RDOS/DOS

Macroassembler

User's Manual

093-000081-05

For the latest enhancements, cautions, documentation changes, and other information on this product, please see the Release Notice (085-series) supplied with the software.

Ordering No. 093-000081
© Data General Corporation, 1972, 1973, 1974, 1975, 1978
All Rights Reserved
Printed in the United States of America
Revision 05, August 1978
Licensed Material - Property of Data General Corporation

Licensed Material - Property of Data General Corporation

DataGeneral
SOFTWARE DOCUMENTATION

NOTICE

Data General Corporation (DGC) has prepared this manual for use by DGC personnel, licensees, and customers. The information contained herein is the property of DGC and shall not be reproduced in whole or in part without DGC prior written approval.

DGC reserves the right to make changes without notice in the specifications and materials contained herein and shall not be responsible for any damages (including consequential) caused by reliance on the materials presented, including but not limited to typographical, arithmetic, or listing errors.

RDOS/DOS
Macroassembler
User's Manual
093-000081

Revision History:

Original Release - July 1972
First Revision - November 1972
Second Revision - June 1973
Third Revision - March 1974
Fourth Revision - January 1975
Fifth Revision - August 1978

This manual replaces manual no. 093-000131.

This document has been extensively revised from revision 04; therefore, change indicators have not been used.

The following are trademarks of Data General Corporation, Westboro, Massachusetts:

U.S. Registered Trademarks			Trademarks
CONTOUR I	INFOS	NOVALITE	DASHER
DATAPREP	NOVA	SUPERNOVA	microNOVA
ECLIPSE	NOVADISC		

Preface

This manual describes the Data General Macroassembler, as provided for use within our Real-Time Disk Operating system (RDOS) or Diskette Operating System (DOS). It assumes that you plan to program in assembly language, and that you have a background in this language.

The Macroassembler (which you invoke with the CLI command MAC), processes source files written in assembly language and produces relocatable binary files, called RBs by convention. After assembling a source file with MAC, you can make the RBs an executable save file with the RLDR utility, or generate an RB library with the LFE utility.

Within this manual:

Chapter 1 introduces the Macroassembler; it describes input, output, and relocatability.

Chapter 2 details the characters you input to the assembler, and its behavior thereafter; it covers character, number, and symbol formats.

Chapter 3, Syntax, explains expressions and introductions to the assembler, like LDA and DIA.

Chapter 4 lists and describes the pseudo-ops and value symbols which you can define in your source program. It gives these directives by category and in alphabetical order. The alphabetical section begins on a yellow page.

Chapter 5 explains advanced features of the assembler: macros and generated labels.

Chapter 6 describes operating the assembler from the console, and includes the required files, command switches, and symbol table files.

Among the appendixes, Appendix A explains the assembler error messages, B summarizes permanent symbols, and C contains an ASCII character subset.

Other manuals you will find helpful are:

Programmer's Reference Manual for NOVA Computers (ordering number 015-000023) or *Programmer's Reference Manual for ECLIPSE Computers* (015-000024). One of these details the instruction set for your machine.

RDOS/DOS Command Line Interpreter (CLI) User's Manual (093-000109). This explains how to operate any system utility from the console.

Text Editor User's Manual (093-000018) or *Supereditor User's Manual* (093-000111). Both editors allow you to write and modify source programs; the Supereditor is a more powerful, programmable version of the basic Text Editor.

Library File Editor User's Manual (093-000074). This utility helps you analyze and edit relocatable binary libraries (LBs) and RBs.

Extended Relocatable Loaders User's Manual (093-000080). This utility, RLDR, binds MAC's RB output into an executable program.

RDOS Reference Manual (093-000075) or *DOS (Diskette) Reference Manual* (093-000201). The appropriate book explains the features of your operating system, and all of its system and task calls.

Notation Conventions for This Manual

We use the following conventions for instruction and command formats in this manual. Note that these conventions are used to describe the Macroassembler language, but are not part of the language itself.

COMMAND required *[optional]* ...

Where	Means
COMMAND	You must enter the command (or its accepted abbreviation) as shown.
required	You must enter some argument (such as a filename). Sometimes, we use: { $required_1$ / $required_2$ } which means you must enter *one* of the arguments. Don't enter the braces; they only set off the choice.
[optional]	You have the option of entering some argument. Don't enter the brackets; they only set off what's optional.
...	You may repeat the preceding entry or entries. The explanation will tell you exactly what you may repeat.

Additionally, we use certain symbols in special ways:

Symbol	Means
)	Press the RETURN key on your terminal's keyboard.
□	Be sure to put a space here. (We use this only when we must; normally, you can see where to put spaces.)

All numbers are decimal unless we indicate otherwise; e.g., 35_8.

Finally, we usually show all examples of entries and system responses in THIS TYPEFACE. But, where we *must* clearly differentiate your entries from system responses in a dialog, we will use

THIS TYPEFACE TO SHOW YOUR ENTRY)
THIS TYPEFACE FOR THE SYSTEM RESPONSE

We welcome your comments and suggestions for this and other Data General publications. To communicate with us, either use the comments form provided at the back of this manual or write directly to:

Software Documentation
Data General Corporation
Westboro, MA 01581

End of Preface

Contents

Chapter 1 - Introduction to the Macroassembler

Assembly Language Processing 1-1
Macro Facility 1-1
Assembler Input and Output 1-1
Assembler Input 1-2
Types of Assembler Output 1-2
Binary File Output 1-2
Relocatable Binaries 1-2
Program Listing 1-3
Cross-Reference Listing 1-4
Error Listing 1-4
Relocatability 1-6

Chapter 2 - Fundamental Assembly Tools

Character Input 2-1
String Mode 2-1
Normal Mode 2-1
Atoms 2-1
Operators 2-2
Break Characters 2-2
Number Atoms 2-2
Number Representations 2-3
Single-Precision Integer Representation 2-4
Special Formats of Single-Precision Integers 2-4
Double-Precision Integer Representation 2-5
Single-Precision Floating-Point Constants 2-6
Examples of Numbers 2-7
Symbols 2-7
Special Characters 2-7
@ Commercial AT Sign 2-7
# Number Sign 2-7
** Asterisks 2-8

Chapter 3 - Syntax

Expressions . . . 3-1
Operators . . . 3-1
Bit Alignment Operator . . . 3-2
Examples of Expressions . . . 3-2
Relocation Properties of Expressions . . . 3-3
Symbols . . . 3-4
Permanent Symbols . . . 3-4
Semipermanent Symbols . . . 3-5
User Symbols . . . 3-5
Instructions . . . 3-5
Arithmetic and Logical (ALC) Instructions . . . 3-6
I/O Instructions without Accumulator . . . 3-7
I/O Instructions with Accumulator . . . 3-8
I/O Instructions without Device Code . . . 3-9
I/O Instructions without Argument Fields . . . 3-10
Memory Reference (MR) Instructions . . . 3-10
MR Instructions without Accumulator . . . 3-10
MR Instructions with Accumulator . . . 3-13
ECLIPSE Instructions . . . 3-14
Extended MR Instructions . . . 3-14
Commercial MR Instructions . . . 3-15
Floating-Point Instructions . . . 3-15

Chapter 4 - Pseudo-ops and Value Symbols

Symbol Table Pseudo-ops . . . 4-1
Symbol Table Pseudo-op Format . . . 4-2
Location Counter Pseudo-ops . . . 4-3
Intermodule Communication Pseudo-ops . . . 4-3
Repetition and Conditional Pseudo-ops . . . 4-4
Macro Definition Pseudo-op and Values . . . 4-4
Stack Pseudo-ops and Values . . . 4-4
Text String Pseudo-ops and Values . . . 4-4
Listing Pseudo-ops and Values . . . 4-5
Miscellaneous Pseudo-ops . . . 4-5
(.) Current Location Counter . . . 4-5
.ARGCT . . . 4-6
.BLK . . . 4-6
.COMM . . . 4-7
.CSIZ . . . 4-8
.DALC . . . 4-9
.DCMR . . . 4-10
.DEMR . . . 4-11
.DERA . . . 4-12
.DEUR . . . 4-13
.DFLM . . . 4-13
.DFLS . . . 4-14
.DIAC . . . 4-14
.DICD . . . 4-15
.DIMM . . . 4-15
.DIO . . . 4-16
.DIOA . . . 4-16
.DISD . . . 4-17
.DISS . . . 4-17

Chapter 4 (continued)

Miscellaneous Pseudo-ops (continued)
.DMR . . . 4-18
.DMRA . . . 4-18
.DO . . . 4-19
.DUSR . . . 4-19
.DXOP . . . 4-20
.EJEC . . . 4-20
.END . . . 4-21
.ENDC . . . 4-21
.ENT . . . 4-22
.ENTO . . . 4-22
.EOF, .EOT . . . 4-23
.EXTD . . . 4-23
.EXTN . . . 4-24
.EXTU . . . 4-24
.GADD . . . 4-25
.GLOC . . . 4-25
.GOTO . . . 4-26
.GREF . . . 4-26
.IFE, .IFG, .IFL, .IFN . . . 4-27
.LMIT . . . 4-28
.LOC . . . 4-29
.MACRO . . . 4-29
.MCALL . . . 4-30
.NOCON . . . 4-30
.NOLOC . . . 4-31
.NOMAC . . . 4-31
.NREL . . . 4-32
.PASS . . . 4-32
.POP . . . 4-33
.PUSH . . . 4-33
.RB . . . 4-34
.RDX . . . 4-34
.RDXO . . . 4-35
.REV . . . 4-35
.TITL . . . 4-36
.TOP . . . 4-36
.TXT, .TXTE, .TXTF, .TXTO . . . 4-37
.TXTM . . . 4-38
.TXTN . . . 4-38
.XPNG . . . 4-39
.ZREL . . . 4-39

