too large to start with, or it contains a condition that leads to excessive size, such as a macro expansion that recursively calls itself without a terminating condition.

CHAPTER 9

RSTS/RT-11 OPERATING PROCEDURES

9.1 MACRO-11 UNDER RSTS

The only way a MACRO-11 program can run on a RSTS system is through either the RT-11 or RSX run-time systems.

9.1.1 RT-11 Through RSTS

There are two ways to run a MACRO program under the RT-11 run-time system:

1. Use the RT-11 Emulator. This is done by typing: SW RT11. The terminal will respond with the RT-11 prompt (a dot printed by the keyboard monitor). You can then use the RT-11 commands (see Section 9.2).

2. Type the command: RUN $MACRO.SAV. The terminal will respond with an asterisk (*) prompt. You can then enter a command string of the form:

   ```
   OBJFIL,LSTFIL=SRC...SRC6
   ```

   where: OBJFIL is an object (output) file with the default extension .OBJ.

   LSTFIL is a listing (output) file with the default extension .LST.

   SRC... SRC6 are source (input) files with the default extension .MAC. Six input files are allowed in this command.

9.1.2 RSX Through RSTS

To run a MACRO program under the RSX run-time system, type the command: RUN $MAC.TSK. The terminal will respond with:

```
MAC>
```

In answer you enter a command string of the form:

```
OBJFIL,LSTFIL=SRC...SRCN
```

where:

OBJFIL	is an object (output) file with the default extension .OBJ.
LSTFIL	is a listing (output) file with the default extension .LST.
SRC... SRCN	are source (input) files with the default extension .MAC.

NOTE

There are other commands that can be used to call RT-11 and RSX but they are site dependent and so are not mentioned here.

9.2 INITIATING MACRO-11 UNDER RT-11

The following sections describe those MACRO-11 operating procedures that apply only to the RT-11 system.

To call the MACRO-11 assembler from the system device, respond to the system prompt (a dot printed by the keyboard monitor) by typing:

```
R MACRO
```

When the assembler responds with an asterisk (*), it is ready to accept command string input.

9.3 RT-11 COMMAND STRING

FORMAT:

```
[dev:obj,dev:list,dev:cref/s:arg]=dev:src1,src2,...,dev:srcn/s:arg
```

where

dev	is any legal RT-11 device for output; any file-structured device for input
obj	is the file specification of the binary object file that the assembly process produces; the device for this file should not be TT or LP
list	is the file specification of the assembly and symbol listing that the assembly process produces
cref	is the file specification of the CREF temporary cross-reference file that the assembly process produces. (Omission of dev:cref does not preclude a cross-reference listing, however.)
/s:arg	is a set of file specification options and arguments (see Section 9.2).
src1, src2,... srcn	represent the ASCII source (input) files containing the MACRO-11 source program or the user-supplied macro library files to be assembled. You can specify as many as six source files.

The following command string calls for an assembly that uses one source file plus the system MACRO library to produce an object file BINF.OBJ and a listing. The listing goes directly to the line printer.

```
*DK:BINK.OBJ,LP:=DK:SRC.MAC
```

All output file specifications are optional. The system does not produce an output file unless the command string contains a specification for that file.

The system determines the file type of an output file specification by its position in the command string, as determined by the number of commas in the string. For example, to omit the object file, you must begin the command string with a comma. The following command produces a listing, including cross-reference tables, but not binary object files.

```
*,LP:/C=(source file specification)
```

Notice that you need not include a comma after the final output file specification in the command string.

Table 9-1 lists the default values for each file specification.

Table 9-1
Default File Specification Values

File	Default Device	Default File Name	Default File Type
Object	DK:	Must specify	.OBJ
Listing	Same as for object file	Must specify	.LST
Cref	DK:	Must specify	.TMP
First source	DK:	Must specify	.MAC
Additional source	Same as for preceding source file	Must specify	.MAC
System MACRO Library	System device SY:	SYSMAC	.SML
User MACRO Library	DK: if first file, otherwise same as for preceding source file	Must specify	.MAC

NOTE

Some assemblies need more symbol table space than available memory can contain. When this occurs the system automatically creates a temporary work file called WRK.TMP to provide extended symbol table space.

The default device for WRK.TMP is DK. To cause the system to assign a different device, enter the following command:

```
.ASSIGN dev: WF
```

where: dev is the file-structured device that will hold WRK.TMP.

9.4 FILE SPECIFICATION OPTIONS

At assembly time you may need to override certain MACRO directives appearing in the source programs. You may also need to direct MACRO-11 on the handling of certain files during assembly. You can satisfy these needs by using the switches described in Table 9-2.

Table 9-2
File Specification Options

Option	Usage
/L:arg /N:arg	Listing control switches; these options accept ASCII switch values (arg) which are equivalent in function and name to the arguments of the .LIST and .NLIST directives specified in the source program (see Section 6.1.1). This switch overrides the arguments of the directives and remains in effect for the entire assembly process.
/E:arg /D:arg	Function control switches; these options accept ASCII switch values (arg) which are equivalent in function and name to the arguments of the .ENABL and .DSABL directives specified in the source program (see Section 6.2). This switch overrides the arguments of the directives and remains in effect for the entire assembly process.
/M	Indicates input file is MACRO library file. When the assembler encounters an .MCALL directive in the source code, it searches macro libraries according to their order of appearance in the command string. When it locates a macro record whose name matches that given in the .MCALL, it assembles the macro as indicated by that definition. Thus, if two or more macro libraries contain definitions of the same macro name, the macro library that appears leftmost in the command string takes precedence.

(continued on next page)

Table 9-2 (Cont.)
File Specification Options

Option	Usage
/M (cont.)	Consider the following command string: *(output file specification)=ALIB.MAC/M, BLIB.MAC/M,XIZ Assume that each of the two macro libraries, ALIB and BLIB, contain a macro called .BIG, but with different definitions. Then, if source file XIZ contains a macro call .MCALL .BIG, the system includes the definition of .BIG in the program as it appears in the macro library ALIB. If the command string does not include the standard system macro library SYSMAC.SML, the system automatically includes it as the last source file in the command string. Therefore, if macro library ALIB contains a definition of a macro called .READ, that definition of .READ overrides the standard .READ macro definition in SYSMAC.SML.
/C:arg	Controls contents of cross-reference listing.
/P:1	Assembles the associated file during assembly pass 1 only.
/P:2	Assembles the associated file during assembly pass 2 only.

The /M and /P switches affect only the source file to which they are appended. The other options affect the entire command string.

9.5 CROSS-REFERENCE (CREF) TABLE GENERATION OPTION

A cross-reference (CREF) table lists all or a subset of the symbols in a source program, identifying the statements that define and use symbols.

9.5.1 Obtaining a Cross-Reference Table

To obtain a CREF table you must include the /C:arg option in the command string. Usually you include the /C:arg option with the assembly listing file specification.

If the command string does not include a cref file specification, the system automatically generates a temporary file on device DK:. If you need to have a device other than DK: contain the temporary cref file, you must include the dev:cref field in the command string.

A complete CREF listing contains the following six sections:

1. A cross reference of program symbols--labels used in the program and symbols followed by an operator.

2. A cross reference of register equate symbols--symbols defined in the program by the construct:

   symbol-n

   with 0>n>7.

   Normally, these symbols include R0, R1, R2, R3, R4, R5, SP, and PC.

3. A cross reference of MACRO symbols--those symbols defined by .MACRO and .MCALL directives.

4. A cross reference of permanent symbols--all operation mnemonics and assembler directives.

5. A cross reference of program sections--the names you specify as operands of .CSECT or .PSECT directives.

6. A cross reference of errors--the system groups and lists all flagged errors from the assembly by error type.

You can include any or all of these six sections on the cross-reference listing by specifying the appropriate arguments with the /C option. These arguments are listed and described in Table 9-3.

Table 9-3
/C Option Arguments

Argument	CREF Section
S	User defined symbols
R	Register symbols
M	MACRO symbolic names
P	Permanent symbols including instructions and directives
C	Control and program sections
E	Error code grouping

NOTE

Specifying /C with no arguments is equivalent to specifying /C:S:M:E. That special case excepted, you must explicitly request each CREF section by including its arguments. No cross-reference file occurs if the /C option is not specified, even if the command string includes a CREF file specification.

9.5.2 Handling Cross-Reference Table Files

When you request a cross-reference listing by means of the /C option, you cause the system to generate a temporary file, DK:CREF.TMP.

If device DK: is write-locked or if it contains insufficient free space for the temporary file, you can allocate another device for the file. To allocate another device, specify a third output file in the command string; that is, include a dev:cref specification. (You must still include the /C option to control the form and content of the listing. The dev:cref specification is ignored if the /C option is not also present in the command string.)

The system then uses the dev:cref file instead of DK:CREF.TMP and deletes it automatically after producing the CREF listing.

The following command string causes the system to use RK2:TEMP.TMP as the temporary CREF file.

```
*,LP:,RK2:TEMP.TMP=SOURCE/C
```

Another way to assign an alternative device for the CREF.TMP file is to enter the following command prior to entering R MACRO:

```
.ASSIGN dev:CF
```

This method is preferred if you intend to do several assemblies, as it relieves you from having to include the dev:cref specification in each command string. If you enter the ASSIGN dev: CF command, and later include a cref specification in a command string, the specification in the command string prevails for that assembly only.

The system lists requested cross-reference tables following the MACRO assembly listing. Each table begins on a new page.

The system prints symbols and also symbol values, control sections, and error codes, if applicable, beginning at the left margin of the page. References to each symbol are listed on the same line, left-to-right across the page. The system lists references in the form P-L; where P is the page in which the symbol, control section, or error code appears, and L is the line number on the page.

A number sign (#) next to a reference indicates a symbol definition. An asterisk (*) next to a reference indicates a destructive reference--an operation that alters the contents of the addressed location.

9.5.3 MACRO-11 Error Messages Under RT-11

MACRO-11 outputs an error message to the command output device when one of the error conditions described below is detected. These error messages reflect operational problems and should not be confused with the error codes (see Appendix D) produced by MACRO-11 during assembly.

Error Message	Meaning
?MACRO-F-Bad option	The specified option was not recognized by the program.
?MACRO-F-Device full	The output volume does not have sufficient room for an output file specified in the command string.
?MACRO-F-File not found	An input file in the command line does not exist on the specified device.
?MACRO-F-Illegal command	The command line contains a syntax error or specifies more than 6 input files.
?MACRO-F-Illegal device	A device specified in the command line does not exist on the system.
?MACRO-F-Insufficient memory	There were too many symbols in the program being assembled.
?MACRO-W-I/O error on cref file: cref file aborted	MACRO ran out of device space while writing the cref file, or a hardware error has occurred. The cref file is aborted but assembly continues.
?MACRO-F-I/O error on dev:filenm.ext	A hardware error occurred while attempting to read from or write to the device on the specified channel.
?MACRO-F-I/O error on work file	MACRO failed to open, read or write to its work file, WRK.TMP.
?MACRO-F-Invalid macro library	The library file has been corrupted or it was not produced by the RT-11 librarian, LIBR.
?MACRO-F-Output device full	There was no room to continue writing the output file.
?MACRO-F-Read error on MACRO library	MACRO detected a bad record in the MACRO library. For example, this error occurs when the library area is bad.
?MACRO-F-Storage limit exceeded (64K)	MACRO's Virtual Symbol Table can store symbols and macros up to 64K in any combination. Your program contains more than 64K worth of one or both of these elements.

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/P (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).

EVEN PARITY BIT	7-BIT OCTAL CODE	CHARACTER	REMARKS
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	
0	111	I	

MACRO-11 CHARACTER SETS

EVEN PARITY BIT	7-BIT OCTAL CODE	CHARACTER	REMARKS
1	112	J	
0	113	K	
1	114	L	
0	115	M	
0	116	N	
1	117	O	
0	120	P	
1	121	Q	
1	122	R	
0	123	S	
1	124	T	
0	125	U	
0	126	V	
1	127	W	
1	130	X	
0	131	Y	
0	132	Z	
1	133	[	shift/k.
0	134	\	shift/l.
1	135	]	shift/m.
1	136	^	*
0	137	_	**
0	140	`	Accent grave.
1	141	a	
1	142	b	
0	143	c	
1	144	d	
0	145	e	
0	146	f	
1	147	g	
1	150	h	
0	151	i	
0	152	j	
1	153	k	
0	154	l	
1	155	m	
1	156	n	
0	157	o	
1	160	p	
0	161	q	
0	162	r	
1	163	s	
0	164	t	
1	165	u	
1	166	v	
0	167	w	
0	170	x	
1	171	y	
1	172	z	
0	173	{	
1	174	|	
0	175	}	This code generated by ALTMODE.
0	176	~	This code generated by prefix key (if present).
1	177	DEL	Delete, Rubout.

* ^ Appears as # or ^ on some machines.

** _ Appears as < on some machines.

A.2 RADIX-50 CHARACTER SET

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

The maximum Radix-50 value is, 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	
Space	000000	Space	000000	Space	000000
A	003100	A	000050	A	000001
B	006200	B	000120	B	000002
C	011300	C	000170	C	000003
D	014400	D	000240	D	000004
E	017500	E	000310	E	000005
F	022600	F	000360	F	000006
G	025700	G	000430	G	000007
H	031000	H	000500	H	000010
I	034100	I	000550	I	000011
J	037200	J	000620	J	000012
K	042300	K	000670	K	000013
L	045400	L	000740	L	000014
M	050500	M	001010	M	000015
N	053600	N	001060	N	000016
O	056700	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

MACRO-11 CHARACTER SETS

Single Char. or First Char.		Second Character		Third Character	
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 DIRECTIVES

B.1 SPECIAL CHARACTERS

Character	Function
:	Label terminator
=	Direct assignment operator
_	Register term indicator
tab	Item terminator or field terminator
space	Item terminator or 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 or concatenation indicator
.	Assembly location counter
<	Initial argument indicator
>	Terminal argument indicator
^	Universal unary operator or argument indicator
\	Macro call numeric argument indicator
vertical tab	Source line terminator

B.2 SUMMARY OF ADDRESS MODE SYNTAX

Symbols used in the table:

n is an integer, 0 to 7, representing a register number

R is a register expression

E is an expression

ER is either a register expression or an expression whose value is in the range 0 to 7.

Format	Address Mode Name	Address Mode Number	Meaning
R	Register	0n	Register R contains the operand.
@R or (ER)	Register deferred	1n	Register R contains the address of the operand.
(ER)+	Autoincrement	2n	The contents of the register specified as (ER) are incremented after being used as the address of the operand.
@(ER)+	Autoincrement Deferred	3n	The register specified as (ER) contains the pointer to the address of the operand; the register (ER) is incremented after use.
-(ER)	Autodecrement	4n	The contents of the register specified as (ER) are decremented before being used as the address of the operand.
@-(ER)	Autodecrement Deferred	5n	The contents of the register specified as (ER) are decremented before being used as the pointer to the address of the operand.
E(ER)	Index	6n	The expression E, plus the contents of the register specified as (ER), form the address of the operand.
@E(ER)	Index Deferred	7n	The expression E, plus the contents of the register specified as (ER), yield a pointer to the address of the operand.
#E	Immediate	27	The expression E is the operand itself.
@#E	Absolute	37	The expression E is the address of the operand.
E	Relative	67	The address of the operand E, relative to the instruction, follows the instruction.
@E	Relative Deferred	77	The address of the operand is pointed to by E whose address, relative to the instruction, follows the instruction.

B.3 ASSEMBLER DIRECTIVES

The MACRO-11 assembler directives are summarized in the following table. For a detailed description of each directive, the table contains references to the appropriate sections in the body of the manual.

Form	Section Reference	Operation
'	6.3.3 7.3.7	Followed by one ASCII character a single quote (apostrophe) generates a word which contains the 7-bit ASCII representation of the character in the low-order byte and zero in the high-order byte. This character is also used as a concatenation indicator in the expansion of macro arguments.
"	6.3.3	Followed by two ASCII characters a double quote generates a word which contains the 7-bit ASCII representation of the two characters. The first character is stored in the low-order byte; the second character is stored in the high-order byte.
^Bn	6.4.1.2	A temporary radix control, causes the value n to be treated as a binary number.
^Cexpr	6.4.2.2	A temporary numeric control, causes the expression's value to be ones-complemented.
^Dn	6.4.1.2	A temporary radix control, causes the value n to be treated as a decimal number.
^Fn	6.4.2.2	A temporary numeric control, causes the value n to be treated as a sixteen-bit floating-point number.
^On	6.4.1.2	A temporary radix control, causes the value n to be treated as an octal number.
^Rccc	6.3.7	Converts ccc to Radix-50 form.
.ASCII /string/	6.3.4	Generates a block of data containing the ASCII equivalent of the character string (enclosed in delimiting characters), one character per byte.

Form	Section Reference	Operation
.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 terminating the specified string.
.ASECT	6.7.2	Begins or resumes the absolute program section.
.BLKB exp	6.5.3	Reserves a block of storage space whose length in bytes is determined by the specified expression.
.BLKW exp	6.5.3	Reserves a block of storage space whose length in words is determined by the specified expression.
.BYTE expl,exp2,..	6.3.1	Generates successive bytes of data; each byte contains the value of the corresponding specified expression.
.CSECT [name]	6.7.2	Begins or resumes named or unnamed relocatable program section. This directive is provided for compatibility with other PDP-11 assemblers.
.DSABL arg	6.2	Disables the function specified by the argument.
.ENABL arg	6.2	Enables (invokes) the function specified by the argument.
.END [exp]	6.6.1	Indicates the logical end of the source program. The optional argument specifies the transfer address where program execution is to begin.
.ENDC	6.9.1	Indicates the end of a conditional assembly block.
.ENDM [name]	7.1.2	Indicates the end of the current repeat block, indefinite repeat block, or macro definition. The optional name, if used, must be identical to the name specified in the macro definition.
.ENDR	7.7	Indicates the end of the current repeat block. This directive is provided for compatibility with other PDP-11 assemblers.

Form	Section Reference	Operation
.EOT	6.6.2	Ignored, indicates end-of-tape which is detected automatically by the hardware. It is included for compatibility with earlier assemblers.
.ERROR exp;text	7.5	A user-invoked error directive, causes output to the listing file or the command output device containing the optional expression and the statement containing the directive.
.EVEN	6.5.1	Ensures that the current location counter contains an even address by adding 1 if it is odd.
.FLT2 arg1,arg2,...	6.4.2.1	Generates successive 2-word floating-point equivalents for the floating-point numbers specified as arguments.
.FLT4 arg1,arg2,...	6.4.2.1	Generates successive 4-word floating-point equivalents for the floating-point numbers specified as arguments.
.GLOBL sym1,sym2,...	6.8	Defines the symbol specified as global symbol.
.IDENT /string/	6.1.5	Provides a means of labeling the object module with the program version number. The version number is the Radix-50 string appearing between the paired delimiting characters.
.IF cond,arg1	6.9.1	Begins a conditional assembly block of source code which is included in the assembly only if the stated condition is met with respect to the argument(s) specified.
.IFF	6.9.2	Appears only within a conditional assembly block, indicating the beginning of a section of code to be assembled if the condition upon entering the block tests false.
.IFT	6.9.2	Appears only within a conditional assembly block, indicating the beginning of a section of code to be assembled if the condition upon entering the block tests true.
.IFTF	6.9.2	Appears only within a conditional assembly block, indicating the beginning of a section of code to be assembled unconditionally.