Chapter 5 - Macros and Other Advanced Features

The Macro Facility . . . 5-1
Macro Definition . . . 5-1
Macro Calls . . . 5-3
Listing of Macro Expansions . . . 5-5
.DO Loops and Conditionals . . . 5-6
Macro Examples . . . 5-8
Generated Labels . . . 5-15
Literals . . . 5-16
Generated Numbers and Symbols . . . 5-17

Chapter 6 - How to Operate the Macroassembler

Assembler Files . . . 6-1
File LITMACS.SR . . . 6-1
Operating Procedures . . . 6-2
Global Switches . . . 6-2
Local Switches . . . 6-3
Macroassembler Symbol Table Files . . . 6-4

Appendix A - Error Codes

Assembly Errors . . . A-1
Addressing Error (A) . . . A-2
Bad Character (B) . . . A-2
Macro Error (C) . . . A-2
Radix Error (D) . . . A-2
Equivalence Error (E) . . . A-3
Format Error (F) . . . A-3
Global Symbol Error (G) . . . A-3
Input (Parity) Error (I) . . . A-3
Conditional Assembly Error (K) . . . A-3
Location Error (L) . . . A-3
Multiple Definition Error (M) . . . A-4
Number Error (N) . . . A-4
Field Overflow Error (O) . . . A-4
Phase Error (P) . . . A-4
Questionable Line (Q) . . . A-5
Relocation Error (R) . . . A-5
Undefined Symbol Error (U) . . . A-5
Variable Label Error (V) . . . A-5
Text Error (X) . . . A-5
Fatal Errors . . . A-6

Appendix B - Permanent Symbols

Appendix C - ASCII Character Subset

Illustrations

Figure Caption

1-1 Macroassembler Output 1-2
1-2 Program Listing 1-3
1-3 Cross-Reference Listing 1-5
1-4 How RLDR Operates on Binary Modules 1-6

3-1 Assembly of ALC Instruction 3-7
3-2 Assembly of an I/O Instruction without Accumulator 3-8
3-3 Assembly of I/O Instructions with Accumulator 3-9
3-4 Assembly of I/O Instructions without Device Code 3-10
3-5 Assembly I/O Instructions without Argument Fields 3-12
3-6 Assembly of MR Instructions without Accumulator 3-12
3-7 Formation of Effective Address for MR Instruction 3-13
3-8 Assembly of MR Instructions with Accumulator 3-13

5-1 Macro Calls and Expansions 5-4
5-2 Forms 2 and 3 Macro Calls 5-5
5-3 Logical OR Macro 5-8
5-4 Factorial Macro 5-9
5-5 Packet Decimal Macro 5-10
5-6 VFD, ERROR, and SPECL Macros 5-14
5-7 Generating Labels 5-15

Chapter 1
Introduction to the Macroassembler

A language is a set of common representations, conventions and rules that convey information in a well-defined way. Computer languages range from those which resemble and are tailored to computer hardware operations, to those which are more like human language. Machine language uses numeric codes that a computer can understand directly, while FORTRAN, BASIC, and other high-level languages are more akin to human expressions. The Macroassembler employs a language that is somewhere in between. It accepts symbols which are often only names for machine code, yet frees you from the need to be concerned with exact memory locations; it provides mathematical and logical operations for symbol manipulation; and it provides a macro facility that permits your own character sequences to be expanded into different forms by the assembler.

Assembly Language Processing

The assembler translates symbolic instruction codes (such as "LDA 0,2") and symbolic addresses (such as "TEMP") into numeric codes and numeric addresses. These addresses may be either absolute (i.e., "real") or relocatable; these terms are defined at greater length later in this chapter.

Symbolic language that you input to the assembler is called a *source file* or *source module*. The assembler's output is called a *relocatable binary file* or RB file or RB *module*. The computer cannot execute a source file (it is symbolic), nor can it execute a binary file directly. One or more binary files must be further processed to make them executable. This process, called loading, forms the binary file(s) into an executable *save file* which the computer can execute.

Macro Facility

Symbolic assembly language programming is simpler than machine language programming. Macro assembly can further simplify programming.

Quite often a program uses the same set of symbolic instructions many times. Macro assembly permits you to write a set of instructions only once, and substitute this set wherever you wish during assembly of a source file.

Fundamentally, a macro facility works as follows:

1. You write a set of symbolic instructions, called a *macro definition*, and give the macro definition a name.
2. Wherever you want that set of symbolic instructions in your source file, write a *macro call*. At minimum, the macro call contains the name of the macro definition.
3. The assembler contains a macro processor which substitutes the sequence of instructions (macro definition) for the macro call. This substitution is called *macro expansion*.

The macro facility also offers more sophisticated features. For example, the same set of instructions (differing in only accumulators and addresses) may be used many times within a program. If so, you can write formal (dummy) arguments for accumulators and addresses into the macro definition. The macro call in the program will contain the actual arguments. During macro expansion, the Macroassembler will substitute the actual arguments for the dummies. Thus, a macro definition is usually a skeleton of the actual instruction set that will result from macro expansion.

Assembler Input and Output

Figure 1-1 shows the input to and the possible outputs from the Macroassembler. Input consists of one or more source files written in an ASCII character subset. Output includes, at minimum, a list of any source program errors. Maximum assembler output includes a program listing (which includes any errors), a separate error listing, and a binary file. The program and error listings give you information; the binary file can be processed by RLDR to make it executable.

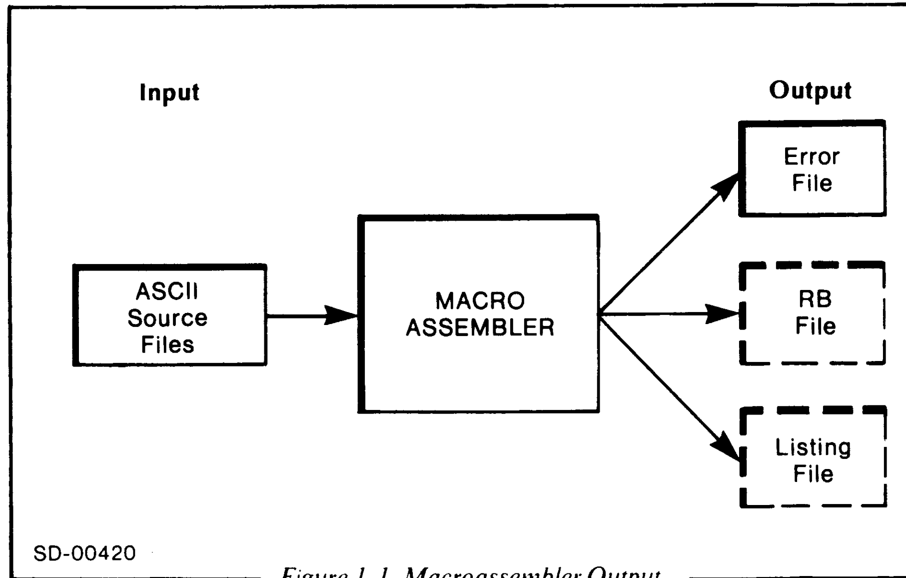


Figure 1-1. Macroassembler Output

Assembler Input

The source program input to the assembler consists of characters which are a subset of the ASCII character set. The assembler reads the source program twice; each read is called a pass. On each pass it performs the following elementary functions:

1. It reads a *line* of source consisting of a character string terminated by a carriage return ( ) -- ASCII 15) or a form feed (↓ -- ASCII 14) character.
2. It ignores three characters unconditionally. These are null (value 000), line feed (012) and rubout (value 177).
3. It replaces characters having incorrect parity with the ASCII character backslash "\". This character is then transparent to the assembler; for example, L\A is processed as LA.

Types of Assembler Output

There are three possible outputs from assembly:

1. A relocatable binary file.
2. A program listing.
3. An error listing.

Binary File Output

The assembler begins the translation into binary output by reading the source line. To translate the source line, the assembler must:

1. Build syntactically recognizable elements called atoms. Atoms are numbers, symbols, operators, break characters, or special characters.
2. Recognize and act upon each basic atom.

The binary output is a translation of source program lines into a special blocked binary code. Most lines of source input translate into single 16-bit (one-word) binary numbers for input to RLDR. The assembler gives each number an address. This address is not necessarily the final memory address for the number; it may be a relative address that RLDR will relocate. The assembler produces as part of the binary file the information which RLDR needs to map each address and its contents.

You may choose not to output a binary file.

Relocatable Binaries

An RB can have three different sections of code: absolute, ZREL (page zero relocatable) and NREL (normal relocatable). Within a source program, you specify absolute code with the .LOC pseudo-op, ZREL code with the .ZREL pseudo-op, and NREL code with the .NREL pseudo-op.