Form	Section Reference	Operation
.IIF cond,arg, statement	6.9.3	Acts as a 1-line conditional assembly block where the condition is tested for the argument specified. The statement is assembled only if the condition tests true.
.IRP sym, <arg1,arg2,...>	7.6.1	Indicates the beginning of an indefinite repeat block in which the symbol specified is replaced with successive elements of the real argument list enclosed within angle brackets.
.IRPC sym,<string>	7.6.2	Indicates the beginning of an indefinite repeat block in which the specified symbol takes on the value of successive characters, optionally enclosed within angle brackets.
.LIMIT	6.5.4	Reserves two words into which the Task Builder inserts the low and high addresses of the task image.
.LIST [arg]	6.1.1	Without an argument, the .LIST directive increments the listing level count by 1. With an argument, this directive does not alter the listing level count, but formats the assembly listing according to the argument specified.
.MACRO name,arg1, arg2,...	7.1.1	Indicates the start of a macro definition having the specified name and the following dummy arguments.
.MCALL arg1,arg2,...	7.8	Specifies the symbolic names of the user or system macro definitions required in the assembly of the current user program, but which are not defined within the program.
.MEXIT	7.1.3	Causes an exit from the current macro expansion or indefinite repeat block.
.NARG symbol	7.4.1	Appearing only within a macro definition, equates the specified symbol to the number of arguments in the macro call currently being expanded.
.NCHR symbol,<string>	7.4.2	Appearing anywhere in a source program, equates the symbol specified to the number of characters in the specified string.

Form	Section Reference	Operation
.NLIST [arg]	6.1.1	Without an argument, decrements the listing level count by 1. With an argument, this directive suppresses that portion of the listing specified by the argument.
.NTYPE symbol,aexp	7.4.3	Appearing only within a macro definition, equates the symbol to the 6-bit addressing mode of the specified address expression.
.ODD	6.5.2	Ensures that the current location counter contains an odd address by adding 1 if it is even.
.PACKED	6.3.8	Causes a decimal number of 31(10) digits or less to be packed 2 digits per byte.
.PAGE	6.1.6	Causes the assembly listing to skip to the top of the next page and to increment the page count.
.PRINT exp;text	7.5	User-invoked message directive; causes output to the listing file or the command output device containing the optional expression and the statement containing the directive.
.PSECT name,attl,... attn	6.7.1	Begins or resumes a named or unnamed program section having the specified attributes.
.RADIX n	6.4.1.1	Alters the current program radix to n, where n is 2, 8, or 10.
.RAD50 /string/	6.3.6	Generates a block of data containing the Radix-50 equivalent of the character string enclosed within delimiting characters.
.REPT exp	7.7	Begins a repeat block; causes the section of code up to the next .ENDM or .ENDR directive to be repeated the number of times specified as exp.
.RESTORE	6.7.4	Retrieves a previously .SAVEd program section from the top of the program section context stack leaving the current program section in effect.

Form	Section Reference	Operation
.SAVE	6.7.3	Stores the current program section on the top of the program section context stack leaving the current program section in effect.
.SBTTL string	6.1.4	Causes the specified string to be printed as part of the assembly listing page header. The string component of each .SBTTL directive is collected into a table of contents at the beginning of the assembly listing.
.TITLE string	6.1.3	Assigns the first six Radix-50 characters in the string as an object module name and causes the string to appear on each page of the assembly listing.
.WORD expl,exp2,..	6.3.2	Generates successive words of data; each word contains the value of the corresponding specified expression.

APPENDIX C

PERMANENT SYMBOL TABLE (PST)

The mnemonics for the PDP-11 operation (op) codes and MACRO-11 assembler directives are stored in the Permanent Symbol Table (PST). The PST contains the symbols that are automatically recognized by MACRO-11.

For a detailed description of the op codes, see the PDP-11 Processor Handbook.

C.1 OP CODES

Instruction Mnemonic	Octal Value	Operation
ADC	005500	Add Carry
ADCB	105500	Add Carry (Byte)
ADD	060000	Add Source To Destination
ASH	072000	Shift Arithmetically
ASHC	073000	Arithmetic Shift Combined
ASL	006300	Arithmetic Shift Left
ASLB	106300	Arithmetic Shift Left (Byte)
ASR	006200	Arithmetic Shift Right
ASRB	106200	Arithmetic Shift Right (Byte)
BCC	103000	Branch If Carry Is Clear
BCS	103400	Branch If Carry Is Set
BEQ	001400	Branch If Equal
BGE	002000	Branch If Greater Than Or Equal
BGT	003000	Branch If Greater Than
BHI	101000	Branch If Higher
BHIS	103000	Branch If Higher Or Same
BIC	040000	Bit Clear
BICB	140000	Bit Clear (Byte)
BIS	050000	Bit Set
BISB	150000	Bit Set (Byte)
BIT	030000	Bit Test
BITB	130000	Bit Test (Byte)
BLE	003400	Branch If Less Than Or Equal
BLO	103400	Branch If Lower
BLOS	101400	Branch If Lower Or Same
BLT	002400	Branch If Less Than
BMI	100400	Branch If Minus
BNE	001000	Branch If Not Equal
BPL	100000	Branch If Plus
BPT	000003	Breakpoint Trap
BR	000400	Branch Unconditional
BVC	102000	Branch If Overflow Is Clear
BVS	102400	Branch If Overflow Is Set

PERMANENT SYMBOL TABLE (PST)

Instruction Mnemonic	Octal Value	Operation
CALL	004700	Jump To Subroutine (JSR PC,xxx)
CCC	000257	Clear All Condition Codes
CLC	000241	Clear C Condition Code Bit
CLN	000250	Clear N Condition Code Bit
CLR	005000	Clear Destination
CLRB	105000	Clear Destination (Byte)
CLV	000242	Clear V Condition Code Bit
CLZ	000244	Clear Z Condition Code Bit
CMP	020000	Compare Source To Destination
CMPB	120000	Compare Source To Destination (Byte)
COM	005100	Complement Destination
COMB	105100	Complement Destination (Byte)
DEC	005300	Decrement Destination
DECB	105300	Decrement Destination (Byte)
DIV	071000	Divide
EMT	104000	Emulator Trap
FADD	075000	Floating Add
FDIV	075030	Floating Divide
FMUL	075020	Floating Multiply
FSUB	075010	Floating Subtract
HALT	000000	Halt
INC	005200	Increment Destination
INCB	105200	Increment Destination (Byte)
IOT	000004	Input/Output Trap
JMP	000100	Jump
JSR	004000	Jump To Subroutine
MARK	006400	Mark
MED6X	076600	PDP-11/60 Maintenance
MED74C	076601	PDP-11/74 CIS Maintenance
MFPI	006500	Move From Previous Instruction Space
MFPS	106700	Move from PS (LSI-11, LSI-11/23, LSI-11/2)
MFPT	000007	Move From Processor Type
MOV	010000	Move Source To Destination
MOVB	110000	Move Source To Destination (Byte)
MTPI	006600	Move To Previous Instruction Space
MTPS	106400	Move to PS (LSI-11, LSI-11/23, LSI-11/2)
MUL	070000	Multiply
NEG	005400	Negate Destination
NEGB	105400	Negate Destination (Byte)
NOP	000240	No Operation
RESET	000005	Reset External Bus
RETURN	000207	Return From Subroutine (RTS PC)
ROL	006100	Rotate Left
ROLB	106100	Rotate Left (Byte)
ROR	006000	Rotate Right
RORB	106000	Rotate Right (Byte)
RTI	000002	Return From Interrupt (Permits a trace trap)

Instruction Mnemonic	Octal Value	Operation
RTS	000200	Return From Subroutine
RTT	000006	Return From Interrupt (inhibits trace trap)
SBC	005600	Subtract Carry
SBCB	105600	Subtract Carry (Byte)
SCC	000277	Set All Condition Code Bits
SEC	000261	Set C Condition Code Bit
SEN	000270	Set N Condition Code Bit
SEV	000262	Set V Condition Code Bit
SEZ	000264	Set Z Condition Code Bit
SOB	077000	Subtract One And Branch
SUB	160000	Subtract Source From Destination
SWAB	000300	Swap Bytes
SXT	006700	Sign Extend
TRAP	104400	Trap
TST	005700	Test Destination
TSTB	105700	Test Destination (Byte)
WAIT	000001	Wait For Interrupt
XFC	076700	Extended Function Code
XOR	074000	Exclusive OR

COMMERCIAL INSTRUCTION SET (CIS) OP CODES

Every operation listed in the CIS table has two instruction mnemonics. The suffix "I", attached to every second mnemonic, indicates that the addresses are inline. The inline instructions require two arguments; the other instructions (excepting L2DN and L3DN) require no arguments.

Instruction Mnemonic	Octal Value	Operation
ADDN	076050	Add Numeric
ADDNI	076150	Add Numeric
ADDP	076070	Add Packed
ADDPI	076170	Add Packed
ASHN	076056	Arithmetic Shift Numeric
ASHNI	076156	Arithmetic Shift Numeric
ASHP	076076	Arithmetic Shift Packed
ASHPI	076176	Arithmetic Shift Packed
CMPC	076044	Compare Character String
CMPCI	076144	Compare Character String
CMPN	076052	Compare Numeric
CMPNI	076152	Compare Numeric
CMPP	076072	Compare Packed
CMPPI	076172	Compare Packed
CVTLN	076057	Convert Long to Numeric
CVTLNI	076157	Convert Long to Numeric
CVTLP	076077	Convert Long to Packed
CVTLPI	076177	Convert Long to Packed
CVTNP	076055	Convert Numeric to Packed
CVTNPI	076155	Convert Numeric to Packed
CVTPN	076054	Convert Packed to Numeric
CVTPNI	076154	Convert Packed to Numeric
DIVP	076075	Divide Decimal
DIVPI	076175	Divide Decimal
LOCC	076040	Locate Character
LOCCI	076140	Locate Character

Instruction Mnemonic	Octal Value	Operation
L2DN*	07602N	Load 2 Descriptors @(RN)+
L3DN*	07606N	Load 3 Descriptors @(RN)+
MATC	076045	Match Character
MATCI	076145	Match Character
MOVC	076030	Move Character
MOVCI	076130	Move Character
MOVRC	076031	Move Reverse Justified Character
MOVRCI	076131	Move Reverse Justified Character
MOVTC	076032	Move Translated Character
MOVTCI	076132	Move Translated Character
MULP	076074	Multiply Decimal
MULPI	076174	Multiply Decimal
SCANC	076042	Scan Character
SCANCI	076142	Scan Character
SKPC	076041	Skip Character
SKPCI	076141	Skip Character
SPANC	076043	Span Character
SPANCI	076143	Span Character
SUBN	076051	Subtract Numeric
SUBNI	076151	Subtract Numeric
SUBP	076071	Subtract Packed
SUBPI	076171	Subtract Packed

FLOATING POINT PROCESSOR OP CODES

Instruction Mnemonic	Octal Value	Operation
ABSD	170600	Make Absolute Double
ABSF	170600	Make Absolute Floating
ADDD	172000	Add Double
ADDF	172000	Add Floating
CFCC	170000	Copy Floating Condition Codes
CLRD	170400	Clear Double
CLRF	170400	Clear Floating
CMPD	173400	Compare Double
CMPF	173400	Compare Floating
DIVD	174400	Divide Double
DIVF	174400	Divide Floating
LDCDF	177400	Load And Convert From Double To Floating
LDCFD	177400	Load And Convert From Floating To Double
LDCID	177000	Load And Convert Integer To Double
LDCIF	177000	Load And Convert Integer To Floating
LDCLD	177000	Load And Convert Long integer To Double

* where N=0...7

Instruction Mnemonic	Octal Value	Operation
LDCLF	177000	Load And Convert Long Integer To Floating
LDD	172400	Load Double
LDEXP	176400	Load Exponent
LDF	172400	Load Floating
LDFPS	170100	Load FPPs Program Status
MFPD	106500	Move From Previous Data Space
MODD	171400	Multiply And Integerize Double
MODF	171400	Multiply And Integerize Floating
MTPD	106600	Move To Previous Data Space
MULD	171000	Multiply Double
MULF	171000	Multiply Floating
NEGD	170700	Negate Double
NEGF	170700	Negate Floating
SETD	170011	Set Double Mode
SETF	170001	Set Floating Mode
SETI	170002	Set Integer Mode
SETL	170012	Set Long Integer Mode
SPL	000230	Set Priority Level
STA0	170005	Diagnostic Floating Point
STB0	170006	Diagnostic Floating Point
STCDF	176000	Store And Convert From Double To Floating
STCDI	175400	Store And Convert From Double To Integer
STCDL	175400	Store And Convert From Double To Long Integer
STCFD	176000	Store And Convert From Floating To Double
STCFI	175400	Store And Convert From Floating To Integer
STCFL	175400	Store And Convert From Floating To Long Integer
STD	174000	Store Double
STEXP	175000	Store Exponent
STF	174000	Store Floating
STFPS	170200	Store FPPs Program Status
STST	170300	Store FPPs Status
SUBD	173000	Subtract Double
SUBF	173000	Subtract Floating
TSTD	170500	Test Double
TSTF	170500	Test Floating

C.2 MACRO-11 DIRECTIVES

The MACRO-11 directives that follow are described in greater detail in Appendix B.

Directive	Function
.ASCII	Translates character string to ASCII equivalents.
.ASCIZ	Translates character string to ASCII equivalents; inserts zero byte as last character.
.ASECT	Begins absolute program section (provided for compatibility with other PDP-11 assembliers).
.BLKB	Reserves byte block in accordance with value of specified argument.
.BLKW	Reserves word block in accordance with value of specified argument.
.BYTE	Generates successive byte data in accordance with specified arguments.
.CSECT	Begins relocatable program section (provided for compatibility with other PDP-11 assemblers).
.DSABL	Disables specified function.
.ENABL	Enables specified function.
.END	Defines logical end of source program.
.ENDC	Defines end of conditional assembly block.
.ENDM	Defines end of macro definition, repeat block, or indefinite repeat block.
.ENDR	Defines end of current repeat block (provided for compatibility with other PDP-11 assemblers).
.EOT	Define End of Tape condition (ignored).
.ERROR	Outputs diagnostic message to listing file or command output device.
.EVEN	Word-aligns the current location counter.
.FLT2	Causes two words of storage to be generated for each floating-point argument.
.FLT4	Causes four words of storage to be generated for each floating-point argument.
.GLOBL	Declares global attribute for specified symbol(s).
.IDENT	Labels object module with specified program version number.
.IF	Begins conditional assembly block.
.IFF	Begins subconditional assembly block (if conditional assembly block test is false).
.IFT	Begins subconditional assembly block (if conditional assembly block test is true).
.IFTF	Begins subconditional assembly block (whether conditional assembly block test is true or false).
.IIF	Assembles immediate conditional assembly statement (if specified condition is satisfied).
.IRP	Begins indefinite repeat block; replaces specified symbol with specified successive real arguments.
.IRPC	Begins indefinite repeat block; replaces specified symbol with value of successive characters in specified string.
.LIMIT	Reserves two words of storage for high and low addresses of task image.
.LIST	Controls listing level count and format of assembly listing. .MACRO Denotes start of macro definition.
.MCALL	Identifies required macro definition(s) for assembly.

Directive	Function
.MEXIT	Exit from current macro definition or indefinite repeat block.
.NARG	Equates specified symbol to the number of arguments in the macro expansion.
.NCHR	Equates specified symbol to the number of characters in the specified character string.
.NLIST	Controls listing level count and suppresses specified portions of the assembly listing.
.NTYPE	Equates specified symbols to the addressing mode of the specified argument.
.ODD	Byte-aligns the current location counter.
.PACKED	Generates packed decimal data, 2 digits per byte.
.PAGE	Advances form to top of next page.
.PRINT	Prints specified message on command output device.
.PSECT	Begins specified program section having specified attributes.
.RADIX	Changes current program radix to specified radix.
.RAD50	Generates data block having Radix-50 equivalents of specified character string.
.REPT	Begins repeat block and replicates it according to the value of the specified expression.
.RESTORE	Stores the current program section context on the top of the program section context stack.
.SAVE	Retrieves the program section from the top of the program section context stack.
.SBTTL	Prints specified subtitle text as the second line of the assembly listing page header.
.TITLE	Prints specified title text as object module name in the first line of the assembly listing page header.
.WORD	Generates successive word data in accordance with specified arguments.

APPENDIX D

ERROR MESSAGES

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

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

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

Error Code	Meaning
A	Assembly error. Because many different conditions produce this error message, the directives which may yield a general assembly error have been categorized below to reflect these error conditions: CATEGORY 1: ILLEGAL ARGUMENT SPECIFIED. .RADIX -- A value other than 2, 8, or 10 is specified as a new radix. .LIST/.NLIST -- Other than a legally defined argument (see Table 6-2) is specified with the directive. .ENABL/.DSABL -- Other than a legally defined argument (see Table 6-3) is specified with the directive, or the attribute arguments of a previously declared program section .PSECT -- Other than a legally-defined argument (see Table 6-4) is specified with the directive, or the attribute arguments of a previously declared program section change (see Section 6.7.1.1). .IF/.IIF -- Other than a legally defined conditional test (see Table 6-6) or an illegal argument expression value is specified with the directive. .MACRO -- An illegal or duplicate symbol found in dummy argument list.

Error Code	Meaning
A (cont.)	.TITLE -- Program name is not specified in the directive, or first non-blank character following the directive is a non-Radix-50 character. .IRP/.IRPC -- No dummy argument is specified in the directive. .NARG/ .NCHR/ .NTYPE -- No symbol is specified in the directive. .IF/.IIF -- No conditional argument is specified in the directive. CATEGORY 3: UNMATCHED DELIMITER/ILLEGAL ARGUMENT CONSTRUCTION. .ASCII/.ASCIZ/.RAD50/.IDENT -- Character string or argument string delimiters do not match, or an illegal character is used as a delimiter, or an illegal argument construction is used in the directive. .NCHAR -- Character string delimiters do not match, or an illegal character is used as a delimiter in the directive. CATEGORY 4: GENERAL ADDRESSING ERRORS. This type of error results from one of several possible conditions: 1. Permissible range of a branch instruction (from -128(10) to +127(10) words) has been exceeded. 2. A statement makes invalid use of the current location counter. For example, a ".=expression" statement attempts to force the current location counter to cross program section (.PSECT) boundaries. 3. A statement contains an invalid address expression: In cases where an absolute address expression is required, specifying a global symbol, a relocatable value, or a complex relocatable value (see Section 3.9) results in an invalid address expression. If an undefined symbol is made a default global reference by the .ENABL GBL directive (see Section 6.2) during pass1, any attempt to redefine the symbol during pass 2 will result in an invalid address expression.