Program Listing

The program listing permits you to compare your input against the assembler output. A line of the program listing contains the following information:

Columns 1 - 3 — If the assembler finds no errors in the input, columns 1-3 contain a two-digit line number followed by a blank space. If there are any input errors, each error generates a single letter code. The first error generates a letter code in column 3, the next in column 2, and a third in column 1. Only three error codes can be listed per line. Lines which have errors receive no line number.

Columns 4 - 8 — Contain the location counter, if relevant. Otherwise, columns 4-8 are left blank.

Column 9 — Contains the relocation flag pertaining to the location counter.

Columns 10 - 15 — Contain the data field, if relevant. Otherwise, these contain the value, in the current radix, of an equivalence expression (such as A = 2*3) or of a pseudo-op argument (such as .RDX 16). In other cases, columns 10-15 are left blank.

Column 16 — Contains the relocation flag pertaining to the data field.

Column 17 - on — Contain the source line as written and as expanded by macro calls.

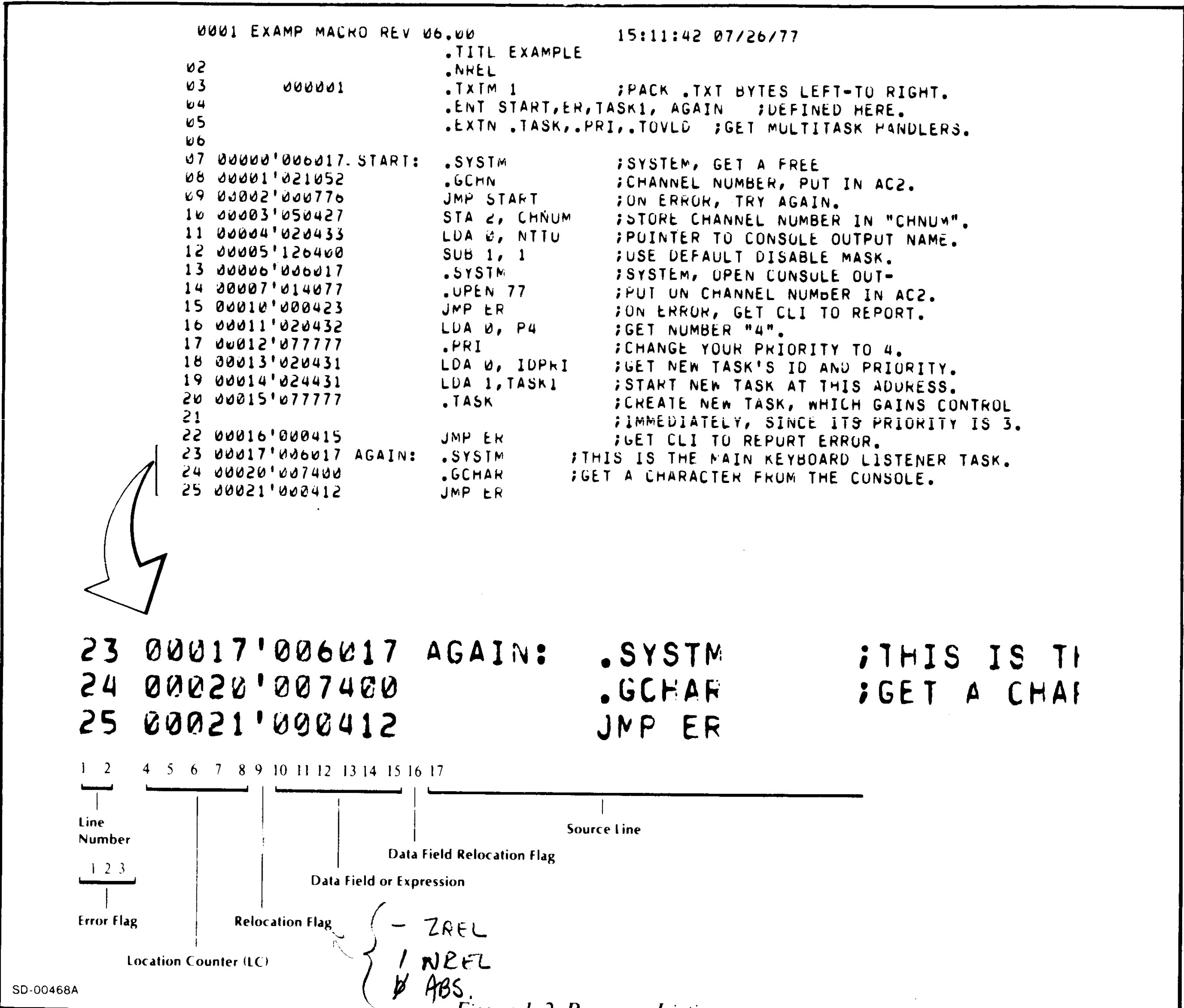


Figure 1-2. Program Listing

An error flag is a single letter indicating the type of error which appeared in the source line. A parity error on input, for example, produces the flag I in column 3 of the program listing line. Up to three error flags may appear on a given line.

The 5-digit location counter (LC) assigned by the assembler to an instruction or datum is displayed in columns 4-8. The LC is immediately followed by a single-character flag indicating the relocation mode of the address:

Flags	Meaning
space	absolute
-	page zero relocatable
'	normal relocatable

Following the LC flag is the 6-column data value field, immediately followed by a single-character flag indicating the relocation mode of the value.

Flags	Meaning
space	absolute
-	page zero relocatable
=	page zero, byte-relocatable
'	NREL code
"	NREL code, byte-relocatable
$	displacement field is externally defined

The last item on each program listing line is the ASCII source line. This line is given as input, except for expansion by macro calls.

You may choose to suppress certain lines of the listing (macro expansion, for example). You may also choose not to output a program listing.

Cross-Reference Listing

A program listing always includes a cross-reference listing of the symbol table, which includes user symbols alone or both user and semipermanent symbols. A sample cross-reference listing follows in Figure 1-3.

Here is an explanation of all cross-reference symbols:

□□	user-symbol
EN	entry (.ENT pseudo-op)
EO	overlay entry (.ENTO pseudo-op)
XD	external displacement (.EXTD pseudo-op)
XN	external normal (.EXTN pseudo-op)
MC	macro
NC	named common (.COMM pseudo-op)

Error Listing

The error listing contains the title of the source module and all source lines that have been flagged with an error code. The error listing is useful in programs with very long listings since it acts as an abstract; nonetheless, it contains no information which is not also present in the assembly listing.

```
        0006 MTACA

     C377    000075'          4/20   4/47   5/04
     C5      000023'          4/06   4/10
     CTCB    000001$    XD    2/13   2/31   4/11
     CTSUS   000077'          4/42   5/08
     CXMT    000076'          4/44   5/06
     KILL    000072'    XN    2/30   4/59
     MSW     000001'          2/01   2/04   2/37   2/40   2/51
                              3/08   4/02   4/05
     PCTMP   000002$    XD    2/14   2/31   2/43   2/57   3/11
                              4/26
     TACM1   000031'          4/17   4/25
     TACM2   000040'          4/24   4/35
     TACMN   000016'          2/46   2/60   4/01
     TAK1    000054'          4/29   4/39
     TAK2    000003'          2/45
     TAK3    000064'          4/47
     TAKIL   000000'    EN    2/17   2/29   2/36
     TAPEN   000005'    EN    2/18   2/29   2/50
     TAPR    000042'          4/23   4/26
     TAPRX   000052'          4/34   4/46   4/49
     TAUNP   000012'    EN    2/19   2/29   3/04
     TMAX2   000073'    XN    2/30   4/60
     TSAVE   000074'    XN    2/30   5/01
     .AKRT   000067'          4/39   4/51
     .TMN1   000073'          2/14   4/18   4/60
     .TSAV   000074'          2/14   2/44   2/58   3/12   5/01
```

Symbol — Symbol's Address — Relocation Flag — Type of Symbol — Page and line where referenced, for example 4/20 indicates page 4, line 20

SD-00469

Figure 1-3. Cross-Reference Listing

Relocatability

MAC is a relocatable assembler, which means that it assigns each storage word a *relative* location counter value. RLDR takes each *relative* value and gives it an *absolute* memory address.

MAC can produce output that will be placed by RLDR for execution in either the absolute, the ZREL, or the NREL sections of memory.

MAC assigns relative values via three counters which it maintains for each type of relocatability: one each for absolute, ZREL, and NREL code. The ZREL and NREL counters are initially zero; MAC increases them by one for every storage word it generates. When a program has been completely assembled, it has used *z* ZREL words and *n* NREL words. These words have been assigned *relative* addresses ZREL 0 to (z-1) and NREL 0 to (n-1).

RLDR's role is to take a number of assembled modules and form a nonoverlapping save file for execution. It does this by taking the assembler's *relative* addresses and making them into absolute addresses, via its own counters. Like MAC, RLDR maintains three counters (for absolute, ZREL, and NREL code). Using these counters, RLDR establishes an absolute address for each relative address of the modules it processes. It initializes the ZREL counter to 50_8 and initializes the NREL counter(s) according to the UST length, the number of tasks specified, and the size and number of overlay nodes in the program. RLDR assigns each symbol an absolute address by adding its *relative* address to the ZREL and NREL counter(s). After loading each module, RLDR updates its counters to include the number of ZREL and NREl words used by that module, thus setting up the starting addresses of absolute memory for the next module.

In this way, RLDR loads any number of separately assembled modules together, without storage conflict. This is the major advantage of relocatability. Figure 1-4 shows RLDR's action in a simple case, where only three modules, A, B, and C are loaded together to form a simple program. Note that the binding of real modules is more complex; for example, true programs do not normally begin at location 0. .ZREL code usually begins at location 50_8, and .NREL code begins after the last system-generated table. For more on tasks and the system tables see Chapter 5 of your operating system manual; for more on overlays, see Chapter 4 of the same book.

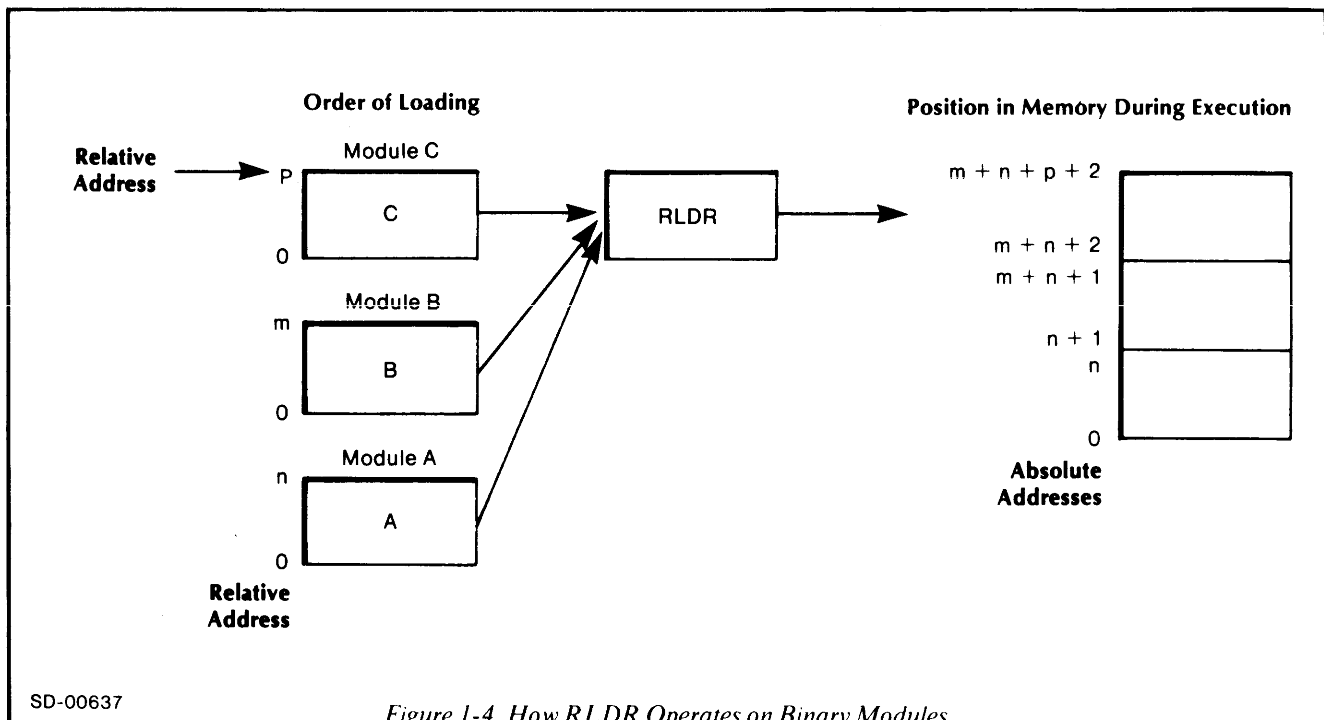


Figure 1-4. How RLDR Operates on Binary Modules.

End of Chapter

Chapter 2
Fundamental Assembly Tools

Character Input

You can input characters to the assembler in one of two modes: *normal* and *string*.

String Mode

In string mode, the assembler accepts any ASCII character and returns it unchanged. String mode input is not interpreted. You can set string mode in one of the three different forms:

1. Comments

   A comment begins with a semicolon and is terminated with a carriage return or form feed, e.g., ;SET MASK BITS.

2. Macro Definition Strings (not macro calls)

   A macro definition string begins with the pseudo-op *.MACRO,* is followed by one or more spaces, tabs, or commas, and is terminated with the character %. For example, the following three source lines define a macro:

```
.MACRO X)
LDA 0,2)
MOVZL 1,1)
%
```

3. Text Strings

   A text string begins with a text pseudo-op, followed by a standard delimiter, followed by a delimiter that is any character not used in the character string. The text string is terminated by the appearance of the same delimiting character that was used at the beginning. For example,

```
.TXT "EXPECTED VALUE=60% OF GROSS $."
```

Normal Mode

All other input is in normal mode. In normal mode, the input string consists of characters in a subset of the ASCII character set, divided into lines. Each line of characters is terminated by either a carriage return or a form feed. In normal mode, the assembler recognizes the following ASCII codes:

All alphabetics, numerals, relational operators, most conventional punctuation, and certain special characters. See the ASCII subset in Appendix C.

In normal mode, lowercase alphabetics are always translated to uppercase. During assembly, any character not within the subset in Appendix C is given a B (bad character) flag and is syntactically ignored.

Atoms

In normal mode, the assembler recognizes certain characters and certain groups of characters as different types of atoms. An *atom* is a syntactic element of assembly language recognized by its specific class. Atoms fall into five classes:

1. Operators
2. Break Characters
3. Numbers
4. Symbols
5. Special Characters

Operators

Operator characters are used with single-precision integers and symbols to form expressions. There are three classes of operators:

Class	Operator	Meaning
Arithmetic	B	Bit alignment (shift)
	+	Addition
	-	Subtraction
	*	Multiplication
	/	Division
Logical	&	Logical AND
	!	Inclusive OR
Relational	==	Equal
	>=	Greater than or equal
	>	Greater than
	<	Less than
	<=	Less than or equal
	<>	Not equal

The assembler distinguishes the bit shift operator *B* from the ordinary ASCII *B* in two ways. A bit shift operator is implied if the preceding atom is a single-precision integer, or if the *B* immediately follows a right parenthesis; e.g., 8B7 or (377) B7.

Break Characters

Break characters are used primarily as separators. They are:

□ represents the class of spaces -- a space, a comma, a horizontal tab, or any number or combination of spaces, commas, horizontal tabs. The meaning of □ is changed if a colon or equal sign *((:)* or *(=))* immediately follows it. We use □ only where we must; normally, spaces are obvious in the formats.

Note that when a *macro call* references arguments, comma(s) and space(s)/tab(s) do not produce the same results. For example, the macro calls:

```
TEST 1 2
TEST 1,2
```

□ each have two arguments, and assemble the same way; but the calls:

```
TEST,1,2
TEST,1 2
```

each have three arguments; the first a null. Also, a space *with* a comma in a macro call may produce a format error, because the assembler will try to use it to expand the macro; e.g.,

```
TEST 1, 2
```

See Chapter 5 for more on break character usage in macros.

: A colon (:) defines the symbol preceding it, for example COUNT:0

= An equal sign also defines the symbol preceding it, for example DATA3=4*DATA

( ) Parentheses may enclose a symbol or an expression.

[ ] Square brackets may enclose the actual arguments of a macro call; for example

```
MYMACRO [3,4)
5,6]
```

; A semicolon indicates the beginning of a comment string (string mode); for example

```
LDA 0,@20 ;GET NEXT ADDRESS IN TABLE.
```

) A carriage return terminates a line of source code; for example

```
MOVZR 0,0,SNR ;CHECK FOR ODD NUMBER )
```

↓ A form feed also terminates a line of source code.

Number Atoms

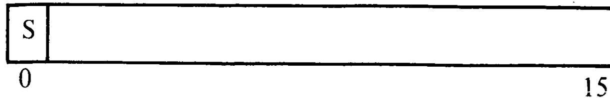
Three types of numbers are defined for the Macroassembler. They are:

1. Single-precision integer - stored in one word.
2. Double-precision integer - stored in two words.
3. Single-precision floating-point constant - stored in two words.

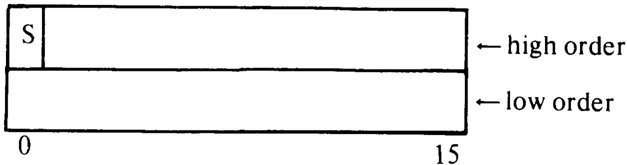
You can use single-precision integers in expressions and data statements, but you can use double-precision integers and floating-point numbers only in data statements.

Number Representations

A single-precision integer is represented as a single word of 16 bits, in the range 0 to 65,535 (0 to 177777_8). The integer may be interpreted as signed, using two's complement arithmetic. If bit 0 equals 0, it indicates a positive integer; if bit 0 equals 1, it indicates a negative integer, in the range 0 - 32767.