Error Code	Meaning
A (cont.)	In cases where a relocatable address expression is required, either a relocatable or absolute value is permissible, but a global symbol or a complex relocatable value in the statement results in an invalid address expression. For example: .BLKB/.BLKW/.REPT -- Other than an absolute value or an expression which reduces to an absolute value has been specified with the directive. 4. Multiple expressions are not separated by a comma. This condition causes the next symbol to be evaluated as part of the current expression. 5. .SAVE -- The stack is full when the .SAVE directive is issued. 6. .RESTORE -- The stack is empty when the .RESTORE directive is issued. CATEGORY 5: ILLEGAL FORWARD REFERENCE. This type of error results from either of two possible conditions: 1. A global assignment statement (symbol==expression or symbol==:expression*) contains a forward reference to another symbol. 2. An expression defining the value of the current location counter contains a forward reference.
B	Bounding error. Instructions or word data are being assembled at an odd address. The location counter is incremented by 1.
D	Doubly-defined symbol referenced. Reference was made to a symbol which is defined more than once.
E	End directive not found. When the end-of-file is reached during source input and the .END directive has not yet been encountered, MACRO-11 generates this error code, ends assembly pass 1, and proceeds with assembly pass 2. Also caused by assembler-stack overflow. In this case MACRO-11 will place a question mark (?) into the line at the point where the overflow occurred.
I	Illegal character detected. Illegal characters which are also non-printable are replaced by a question mark (?) on the listing. The character is then ignored.

* RT-11 V4.0 only.

Error Code	Meaning
L	Input line is greater than 132(10) characters in length. Currently, this error condition is caused only during macro expansion when longer real arguments, replacing the dummy arguments, cause a line to exceed 132(10) characters.
M	Multiple definition of a label. A label was encountered which was equivalent (in the first six characters) to a label previously encountered.
N	A number contains a digit that is not in the current program radix. The number is evaluated as a decimal value.
O	Opcode error. Directive out of context. Permissible nesting level depth for conditional assemblies has been exceeded. Attempt to expand a macro which was unidentified after .MCALL search.
P	Phase error. A label's definition of value varies from one assembly pass to another or a multiple definition of a local symbol has occurred within a local symbol block. Also, when in a local symbol block defined by the .ENABL LSB directive, an attempt has occurred to define a local symbol in a program section other than that which was in effect when the block was entered. An error code P also appears if an .ERROR directive is assembled.
Q	Questionable syntax. Arguments are missing, too many arguments are specified, or the instruction scan was not completed.
R	Register-type error. An invalid use of or reference to a register has been made, or an attempt has been made to redefine a standard register symbol without first issuing the .DSABL REG directive.
T	Truncation error. A number generated more than 16 bits in a word, or an expression generated more than 8 significant bits during the use of the .BYTE directive or trap (EMT or TRAP) instruction.
U	Undefined symbol. An undefined symbol was encountered during the evaluation of an expression; such an undefined symbol is assigned a value of zero. Other possible conditions which result in this error code include unsatisfied macro names in the list of .MCALL arguments and a direct assigment (symbol=expression or symbol=:expression) statement which contains a forward reference to a symbol whose definition also contains a forward reference; also, a local symbol may have been referenced that does not exist in the current local symbol block.
Z	Instruction error. The instruction so flagged is not compatible among all members of the PDP-11 family. See Section 5.3 for details.

APPENDIX E

SAMPLE CODING STANDARD

Local user requirements must be met in a coding standard, but following this model as closely as possible helps you and DIGITAL by simplifying communication and software maintenance. Remember that this is a sample and may not entirely apply to your system.

E.1 LINE FORMAT

Source lines are from one to eighty characters in length with the following format:

1. Label Field - if present, begins in column 1
2. Operation field - begins in column 9 (tab stop 1)
3. Operand field - begins in column 17 (tab stop 2)
4. Comment field - begins in column 33 (tab stop 4). If the operand field extends beyond column 33 (tab stop 4) leave a space and start the comment.

E.2 COMMENTS

To make the program easier to understand, comments should be used to explain the logic behind the instructions. In general this will consist of a comment per line of code. However, if a particularly difficult or obscure section of code is used, precede that section with a longer explanation.

Comments that are too long for the comment field may be continued on the following line. Begin the new line with a semicolon, space over to the column the comment began in and continue writing.

If a lengthy text is needed for an explanation, begin the comment with a line containing only the characters ;+ and end it with a line containing only the characters ;-. The lines between these delimiters should each begin with a semicolon and a space. 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 Registers

E.3.1.1 **General Purpose Registers** - Use the default name:

```
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)
```

NOTE

These register names will be defined within the assembler; other standard symbols must be put in a file and linked with the program.

E.3.1.2 **Hardware Registers** - Use the hardware definition. For example, PS (Program Status Register) and SWR (Switch Register).

E.3.1.3 **Device Registers** - Use the hardware notation. For example, the control status register for the RK disk is RKCS.

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.

E.3.3 Symbols*

The following chart diagrams the syntax of the 5 major types of symbol names:

symbol	pos-1	pos-2	pos-3	pos-4	pos-5	pos-6	length
non-global symbol	letter	a-num/ null	a-num/ null	a-num/ null	a-num/ null	a-num/ null	>=1
global symbol	$/. ***	a-num null	a-num/ null	a-num/ null	a-num/ null	a-num/ null	>=1
global offset	letter	$/. ***	a-num	a-num/ null	a-num/ null	a-num/ null	>=3
global bit pattern	letter	a-num	$/. ***	a-num/	a-num/ null	a-num/ null	>=4
local symbol	number **	$					>=2

where: a-num is an alphanumeric character.

E.3.3.1 Symbol Examples

Non-Global Symbols

```
AlB

ZXCJl

INSRT
```

Global Address Symbols

```
$JIM

.VECTR

$SEC
```

Global Absolute Offset Symbols

```
A$JIM

A$XT

A.ENT
```

---

* Symbols that are branch targets are also called labels, but we will always use the term "symbol".

** Number is in the range 0<number<65535.

*** The use of $ or . for global names is reserved for DEC-supplied software.

Global Bit Pattern Symbols

```
A1$20

B3.6

JI.M
```

Local Symbols

```
37$

271$

6$
```

E.3.3.2 **Local Symbols** - Target symbols for branches that exist solely for positional reference will use local symbols of the form

```
<num>$:
```

Local symbols are formatted such that the numbers proceed sequentially down the page and from page to page.

E.3.3.3 **Global Symbols** - Use of global symbols is restricted, within reason, to those cases where reference to the code occurs external to the code.

A program never contains a .GLOBL statement without showing cause.

E.3.3.4 **Macro Names** - In a macro name the last two characters (last character possibly being null) have special significance; the next to last character is a $, the last character specifies the mode of the macro.

For example, in the three macro forms in-line, stack, and p-section, the in-line form has no suffix, the stack has an <S> suffix, and the p-section a <C>. Thus the Queue I/O macro can be written as any of

```
QIO$

QIO$S

QIO$C
```

depending on the form required. These are not reserved letters.

E.3.3.5 **General Symbols** - Make frequently used bit patterns such as carriage return (CR) and line feed (LF) conventional symbols as they are needed.

E.4 PROGRAM MODULES

There are no limits on program size. However, since the virtual memory capacity of a computer is finite keep programs as compact as possible by:

1. creating them for a single function
2. writing them in accordance with the memory allocation guidelines in Appendix F.

Code areas are different than data areas. Code is read-only but data can be read-only or read-write; read-only data should be segregated from read-write data. Both areas, code and data, should have explanatory comments.

E.4.1 The Module Preface

Put each program module in a separate file. For easy reference the file name should be similar to the name of the module. The file type is of the form 'NNN' where 'NNN' is the edit or the version number (see Section E-8). The availability of File Control Services and File Control Primitives will greatly simplify version number maintenance.

E.4.2 The Module

Below is a list of the information included in each module, it is formatted as follows:

1. The first eight items appear on the same page and do not have explicit headings. Item 3 may be omitted if the blank p-section is being used.
2. Headings start at the left margin*; descriptive text is indented 1 tab position.
3. Items 7-14 have headings which start at the left margin, preceded and followed by lines containing only a leading <;>. Items which do not apply may be omitted.
4. A .TITLE statement that specifies the name of the module. If a module contains more than one routine, subtitles may be used.
5. An .IDENT statement specifying the version number (see Section E-8).
6. A .PSECT statement that defines the program section in which the module resides.

---

* The left margin consists a semicolon, a space and then the heading, therefore, the text of the heading begins in column 3.

7. A copyright statement, and the disclaimer.

   COPYRIGHT (C) 1979 BY
   DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASS.

   THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED AND COPIED ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE AND WITH THE INCLUSION OF THE ABOVE COPYRIGHT NOTICE. THIS SOFTWARE OR ANY OTHER COPIES THEREOF MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY OTHER PERSON. NO TITLE TO AND OWNERSHIP OF THE SOFTWARE IS HEREBY TRANSFERRED.

   THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT NOTICE AND SHOULD NOT BE CONSTRUED AS A COMMITMENT BY DIGITAL EQUIPMENT CORPORATION.

   DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR RELIABLILITY OF ITS SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DIGITAL.

8. The version number of the file (see Section E-8).

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

10. The name of each modifying author and the date of modification. These names and dates appear one per line and in chronological order.

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

NOTE

Items 3 to 11 should appear on the same page.

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

13. All local macro definitions, preferably in alphabetical order by name.

14. All local data. The data should indicate

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

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

16. A list of the inputs expected by the module. This includes the calling sequence if non-standard, condition code settings, and global data settings.

17. A list of the outputs produced as a result of entering this module. These include delivered results and condition code settings but not side effects. These outputs are visible to the caller.

18. A list of all effects produced as a result of entering this module. Effects include side effects, alterations in the state of the system not explicitly expected in the calling sequence and effects not visible to the caller.