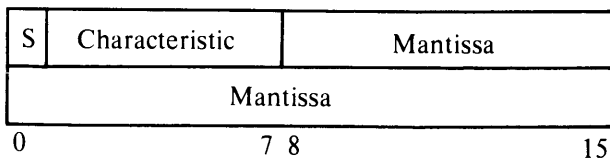


A *double-precision* integer is represented in memory in two contiguous words, where the first word is the high-order word. Using two's complement notation, a double-precision integer is represented as:



Bit 0 of the high-order word is the sign bit.

A *single-precision floating-point constant* is represented in memory in two contiguous words having the format:



Bit 0 of the high-order word is the sign bit, set to zero for positive numbers and set to one for negative numbers.

The integer characteristic is the integer exponent of 16 in excess-64_{10} (100_8) code. Exponents from -64 to +63 are represented by the binary equivalents of 0 to 127_{10} (0 to 177_8). Zero exponent is represented as 100_8.

The mantissa is represented as a 24-bit binary fraction. It can be viewed as six 4-bit hexadecimal digits. The range of the mantissa's magnitude is:

$16^{-1} <$ mantissa $< (1-16^{-6})$

The negative form of a number is obtained by complementing bit 0 (from 0 to 1 or 1 to 0). The characteristic and mantissa remain the same. When an expression is evaluated as zero, it is represented as true zero, two words of all zeroes in sign, characteristic, and mantissa.

The range of magnitude of a floating-point number is:

$16^{-1} * 16^{-16} <$ floating-number $< (1-16^{-6}) * 16^{63}$

which is approximately

$5.4 * 10^{-79} <$ floating-number $< 7.2 * 10^{75}$

Most routines that process floating-point numbers assume that all nonzero operands are normalized, and they normalize a nonzero result. A floating-point number is considered normalized if the fraction is greater than or equal to 1/16 and less than 1. In other words, it has a 1 in the first four bits (8-11) of the high-order word. All floating-point conversions by the assembler are normalized.

Single-Precision Integer Representation

The source format of a single-precision integer is:

{ [+] / [-] } d [d ... d] [.] break

where: each *d* is a digit within the range of the current input radix. The initial d must be in the range 0-9.

break is any character or digit outside the range of the current radix, or a period (".").

If the decimal point precedes the break character, the integer is evaluated as decimal. If there is no decimal point, the integer will be evaluated in the current input radix. The range of input radix values is 2 through 20 as set by the .RDX pseudo-op. The following table shows digit representation.

If your highest digit will be	the digit value will be	and your radix must be > =
0	0	any
1	1	any
2	2	3
3	3	4
4	4	5
5	5	6
6	6	7
7	7	8
8	8	9
9	9	10
A	10	11
B	11	12
C	12	13
D	13	14
E	14	15
F	15	16
G	16	17
H	17	18
I	18	19
J	19	20

If the input radix is 11 or greater, a number that would normally begin with a letter must be preceded by an initial zero to distinguish the number from a symbol. The following example shows how to represent the decimal numbers 15, 255, 4095, and 65,535 in hexadecimal. The source representation is shown in the right column, and the created storage word is shown in the left column.

```
000020  .RDX 16
000017  0F      ;Decimal 15.
000377  0FF     ;Decimal 255.
007777  0FFF    ;Decimal 4095.
177777  0FFFF   ;Decimal 65535.
```

Normally, you terminate a single-precision integer by one of the following operators or terminals:

Operators: + - * / B)&
== <> <= >= ><

Terminals: □)) ;

Note the following exception. The bit shift operator *B* will be interpreted as a digit if the radix is 12 or greater. To force the assembler to interpret *B* as a bit operator, use the backarrow (←) convention. This breaks the number string and is then ignored. The following example illustrates this. As usual, the left column indicates the storage word, the right column the source line (as in an assembly listing).

```
000020          .RDX 16
025423 02B13    ;B represents digit
                ;11 (2B13=25423 octal).

000010 02_B13   ;B represents bit shift
                ;operator.
```

Within an expression, one integer may have the current radix while another is given in radix 10 by the decimal point convention. Some assembled expressions which use single-precision integers of different radixes are shown below.

```
000002 .RDX 2
000012 101+101

000010 .RDX 8
000202 101+101

000012 .RDX 10
000312 101+101

000020 .RDX 16
001002 101+101
```

Special Formats of Single-Precision Integers

There is a special input format that converts a single ASCII character to its single-precision 7-bit octal value. The input format is:

"*a*

where: *a* represents any ASCII character except line feed (012_8), rubout (177_8) or null (000).

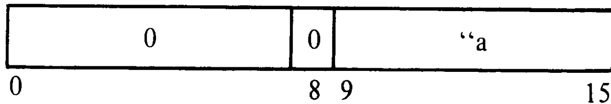
Only the single ASCII character immediately following the quotation mark is interpreted. The ASCII characters null, line feed, and rubout are invisible to the assembler, and cannot be input with this format. All other ASCII characters can be converted to single-precision integers:

```
000101 "A
000065 "5
000045 "%
000100 "@
```

The format can also be used as an operand within an expression.

```
000103 "A+2
000026 "B/3
177751 "*-"A
```

In every case, ") assembles an octal 15 and also terminates the line. An "a format character is packed this way:

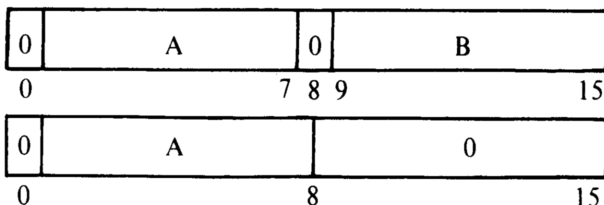


A second format, which uses apostrophes, can convert up to two ASCII characters to a single-precision integer. The format is:

"string" or *"string"*)

where: *string* consists of any number of ASCII characters; only the first two characters will generate a 16-bit value.

String characters, unlike an "a character, are packed left to right in the word:



Two apostrophes without an intervening character will generate a word containing absolute (as opposed to relocatable) zero.

You may use special formats wherever integers are allowed. Some simple expressions using the string format follow:

```
040502 'AB'
041101 'BA'
020040 ' '
000003 ''+5-2
041005 'B'+5
020101 ' A'
040501 "A+'A'
```

A return entered before the second apostrophe terminates the *"string"* format. For example:

```
006400 '
040415 'A
040502 'AB
```

Double-Precision Integer Representation

A double-precision integer has the following source format:

{ *[+]* / *[-]* } d *[dd ... d] [.]* D break

where: each d is a digit within the current radix. The initial d must be in the range 0-9.

The character D before the break character indicates a double-precision integer.

The optional decimal point tells the assembler that the integer is decimal.

break is a terminal character (typically □ or ; or)) that indicates the end of the integer.

An operator may not terminate a double-precision integer; if it does, a format error (F) will result.

The radix of a double-precision integer may be in the range 2 - 20. If the radix is greater than or equal to 14, the letter *D* will be interpreted as a digit. To force the assembler to interpret *D* as indicating double-precision, use the ← (backarrow) convention:

```
000020  .RDX 16
000455  12D     ;D represents digit
                ;13 (decimal).
000000  12_D    ;12 is a double-prec-
000022          ;ision integer.
```

On some consoles, you enter the backarrow as a shift-O.

Some assembled data statements which contain double-precision integers are:

```
000010  .RDX 8
000000  1D
000001
177777  -1D
177777
000001  200000D
000000
000004  262147.D
000003
000001  100000.D
103240
```

Single-Precision Floating-Point Constants

Much of the floating-point number format is optional. The minimal format of a floating-point number is one digit in the range 0 to 9, followed by either a decimal point or the letter E (exponent), followed by one digit in the range 0 to 9. The minimal floating-point format is:

d { . E } d break

where: d is a digit in the range 0 - 9.

A single-precision floating-point number is represented in source format as:

{ + - } d *[d..d]* . d *[d..d] [E[* { + - } *d]d]*

{ + - } d *[d..d]* E { + - } d *[d]* break

where: each *d* is a digit 0 to 9. The mantissa and exponent are always converted in decimal (e.g., 2E9 = > 2 * 10^9.

One or two digits may represent an exponent following the letter E.

break is typically one of the terminals: □ or ; or)

You can format the same floating-point number with the letter E, the decimal point, or both as shown below:

```
041376  254.33
052172
041376  254.33E0
052172
041376  25433E-02
052172
041376  25433E-2
052172
041376  2543.3E-1
052172
```

If the current radix is 15 or larger, the assembler will interpret the letter E preceding number as an integer in the current radix rather than as a floating-point number. To avoid this ambiguity, use the ← convention (backarrow, ASCII 137); for example:

```
000020  .RDX 16
155035  -25E3  ;E is hex 14.
142141  -25_E3 ;E indicates
124000         ;floating point.
```

Examples of floating point constants in source statements, with resulting stored values, follow.

```
        000010  .RDX 8

00000 040420  1.0    ;Note
      000000         ;location
                     ;counter.
00002 040462  3.14159
      041763

00004 140420  -1E0
      000000

00006 040200  +5.0E-1
      000000

00010 041421  +273.0E0
      010000
```

Examples of Numbers

Some additional source program numbers and their assembled values follow.

```
000020  .RDX 16
053175  567D    ;Hex single-prec-
                ;ision integer.
000000  567_D   ;Hex double-prec-
002547          ;ison integer.
001067  567.    ;Decimal single-
                ;precison integer.
000000  567._D  ;Decimal double-
001067          ;precison integer.
002547  567     ;Hex single-
                ;precision integer.
005316  567_B14  ;Hex single-
                ;precision int, bit
                ;shifted one bit.
012634  567_B13  ;Hex single-prec-
                ;ision integer, bit
                ;shifted two bits.
042026  567_E1  ;Floating-point
023000          ;constant (decimal).
```

Symbols

A primary function of the assembler is the recognition and interpretation of symbols. Symbols may direct the action of the assembler or they may represent numeric values. The various classes of symbols will will be discussed in Chapter 3. Their source representation is given below:

a *[b...b]* break

where: a can be one of the characters A through Z or . or ?.

b can be one of the characters A through Z , or 0 through 9 or . or ? or _ (underscore).

break is any character not rated above; e.g., space, comma, etc.

By default, MAC recognizes only the first five characters before the *break* character, although it prints more than five on listings. You *can* specify eight-character symbols with the MAC global /T switch (described in Chapter 6), which produces an *extended* (as opposed to a standard) RB file.

Special Characters

The characters @, # , and ** are transparent during an assembly line scan. These atoms affect a line after it has been scanned. See the appropriate reference manual for your computer for more on @ and # .

Other special characters are a set of square brackets surrounding a symbol (e.g., [MYSYM]), a dollar sign in a macro definition (e.g., TR$=), and a backslash followed by a symbol (e.g., \ ONES). See the .GOTO pseudo-op (Chapter 4) and Chapter 5 for more on these atoms.

@

Commercial AT Sign

In a source program line of memory reference instruction (MRI), in an extended memory reference instruction, or before an expression, a commercial "at" sign (@) (or a series of @ signs) will set bits in the following ways:

1. When the rest of the MRI has been evaluated, an @ sign anywhere in the instruction stores a 1 in bit 5. In the MRI format, bit 5 is the indirect addressing bit.

```
020020  LDA 0, 20
022020  LDA 0, @ 20
```

2. In the data word format, bit 0 is the indirect addressing bit. When the expression has been evaluated, an @ sign sets bit zero of the word to 1.

```
000025  25
100025  @25
```

3. An @ sign in an extended memory instruction sets bit 0 in the second word of the instruction.

```
00000 103470    EJMP     0, 3
      000000
00002 103470    EJMP @   0, 3
      100000
```

#

Number Sign

A number sign (#) may appear in an ALC instruction. When the rest of the ALC has been evaluated, # causes the assembler to store a 1 in bit 12, the no-load bit.

```
00000 101123    MOVZL    0, 0, SNC

00001 101133    MOVZL # 0, 0, SNC
```

**

Asterisks

Two consecutive asterisks *(**)* at the start of a source program line will suppress the listing of that line.

```
;Source program:
        LDA 0, 0, 2
    **  LDA 1, 1, 2
        LDA 0, 0, 3
        .END
;Listing:
00000 021000    LDA 0, 0, 2
00002 021400    LDA 0, 0, 3
                .END
```

Note that the relative location numbers jump from "0" to "02" since all lines of source are assembled but the second source line is not listed.

```
**.NOLOC 0
** PASSWORD: 012345
```

End of Chapter

Chapter 3
Syntax

Expressions

An expression has the format:

[$operand_1$] operator $operand_2$

where: operator is a Macroassembler operator.

$operand_1$ and $operand_2$ may be single-precision integers, or symbols, or expressions evaluated to single-precision integers. An operand must precede each operator, except for the unary operators + and -. Either unary operator may follow an operator or precede an expression. Note that spaces are *not* allowed in expressions.

Operators

The Macroassembler operators are:

	Operator	Meaning
Arithmetic/ Logical	B	Bit alignment
	+	Addition, e.g. (2 + 3), or unary plus, e.g., (+ 3)
	-	Subtraction, e.g., (5-2), or unary minus, e.g., (-7)
	*	Multiplication
	/	Division
	&	Logical AND. The result in a given bit position is 1 only if $operand_1$ = 1 and $operand_2$ = 1.
	!	Inclusive OR. The result in a given bit position is 1 if either or both operands is 1.

	Operator	Meaning
Relational	= =	Equal to
	< >	Not equal to
	< =	Less than or equal to
	<	Less than
	> =	Greater than or equal to
	>	Greater than

More than one type of operator may appear in an expression. Operators are evaluated in the order of their priority:

Operator	Priority Level
B	1 (highest priority)
+ - * / & !	2
< <= > >= == < >	3 (lowest priority)

When operators are of equal priority, they are evaluated from left to right. Parentheses can be used to alter priority; an expression in parentheses is evaluated first. Expressions are evaluated with no check for overflow. An expression containing one of the operators

< <= > >= == < >

is a relational expression. It evaluates either to absolute zero (false) or absolute one (true). These values are called "absolute" because they are not relocatable.

Examples:

```
000010  .RDX 8
000025  A=25
177763  B=-15
000000  A==B     ;False (0) since A
                 ;doesn't equal B.
000001  A<>B     ;True (1) since
                 ;A doesn't equal B.
000001  A+B-10==A-(2*10+5) ;True,
                 ;since 0=0.
000001  A==(-B)+10   ;True,
                 ;since 25=25.
000000  A==(-B)&A  ;False, since
                 ;AND of (-B) and
                 ;A doesn't equal A.
```

Bit Alignment Operator

When the bit alignment operator is used, $operand_1$ preceding *operator B* is the value to be aligned; $operand_2$ following *operator B* represents the right-most bit to which $operand_1$ is aligned. The value of $operand_2$ has the range

$0 < operand_2 < 15_{10}$

The following formula determines the result of a bit alignment operator:

for $operand_1$ *B* $operand_2$, the resultant value will be $operand_1 * 2^{**}(15. - operand_2)$

where: $operand_2$ is implicitly evaluated in decimal unless parentheses are used; e.g.,

```
.RDX 8
1B15=000001
1B(15)=000004
```

The B operator can be misread as a symbol or part of a symbol. If the operand preceding the operator is a symbol, the operand must be enclosed in parentheses to avoid this misinterpretation. Some examples of bit alignment operations are:

```
    000025  A=25     ;The radix is 8.
    100000  (A)B0    ;The right-most bit
                     ;of 25 is in bit pos-
                     ;ition 0-- the rest
                     ;of "25" is lost.
    124000  (A)B4    ;The right-most bit
                     ;of 25 is in bit
                     ;position 4.
    012400  (A)B7    ;Here, in pos. 7.
    000124  (A)B13   ;And so on.
    000025  (A)B15
N   000000  (A)B16   ;Note N error-there
                     ;is no bit 16.
```

Parentheses around $operand_1$ and $operand_2$ will ensure that the correct value is aligned properly. Parentheses affect operands as shown below.

```
   000025  A=25
   000010  C=10
;"B(3+C)" means "align at bit number
;13 octal, 11 decimal.

   000640  (A-C)*2B(3+C)   ;32 octal is
                           ;aligned at bit 11
   000640  (A-C*2)B(3+C)   ;Same.
   177425  A-(C*2)B(3+C)   ;(C*2)B(3+C)
                           ;equals 400. 25-400
                           ;equals 177425.
   000640  A-C*2B(3+C)     ;32 octal is
                           ;aligned at bit 11.
```

Examples of Expressions

Some examples of expression evaluation are:

```
000025  A=25
000015  B=15
000010  A*(B-10)/B      ;In decimal,
                        ;105/13=8--discard
                        ;remainder in integer
                        ;arithmetic.
000015  A&B/A!B         ;A&B=5, 5/15=0, 0!B=B,
                        ;Q.E.D.
000001  (A-10)==B ;True (1) since 15=15.

000016  A/B+B   ;25/15=1, 1+15=16.

000000  A&B/(A!B) ;5/35=0.

000001  ((B/A)+5)>0 ;15/25=0, 0+5 >0,
                    ;thus true (1).
```

Relocation Properties of Expressions

Each operand in an expression has a relocation property. The relocation property of the expression's result depends upon the relocation properties of its operands. Thus far in this manual, expressions have had absolute operands which produced absolute results.

A value, however, may have one of five properties. These are:

absolute

page zero (ZREL) relocatable

page zero (ZREL) byte-relocatable

NREL code

NREL code, byte-relocatable

RLDR makes a relocatable value absolute by adding a relocation constant (called c) during the loading procedure. The relocation constant is added once if the value is word relocatable, and twice if the value is doubly-relocatable (byte-relocatable).

You cannot use certain relocatable operands (such as ZREL and normal relocatable) together in one expression. However, some mixing of similar relocation properties is permitted. The relocation properties of operands and the relocation value of the results are listed below. In the list,

a — represents an absolute value

r — represents a relocatable value (either ZREL or NREL code)

2r — represents a byte-relocatable value (either ZREL or NREL code)

kr — represents a relocatable value that can be converted to an absolute value by addition of a relocation constant, "c", k times. However, if the final value of an expression is k-relocatable, the statement is flagged with a relocation error (R).