19. The module code.

E.4.3 Module Example

FILE-EXAMPL.S01

```
        .TITLE   EXAMPLE
        .IDENT   /01/
        .PSECT   KERNEL

;
; COPYRIGHT (C) 1979 BY
; DIGITAL EQUIPMENT COPORATION, MAYNARD, MASS.
;
; THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED AND COPIED
; ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE AND WITH THE
; INCLUSION OF THE ABOVE COPYRIGHT NOTICE.  THIS SOFTWARE, OR ANY OTHER
; COPIES THEREOF MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY
; OTHER PERSON. NO TITLE TO AND OWNERSHIP OF THE SOFTWARE IS HEREBY
; TRANSFERRED.
;
; THE  INFORMATION IN THIS DOCUMENT IS SUBJECT TO CHANGE WITHOUT NOTICE
; AND SHOULD NOT BE CONSTRUED AS A COMMITMENT BY DIGITAL EQUIPMENT
; CORPORATION.
;
; DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR RELIABILITY OF ITS
; SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DIGITAL.
;
; VERSION 01
;
; JOE PASCUSNIK 1-JAN-72
;
; MODIFIED BY:
;
;       RICHARD DOE 21-JAN-73
;
;       SPENCER THOMAS 12-JUN-78
;
;    Brief statement of the module's function
;
; EQUATED SYMBOLS
;

        List equated symbols

;
; LOCAL MACROS
;
```

```
        Local Macros

;
; LOCAL DATA
;

        Local data

;+
; Module function-details
;
; INPUTS:
;
;       Description of inputs
;
; OUTPUTS:
;
;       Description of outputs
;
; EFFECTS:
;
;       Description of effects
;-

        Begin Module Code
```

E.4.4 Modularity

No other characteristic has more impact on the ultimate engineering success of a system than does modularity. Adherence to a set of call and return conventions helps achieve this modularity.

E.4.4.1 Calling Conventions (Inter-Module/Intra-Module)

Transfer of Control

Macros exist for call and return. The actual transfer is via a JSR PC instruction. For register save routines, a JSR Rn,SAVE is permitted.

The CALL macro is:

```
CALL    subr-name
```

The RETURN macro is:

```
RETURN
```

Register Conventions

On entry, a subroutine minimally saves all registers it intends to alter except result registers. On exit it restores these registers. (The preservation of the register state 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, use R0, R1 and R2 rather than R0, R2, R5. Saving and restoring occurs in one place.

E.4.4.2 **Exiting** - All subroutine exits occur through a single RETURN macro.

E.4.4.3 **Success/Failure Indication** - The C bit is used to return the success/failure indicator, where success equals 0, and failure equals 1. The argument registers can be used to return values or additional success/failure data.

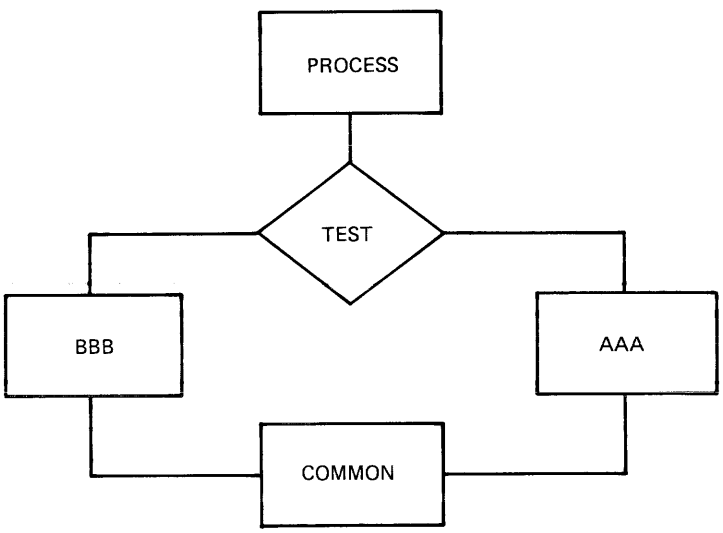
E.4.4.4 **Module Checking Routines** - Modules are responsible for verifying the validity of arguments passed to them. The design of a module's calling sequence should aim at minimizing the validity checks by minimizing invalid combinations. Programmers may add test code to perform additional checks during checkout. All code should aim at discovering an error as close (in terms of instruction executions) to its occurrence as possible.

E.5 CODE FORMAT

E.5.1 Program Flow

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

For example:



appears on the listing as:

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

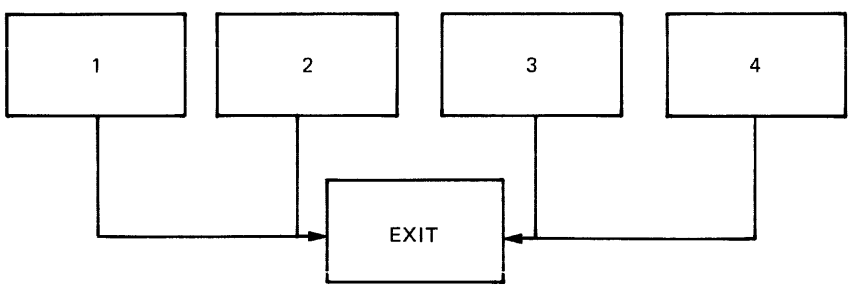
```
BBB:    ....    ....
        ....    ....
        ....    ....
CMN:    ....    ....
        ....    ....
        ....    ....
```

rather than:

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

E.5.2 Common Exits

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



appears on the listing as:

```
PR1:    ....    ....
        ....    ....
        ....    ....
        BR      EXIT

PR2:    ....    ....
        ....    ....
        ....    ....
        BR      EXIT
```

```
PR3:    ....    ....
        ....    ....
        ....    ....
        BR      EXIT

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

EXIT:
```

and not as:

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

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

PR2:    ....    ....
        ....    ....
        ....    ....
        BR      EXIT

PR3:    ....    ....
        ....    ....
        ....    ....
        BR      EXIT

PR4:    ....    ....
        ....    ....
        ....    ....
        BR      EXIT
```

E.5.3 Code with Interrupts Inhibited

Code that is executed with interrupts inhibited, is flagged by a three semicolon (;;;) comment delimiter. For example:

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

        BIS     #PR7,PS         ;;; INHIBIT INTERRUPTS
        BIT     #PR7,+2(SP)     ;;; C
        BEQ     10$             ;;;  O
        RTT                     ;;;   M
                                ;;;    M
10$:    ....    ....            ;;;     E
        ....    ....            ;;;      N
        ....    ....            ;;;       T
        ....    ....            ;;;        S
```

E.5.4 Code in System State

RSX-11M Executive subroutines and other privileged code that is executed in system state is flagged by a two semicolon (;;) comment delimiter. For example:

```
                                ; SWITCH TO SYSTEM STATE, ...
                                ;
                                ; AND EXIT.

        CALL    $SWSTK,EXIT     ; INHIBIT CONTEXT SWITCHING
                                ;; RETURN IN SYSTEM STATE
        ...                     ;;
        ...                     ;;
        ...


        RETURN                  ;; GO BACK TO USER STATE (EXIT)
EXIT:   ...                     ; USER STATE CODE
```

E.6 INSTRUCTION USAGE

E.6.1 Forbidden Instructions

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 problem, 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 (a move trace is unheard of)? As a more general issue, other operations such as ADD and SUB from PC should be discouraged.

3. The seemingly "neat" use of all single word instructions where one double-word instruction could be used and would execute faster and would not consume additional memory. 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.

4. Self-relative address arithmetic (.+n) is absolutely forbidden in branch instructions; its use in other contexts must be avoided if at all possible and practical.

E.6.2 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 (for example, BGT) when comparing two memory addresses. This works 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 results from re-linking the program at different addresses and/or changing the size of the program.

E.7 PROGRAM SOURCE FILES

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

The accumulation of corrections proceeds until a logical breaking point has occurred (a milestone or significant implementation point has been reached). At this time all accumulated corrections are applied to the previous base level to create a new base level and correction files are started for the new base level.

E.8 PDP-11 VERSION NUMBER STANDARD

The PDP-11 Version Number Standard applies to all modules, parameter files, complete programs, and libraries which are 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.

The version number is limited in that only six characters of identification are used. Future implementations of the Macro Assembler, linker, and librarian should provide for at least nine characters, and possibly twelve. It is expected that this standard will be improved as the need arises.

Version Identifier Format:

<version> <edit> <patch>

where: <version> consists of two decimal digits which represent the release number of a program. The version number starts at 00 and is incremented to reflect the number of major changes in the program.

<edit> consists of two decimal digits which represent the number of alterations made to the source program. The edit number begins at 01 (is null if there are no edits) and is incremented with each alteration.

<patch> is a letter between B and Z which represents the number of alterations made to the binary form of the program. The patch number begins at B (is null if there are no patches) and changes alphabetically with each patch.

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.8.1 Displaying the Version Identifier

The visible output of the version identifier should appear as:

Program
Name <key-letter> <version> - <edit> <patch>,

where the following Key Letters have been identified:

X in-house experimental version

Y field test, pre-release, or in-house release version

V released or frozen version

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

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

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

Examples:

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

E.8.2 Use of the Version Number in the Program

All sources must contain the version number in an .IDENT directive. In programs (or libraries) which consist of more than one module, each module must have a version number. 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 (for example, the FORTRAN Library) have an identification module in the first position. An identification module exists solely to provide identification. For example:

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

is an identification module.

APPENDIX F

ALLOCATING VIRTUAL MEMORY

This appendix is intended for the MACRO-11 user who wants to avoid the problem of thrashing, by optimizing the allocation of virtual memory. Users of smaller systems, particularly those with the 8K subset version of MACRO-11, should become thoroughly familiar with the conventions discussed herein. This appendix discusses the following topics:

1. General hints and space-saving guidelines
2. Macro definitions and expansions
3. Operational techniques.

The user is assumed to have pursued a policy of modular programming, as advised in Appendix E. Modular programming results in bodies of code that are small, distinct and highly functional. Using such code, which presents many advantages, one can usually avoid the problem of insufficient dynamic memory during assembly.