NOTE: In the following list, the *&* and *!* operators indicate logical AND and inclusive OR, respectively.

Expression	Relocation
a+a	a
a+r	r
r+r	2r
nr+mr	(n+m)r
a-a	a
r-a	r
a-r	-1r
r-r	a
nr-mr	(n-m)r
a*a	a
a*r	ar
r*r	Illegal
a/a	a
kr/a	(k/a)r (only if k/a yields no remainder)
a/r	Illegal
a&a	a
a!a	a
r&r	Illegal
a&r	Illegal
r!r	Illegal
a!r	Illegal

All expressions involving the operators < = < > = > == or < > result in an absolute value of either zero (false) or one (true). When operands in these expressions have different relocation properties, all comparisons will result in a value of absolute zero (false) except when the operator is < > (not equal to).

Given these rules, expressions that result in a value of "a", "r", or "2r" are legal. Expressions that do not evaluate to a legal relocation property will be flagged as relocation errors (R).

The example below shows the relocation properties of expressions; the assembler cross-reference showing the relocation properties of each symbol is included.

```
          000002        A=2
                        .NREL
    00000'000020'       .+20 ;Normal relocatable.
                             ;
    00001'000000   R:   0    ;R's address is relocatable, but its contents
                             ;aren't, thus no final '.
          000002'  S=   R+1  ;Relocatable-Operator-Absolute is relocatable.
                             ;
    00002'000001        A/A  ;Absolute-Operator-Absolute is Absolute.
                             ;
    00003'000002"       R+R  ;Reloc-Operator-Reloc is Byte-Relocatable.
                             ;
    00004'177777'       R-A  ;Reloc-Oper-Absolute is Relocatable.
                             ;
    00005'000001        S-R  ;Reloc-Oper-Reloc is Absolute.
                             ;
   R00006'000000'       A!R  ;Operators & and ! require absolute
                             ;operands.
                        .END

   0002 .MAIN

   A     000002         1/01    1/07    1/09    1/11
   R     000001'        1/04    1/06    1/08    1/09    1/10    1/11
   S     000002'        1/06    1/10
```

Symbols

The macroassembler recognizes three classes of symbols:

1. Permanent
2. Semipermanent
3. User

To understand the assembly process, you must understand the difference between these classes.

Permanent Symbols

Permanent symbols are defined within the assembler and cannot be altered in any way. These symbols serve two purposes: 1) they direct the assembly process; and, 2) they represent numeric values of internal assembler variables.

Symbols which direct the assembly process are called pseudo-ops. Among other purposes, pseudo-ops set the input radix for numeric conversions, set the location counter mode, and assemble ASCII text. Chapter 4 describes pseudo-ops in detail.

Other permanent symbols represent numeric values of internal assembler variables. For example, the symbol *.PASS* represents the current pass number. On the first assembly pass its value is 0, while on the second its value is 1.

If a symbol could be either a pseudo-op or a value, the assembler recognizes the intended use by the symbol's position in a line. If the first atom of a line is a pseudo-op, it directs the assembler. If the pseudo-op atom occurs anywhere else in the line, it represents a value. A few examples will illustrate these rules.

The assembler pseudo-op *.TXTM* directs the packing of text bytes within a word. The two methods are left/right and right/left. The directive takes the form:

.TXTM *expression*

If *expression* evaluates to zero (the default mode), bytes are packed right/left. If *expression* evaluates to nonzero, bytes are packed left/right.

Example 1

The line

.TXTM 1)

directs the assembler to pack bytes left/right.

Example 2

When enclosed in parentheses,

(.TXTM))

assembles a storage word, which contains the value of the last expression used to set the text mode.

Example 3

In the following usage:

```
      000001    .TXTM 1
00000 000005    +.TXTM+4
```

The first line sets text mode to pack left/right while the second line generates a storage word containing absolute (nonrelocatable) 5. (Note that the first atom of the second line is +.)

Appendix B lists all permanent symbols and Chapter 4 describes each one. These symbols must be used as described in this document; they cannot be redefined. Permanent symbols will never be printed in the cross-reference listing.

Semipermanent Symbols

Semipermanent symbols form a very important class of symbols usually thought of as *operation codes*. Symbols may be defined as semipermanent with appropriate pseudo-ops; these symbols imply future syntax analysis. For example, a symbol may be defined as "requiring an accumulator". This symbol will cause the assembler to scan for an expression following the symbol. If no expression is found, a format error results. If found, the value of the expression sets the accumulator field bit positions to given a 16-bit instruction value. Instruction values are discussed in Instructions later in this chapter.

Semipermanent symbols can be saved and used, without redefinition, for all subsequent assemblies. The DGC assembler contains a number of semipermanent symbols defined specifically for the DGC instruction set. You can eliminate these symbols and define your own set, or you can add to the given set (see Chapter 6).

Semipermanent symbols are not printed in the cross-reference listing unless enabled by the global /A switch (Chapter 6).

User Symbols

You can define any symbol that does not conflict with permanent or semipermanent symbols. User symbols serve many purposes: to symbolically name a location, to assign a numeric parameter to a symbol, to name external values, to define global values, and so on. These user symbols are maintained during assembly in a disk file table that is printed after the assembly source listing.

User symbols can be further classified as local or global. *Local symbols* have a value which is known only for the duration of the single assembly in which they are defined. The value of *global symbols* is known at load time, and thus they may be used for intermodule communication. The assembler always includes global symbols in its RB output; you can instruct it to include local symbols in the RB via the global /U switch (Chapter 6).

By default, MAC recognizes only the first five characters in any user symbol, although it prints longer symbol names on the program listing. The cross-reference listing *always* shows only the first five characters of a symbol. You can specify eight-character symbols with the global /T switch in the MAC command; this also produces an *extended* (as opposed to a standard) RB file. See Chapter 6, global /T switch, for more detail.

Instructions

An instruction is the assembly of one or more fields, initiated by a semipermanent symbol (called the "instruction mnemonic") to form a 16-bit or 32-bit value.

Fields in an instruction must conform in number and type to the requirements of the semipermanent symbol; they can be separated by a space, comma, or tab.

Data General computers recognize a number of instruction types. Each type has a pseudo-op and its own group of semipermanent symbols. These pseudo-ops are described in Chapter 4.

NOTE: Instructions marked with an asterisk (*) can be assembled on any machine, but will execute on ECLIPSE computers only.

Instructions fall into the following types:

Instruction Type	Defining Pseudo-op
Arithmetic and Logical (ALC). (e.g., ADD)	.DALC
*Extended ALC - 2 accumulators, no skip (e.g., IOR)	.DISD
*Extended ALC - 2 accumulators, skip (e.g., SGT)	.DISS
I/O without Accumulator (e.g., SKION)	.DIO
I/O with Accumulator (e.g., DIA)	.DIOA
I/O without Device Code (e.g., RPT)	.DIAC
Memory Reference (e.g., JMP)	.DMR
*Extended Memory Reference (e.g., EJMP)	.DEMR
Memory Reference with Accumulator (e.g., LDA)	.DMRA
*Extended Memory Reference with Accumulator (e.g., ELDA)	.DERA
*Commercial Memory Reference (e.g., ELDB)	.DCMR
Count and Accumulator (e.g., ADI)	.DICD
*Extended Immediate (e.g., ADDI)	.DIMM
*Extended Memory without Argument Fields (e.g., SAVE)	.DEUR
*Extended Memory Operation (e.g., XOP)	.DXOP
*Floating-Point Load/Store (e.g., FLDS)	.DFLM
*Floating-Point Load/Store, no accumulator (e.g., FLST)	.DFLS
Define a User Symbol as Semipermanent without Argument Fields	.DUSR

Arithmetic and Logical (ALC) Instructions

An arithmetic and logical (ALC) instruction is implied by one of the following instruction mnemonics:

```
COM  MOV  ADC  ADD
NEG  INC  SUB  AND
```

The format of the source program instruction is:

alc-mnemonic *[c][s]*□source-ac□dest-ac *[*□ *skip]*

where: alc-mnemonic is one of the eight semipermanent symbols listed above.

c is an optional carry mnemonic (Z, O, or C).

s is an optional shift mnemonic (S,L, or R).

source-ac specifies the source accumulator - 0, 1, 2, or 3.

destination-ac specifies the destination accumulator - 0, 1, 2, or 3.