F.1 GENERAL HINTS AND SPACE-SAVING GUIDELINES

Work-file memory is shared by a number of MACRO-11's tables, each of which is allocated space on demand (64K words of dynamically pageable storage are available to the assembler). The tables and their corresponding entry sizes are as follows:

1. User-defined symbols - four words.*
2. Local symbols - three words.*
3. Program sections - six words.
4. Macro names - four words.*
5. Macro text - nine words.
6. Source files - six words.

In addition, several scratch pad tables are used during the assembly process, as follows:

1. Expression analysis - five words.
2. Object code generation - five words.
3. Macro argument processing - three words.
4. .MCALL argument processing - five words.

---

* Five words on RSX-11M.

The above information can serve as a guide for estimating dynamic storage requirements and for determining ways to reduce such requirements.

For example, the use of local symbols whenever possible is highly encouraged, since their internal representation requires 25 percent less dynamic storage than that required for regular user-defined symbols. The usage of local symbols can often be maximized by extending the scope of local symbol blocks through the .ENABL LSB/.DSABL LSB MACRO-11 directives (see Sections 3.5 and 6.2).

Since MACRO-11 does not support a purge function, once a symbol is defined, it permanently occupies its dynamic memory allocation. Numerous instances occur during conditional assemblies and repeat loops when a temporarily assigned symbol is used as a count or offset indicator. If possible, the symbols so used should be re-used.

In keeping with the same principle, special treatment should be given to the definition of commonly used symbols. Instead of simply appending a prefix file which defines all possibly used symbols for each assembly, users are encouraged to group symbols into logical classes. Each class can then become a shortened prefix file or a macro in a library (see Section F.2 below). In either case, selective definition of symbolic assignments is achieved, resulting in fewer defined (but unreferenced) symbols.

An example of this idea is seen in the definition of IAS/RSX-11M standard symbols. The system macro library, for example, supplies several macros used to define distinct classes of symbols. These groupings and associated macro names are as follows:

DRERR$ - Directive return status codes

FILIO$ - File-related I/O function codes

IOERR$ - I/O return status codes

SPCIO$ - Special I/O function codes

F.2 MACRO DEFINITIONS AND EXPANSIONS

Dynamic storage is used most heavily for the storage of macro text. Upon macro definition or the issuance of an .MCALL directive, the entire macro body is stored, including all comments appearing in the macro definition. For this reason, comments should not be included as part of the macro text. A librarian function switch (/SZ) are available to compress macro source text by removing all trailing blanks and tabs, blank lines, and comments. The system macro library (RSXMAC.SML) has already been compressed. User-supplied macro libraries (.MLB) and macro definition prefix files should also be compressed. For additional information regarding these two utility tasks, consult the applicable RSX-11M or RSX-11M-PLUS Utilities Manual (see Section 0.3 in the Preface).

It often seems practical to include a file of commonly used macro definitions in each assembly. This practice, however, may produce the undesirable allocation of valuable dynamic storage for unnecessary macros. This waste of memory can be avoided by making the file of macro definitions a user-supplied macro library file (see Table 8-1). This means that the names of desired macros must be listed as arguments in the .MCALL directive (see Section 7.8).

Certain types of macros can be redefined to null after they have been invoked. This practice not only frees storage space, it also eliminates the overhead and the dynamic memory wasted by calling a useless macro. The practice of redefining macros to null applies mainly to those that leave define symbolic assignments, as shown in the example below. The redefinition process may be accomplished as follows:

```
        .MACRO  DEFIN
SYM1 = VAL1                     ;DEFINE SYMBOLIC ASSIGNMENTS.
SYM2 = VAL2
          .
          .
          .
OFF1 = SYMBOL                   ;DEFINE SYMBOLIC OFFSETS.
OFF2 = OFF1+SIZ1
OFF3 = OFF2+SIZ2
          .
          .
          .
OFFN = OFFM+SIZM
          .
          .
          .
        .MACRO  DEFIN           ;MACRO NULL REDEFINITION.
        .ENDM

        .ENDM   DEFIN
```

Macros exhibiting this redefinition property should be defined (or read via the .MCALL directive) and invoked before all other macro definition and/or .MCALL processing, a practice that ensures more efficient use of dynamic memory.

The following RSX-11M system macros have the automatic null redefinition property after once being invoked:

BDOFF$ - File Control Services (FCS) buffer descriptor offsets

CSI$ - Command String Interpreter codes and offsets

DRERR$ - Directive return status codes

FCSBT$ - FCS bit value codes

FDOFF$ - FCS file descriptor block offsets

FILIO$ - File-related I/O function codes

FSROF$ - FCS file storage region (FSR) offsets

GCMLD$ - Get Command Line codes and offsets

IOERR$ - I/O return status codes

NBOFF$ - FCS filename block offsets

SPCIO$ - Special I/O function codes

F.3 OPERATIONAL TECHNIQUES

When, despite adhering to the guidelines discussed above, performance still falls below expectations, several additional measures may be taken to increase dynamic memory.

The first measure involves shifting the burden of symbol definition from MACRO-11 to the linker or task builder. In most cases, the definition of system I/O and File Control Services (FCS) symbols (and user-defined symbols of the same nature) is not necessary during the assembly process, since such symbols are defaulted to global references (Appendix D.1, category 4 of error code A). The linker or task builder attempts to resolve all global references from user-specified default libraries and/or the system object library (SYSLIB). Furthermore, by applying the selective search option for object modules consisting only of global symbol definitions, the actual additional burden to the linker is minimal.

The second way is to produce only one output file (either object or listing), as opposed to two. The additional memory required to support the second output file are allocated from available dynamic memory at the start of each assembly.

APPENDIX G

WRITING POSITION INDEPENDENT CODE

G.1 INTRODUCTION TO POSITION INDEPENDENT CODE

The output of a MACRO-11 assembly is a relocatable object module. The Task Builder or Linker binds one or more modules together to create an executable task image. Once created, if the program is to run it must be loaded at the virtual address specified at link time. This is because the Task Builder or Linker has to modify some instructions to reflect the memory locations in which the program is to run. Such a body of code is considered position-dependent (dependent on the virtual addresses to which it is bound).

All PDP-11 processors offer addressing modes that make it possible to write code that does not depend on the virtual addresses to which it is bound. Such code is termed position-independent and to run can be loaded at any virtual address. Position-independent code can improve system efficiency, both in use of virtual address space and in conservation of physical memory.

In multiprogramming systems like IAS, RSX-11M and RSX-11M-PLUS, it is important that many tasks be able to share a single physical copy of common code, for example a library routine. To make the optimum use of a task's virtual address space, shared code should be position-independent. Position-dependent code can also be shared, but it must appear in the same virtual locations in every task using it. This restricts the placement of such code by the Task Builder or Linker and can result in the loss of virtual addressing space.

The construction of position-independent code is closely linked to the proper usage of PDP-11 addressing modes. The remainder of this Appendix assumes you are familiar with the addressing modes described in Chapter 5.

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

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

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

The relative addressing modes are position-independent when a relocatable address is referenced from a relocatable instruction. These modes are as follows:

```
 A        relative mode
@A        relative deferred mode
```

Relative modes are not position-independent when an absolute address (that is a non-relocatable address) is referenced from a relocatable instruction. In this case, absolute addressing (@#A) may be used to make the reference position-independent.

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

```
 X(R)      index mode
@X(R)      index deferred mode
```

If the base, X, is an absolute value (for example, a control block offset), the reference is position-independent. For example:

```
        MOV     2(SP),R0        ;POSITION-INDEPENDENT
N=4
        MOV     N(SP),R0        ;POSITION-INDEPENDENT
```

If, however, X is a relocatable address, the reference is position-dependent. For example:

```
        CLR     ADDR(R1)        ;POSITION-DEPENDENT
```

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

```
#N        immediate mode
```

When an absolute expression defines the value of N, the code is position-independent. When a relocatable expression defines N, the code is position-dependent. That is, immediate mode references are position-independent only when N is an absolute value.

Absolute mode addressing is position-independent only in those cases where an absolute virtual location is being referenced. Absolute mode addressing references are formatted as follows:

```
@#A       absolute mode
```

An example of a position-independent absolute reference is a reference to the directive status word ($DSW) from a relocatable instruction. For example:

```
        MOV     @#$DSW,R0       ;RETRIEVE DIRECTIVE STATUS
```

G.2 EXAMPLES

The RSX-11M library routine, PWRUP, is a FORTRAN callable subroutine that establishes or removes a user power failure Asynchronous System Trap (AST) entry point address. Imbedded within the routine is the AST entry point that saves all registers, effects a call to the user-specified entry point, restores all registers on return, and executes an AST exit directive. The following examples are excerpts from this routine. The first example, Figure G-1 has been modified to illustrate position-dependent references. The second example, Figure G-2, is the position-independent version.

```
;+
; Position dependent code example
;-

PWRUP:: CLR     -(SP)           ;Assume success

; Perform further initialization...

        MOV     $OTSV,R4        ;Point R4 at object time system save area
                                ; the above reference to $OTSV is position-
                                ; dependent
        MOV     (SP)+,R2        ;Retrieve AST entry point address
        BNE     10$             ;Branch if one was specified
        CLR     -(SP)           ;If none, specify no power fail routine
        BR      20$             ;Bypass AST setup
10$:    MOV     R2,F.PF(R4)     ;Set the AST entry point
        MOV     #BA,-(SP)       ;Push our AST service address
                                ; the above reference to BA is position-
                                ; dependent
20$:

; Continue processing...

;+
; AST service routine
;-

BA:     MOV     R0,-(SP)        ;Preserve R0

; Rest of routine follows...
```

Figure G-1 Example of Position-Dependent Code

```
;+
; Position independent code example
;-

PWRUP:: CLR     -(SP)           ;Assume success

; Perform necessary initialization...

        MOV     @#$OTSV,R4      ;Point R4 at object time system save area
                                ; the above reference to $OTSV is position-
                                ; independent
        MOV     (SP)+,R2        ;Retrieve AST entry point address
        BNE     10$             ;Branch if one was specified
        CLR     -(SP)           ;If none, specify no power fail routine
        BR      20$             ;Bypass AST setup
10$:    MOV     R2,F.PF(R4)     ;Set the AST entry point
        MOV     PC,-(SP)        ;Push our PC to relocate our AST service addr
        ADD     #BA-.,(SP)      ;Relocate our AST service address now
                                ; the above reference to BA is position-
                                ; independent; this costs one word to relocate
20$:

; Continue processing...

;+
; AST service routine
;-

BA:     MOV     R0,-(SP)        ;Preserve R0

; Rest of routine follows...
```

Figure G-2 Example of Position-Independent Code

The position-dependent version of the subroutine contains a relative reference to an absolute symbol ($OTSV) and a literal reference to a relocatable symbol (BA). Both references are bound by the Task Builder to fixed memory locations. Therefore, the routine will not execute properly as part of a resident library if its location in virtual memory is not the same as the location specified at link time.

In the position-independent version, the reference to $OTSV has been changed to an absolute reference. In addition, the necessary code has been added to compute the virtual location of BA based upon the value of the program counter. In this case, the value is obtained by adding the value of the program counter to the fixed displacement between the current location and the specified symbol. Thus, execution of the modified routine is not affected by its location in the image's virtual address space.

The MACRO-11 Assembler provides a way of checking whether the code is position-independent. In an assembly listing, MACRO-11 inserts a ' character following the contents of any word which requires the Task Builder or Linker to perform a relocation operation and, therefore, may not be position independent code. The cases which cause an apostrophe to be inserted in the assembly listing are as follows:

1. Absolute mode references when the reference is relocatable. References are not flagged when they are absolute. For example:

```
MOV     @#ADDR,R1       ;PIC ONLY IF ADDR IS ABSOLUTE.
```

2. Index and index deferred mode references when the offset is relocatable. For example:

```
MOV     ADDR(R1),R5     ;NON-PIC IF ADDR IS RELOCATABLE.
MOV     @ADDR(R1),R5    ;NON-PIC IF ADDR IS RELOCATABLE.
```

3. Relative and relative deferred mode references when the address specified is relocatable with respect to another program section. For example:

```
MOV     ADDR1,R1        ;NON-PIC WHEN ADDR1 IS ABSOLUTE
MOV     @ ADDR1,R1
```

4. Immediate mode references to relocatable addresses.

```
MOV     #ADDR,R1        ;NON-PIC WHEN ADDR IS RELOCATABLE.
```

In one case, MACRO-11 does not flag a potential position-dependent reference. This occurs where a relative reference is made to an absolute virtual location from a relocatable instruction (see the MOV $OTSV,R4 instruction in Figure G-1).

References requiring more than simple relocation at link time are indicated in the assembly listing. Simple global references are flagged with the letter G. Statements which contain multiple global references or require complex relocation, are flagged with the letter C (see Section 3.9 and Chapter 4). It is difficult to positively state whether or not a C-flagged statement is position-independent. However, in general, position dependence can be decided by applying the guidelines discussed earlier in this Appendix to the resulting address value produced at link time.

APPENDIX H

SAMPLE ASSEMBLY AND CROSS REFERENCE LISTING

```
R50UNP  MACRO Y04.00  29-AUG-79 16:06:39
TABLE OF CONTENTS

    2-   1      RAD50 unpack routine
```



```
 1                          .TITLE  R50UNP
 2                          .IDENT  /02/
 3
 4                ;
 5                ;                          COPYRIGHT (c) 1979 BY
 6                ;                DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASS.
 7                ;
 8                ; THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED AND  COPIED
 9                ; ONLY  IN  ACCORDANCE  WITH  THE  TERMS  OF   SUCH  LICENSE AND WITH THE
10                ; INCLUSION OF THE ABOVE COPYRIGHT NOTICE.  THIS SOFTWARE OR   ANY  OTHER
11                ; COPIES  THEREOF MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY
12                ; OTHER PERSON.  NO TITLE TO AND OWNERSHIP OF  THE  SOFTWARE  IS  HEREBY
13                ; TRANSFERRED.
14                ;
15                ; THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE  WITHOUT  NOTICE
16                ; AND  SHOULD  NOT  BE  CONSTRUED  AS  A COMMITMENT BY DIGITAL EQUIPMENT
17                ; CORPORATION.
18                ;
19                ; DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR  RELIABILITY  OF  ITS
20                ; SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DIGITAL.
21                ;
22                ; UPDATE HISTORY:
23                ;
24                ;       D.N. CUTLER      10-FEB-73
25                ;
```

```
R50UNP  MACRO Y04.00  29-AUG-79 16:06:39 PAGE 2
RAD50 UNPACK ROUTINE

                        .SBTTL  RAD50 unpack routine

                ;+
                ; R50UNP
                ; Unpack a 6 char RAD50 symbol to ASCII
                ;
                ; Enter with R2 -> Output ASCII string
                ; SYMBOL, SYMBOL+2 = RAD50 symbol to unpack
                ;
                ; Return with R2 -> Past output string
                ; R0, R1, R3 Destroyed
                ;-

000000                  .PSECT  PUREI,I

000000  010446  R50UNP::MOV     R4,-(SP)        ;Save R4
000002  012704          MOV     #$SYMBOL,R4     ;Point at RAD50 symbol buffer
        000000G
000006  012401  1$:     MOV     (R4)+,R1        ;Get next RAD50 word
000010  012703          MOV     #50*50,R3       ;Set divisor for high character
        003100
000014  004767          CALL    10$             ;Unpack and store the character
        000030
000020  012703          MOV     #50,R3          ;Now set divisor for middle character
        000050
000024  004767          CALL    10$             ;Unpack and store the character
        000020
000030  010100          MOV     R1,R0           ;Copy remaining character
000032  004767          CALL    11$             ;Translate and store it
        000016
000036  020427          CMP     R4,#SYMBOL+4    ;Test if last word done
        000046G
000042  001361          BNE     1$              ;Branch if no
000044  012604          MOV     (SP)+,R4        ;Restore R4
000046  000207          RETURN                  ;Return to caller

                ; Divide RAD50 word and convert char to ASCII

000050  005000  10$:    CLR     R0
000052  071003          DIV     R3,R0

                ; Translate RAD50 character code to ASCII
                ; 0 =    space
                ; 1-32 = A-Z
                ; 33 =   $
                ; 34 =   .
                ; 35 =   unused code
                ; 36-47 = 0-9

000054  005700  11$:    TST     R0              ;Test if space
000056  001412          BEQ     23$             ;Branch if so
000060  020027          CMP     R0,#33          ;Test if middle
        000033
000064  002405          BLT     22$             ;Branch if alphabetic
000066  001402          BEQ     21$             ;Branch if dollar sign
000070  062700          ADD     #22-11,R0       ;Dot or digits 0-9
        000011

R50UNP  MACRO Y04.00  29-AUG-79 16:06:39 PAGE 2-1
RAD50 UNPACK ROUTINE

000074  062700  21$:    ADD     #11-100,R0      ;Dollar
        177711
000100  062700  22$:    ADD     #100-40,R0      ;Alphabetic
        000040
000104  062700  23$:    ADD     #40,R0          ;Space
        000040
000110  110022          MOVB    R0,(R2)+        ;Store ASCII char in buffer
000112  000207          RETURN
                .END
000001
```

SAMPLE ASSEMBLY AND CROSS REFERENCE LISTING

```
R50UNP  MACRO Y04.00  29-AUG-79 16:06:39 PAGE 2-2
SYMBOL TABLE

R50UNP  000000RG    002 SYMBOL= ****** GX

. ABS.  000000      000
        000000      001
PUREI   000114      002
ERRORS DETECTED:  0

VIRTUAL MEMORY USED: 7936 WORDS ( 1 PAGES)
DYNAMIC MEMORY AVAILABLE FOR 72 PAGES
,R50UNP=R50UNP/C:C:S:M:E:R:P/E:GBL:LC/L:TTM
```

```
R50UNP  MACRO Y04.00  29-AUG-79 16:06:39 PAGE S-1
CROSS REFERENCE TABLE (CREF V01-08 )

R50UNP   2-16#
SYMBOL   2-17     2-25
```

```
R50UNP  MACRO Y04.00  29-AUG-79 16:06:39 PAGE R-1
CROSS REFERENCE TABLE (CREF V01-08 )

R0       2-23*    2-32*    2-33*    2-43     2-45     2-48*    2-49*
         2-50*    2-51*    2-52
R1       2-18*    2-23
R2       2-52*
R3       2-19*    2-21*    2-33
R4       2-16     2-17*    2-18     2-25     2-27*
SP       2-16*    2-27
```

```
R50UNP  MACRO Y04.00  29-AUG-79 16:06:39 PAGE C-1
CROSS REFERENCE TABLE (CREF V01-08 )

         0-0
. ABS.   0-0
PUREI    2-14
```

INDEX

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

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

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

Did you find errors in this manual? If so, specify the error and the page number.

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

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

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify)_____

Name_____ Date_____

Organization_____

Street_____

City_____ State_____ Zip Code_____
or

Please cut along this line.

— — Do Not Tear - Fold Here and Tape —

digital

No Postage Necessary if Mailed in the United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

RT/C SOFTWARE PUBLICATIONS ML 5-5/E45
DIGITAL EQUIPMENT CORPORATION
146 MAIN STREET
MAYNARD, MASSACHUSETTS 01754

— — Do Not Tear - Fold Here —

digital equipment corporation

Printed in U.S.A.