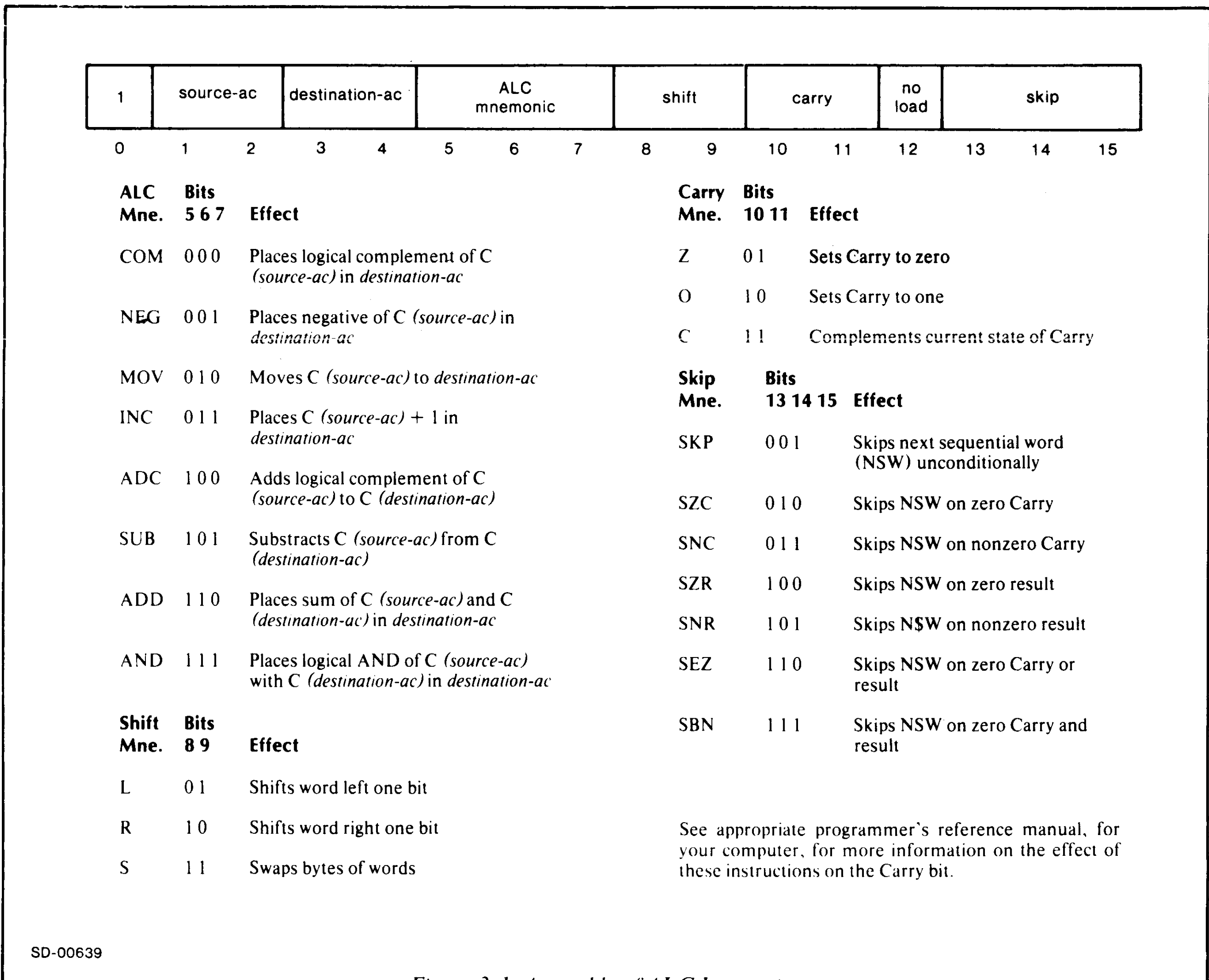
skip is an optional skip mnemonic SNR, SZR, SNC, SEC, SKP, SBN, or SEZ.

In addition, the atom # (number sign) can be specified anywhere in the source line as a break character. This atom assembles a 1 at bit 12, the no-load bit, and this prevents the destination-ac from being loaded and leaves the carry unchanged.

Figure 3-1 shows each assembled ALC instruction, its bit pattern, and the effect of the instruction, shft, carry, and skip mnemonics.

Examples of ALC instructions follow.

```
107000  ADD 0, 1
112412  SUB #  0, 2, SZC
146000  ADC 2, 1
101123  MOVZL 0, 0, SNC
120014  COM # 1, 0  SZR
```



1	source-ac		destination-ac		ALC mnemonic			shift		carry		no load	skip		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

ALC Mne.	Bits 5 6 7	Effect
COM	0 0 0	Places logical complement of C *(source-ac)* in *destination-ac*
NEG	0 0 1	Places negative of C *(source-ac)* in *destination-ac*
MOV	0 1 0	Moves C *(source-ac)* to *destination-ac*
INC	0 1 1	Places C *(source-ac)* + 1 in *destination-ac*
ADC	1 0 0	Adds logical complement of C *(source-ac)* to C *(destination-ac)*
SUB	1 0 1	Substracts C *(source-ac)* from C *(destination-ac)*
ADD	1 1 0	Places sum of C *(source-ac)* and C *(destination-ac)* in *destination-ac*
AND	1 1 1	Places logical AND of C *(source-ac)* with C *(destination-ac)* in *destination-ac*

Shift Mne.	Bits 8 9	Effect
L	0 1	Shifts word left one bit
R	1 0	Shifts word right one bit
S	1 1	Swaps bytes of words

Carry Mne.	Bits 10 11	Effect
Z	0 1	Sets Carry to zero
O	1 0	Sets Carry to one
C	1 1	Complements current state of Carry

Skip Mne.	Bits 13 14 15	Effect
SKP	0 0 1	Skips next sequential word (NSW) unconditionally
SZC	0 1 0	Skips NSW on zero Carry
SNC	0 1 1	Skips NSW on nonzero Carry
SZR	1 0 0	Skips NSW on zero result
SNR	1 0 1	Skips N$W on nonzero result
SEZ	1 1 0	Skips NSW on zero Carry or result
SBN	1 1 1	Skips NSW on zero Carry and result

See appropriate programmer's reference manual, for your computer, for more information on the effect of these instructions on the Carry bit.

SD-00639

Figure 3-1. Assembly of ALC Instruction

I/O Instructions without Accumulator

An input/output instruction without an accumulator field is implied by one of the following instruction mnemonics:

NIO SKPBN SKPDN
SKPBZ SKPDZ

The format of the source program instruction is:

io-mnemonic *[busy/done]* □device-code

where: io-mnemonic is one of the five semipermanent symbols listed above.

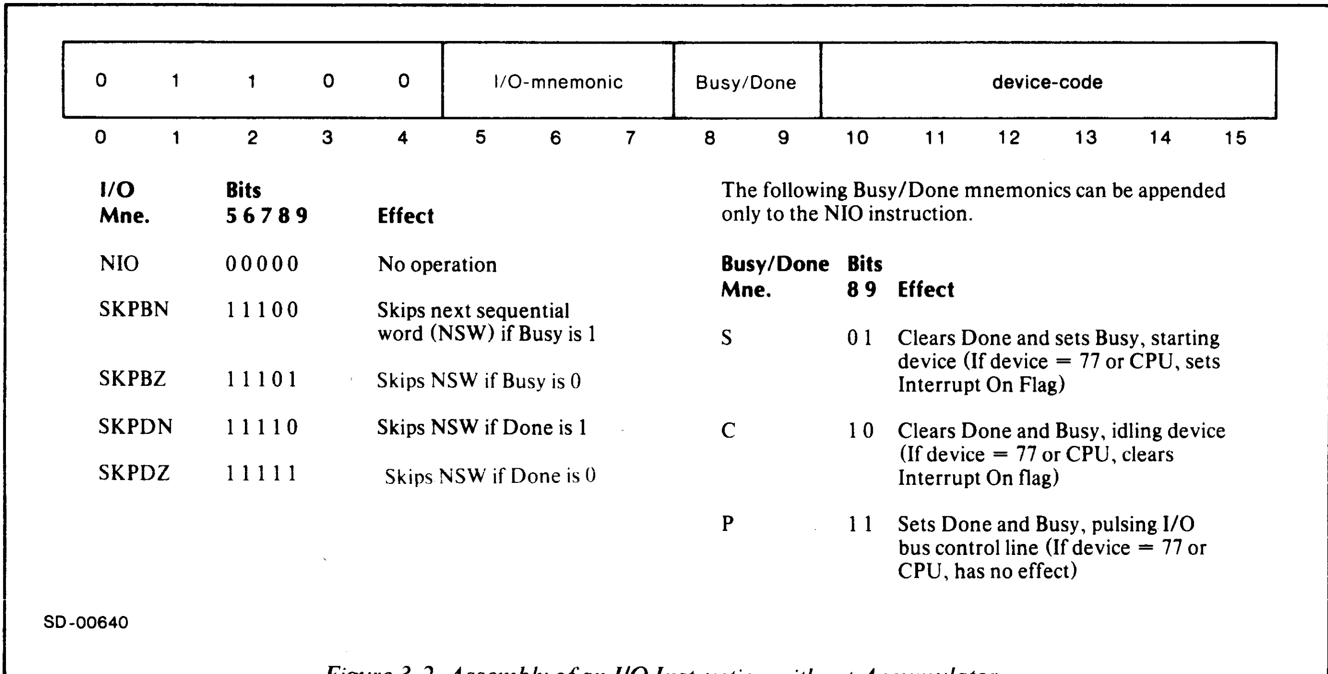
busy/done is an optional Busy/Done bit mnemonic (NIO instruction only).

device-code is any legal expression evaluating to an integer specifies a device.

Figure 3-2 shows each assembled I/O instruction, its bit pattern, the effect of the instruction and Busy/Done mnemonics.

Examples of I/O Instructions without an accumulator follow:

```
060112  NIOS 12
060112  NIOS  PTR
060177  NIOS CPU
060177  NIOS 77
```



I/O Mne.	Bits 5 6 7 8 9	Effect
NIO	0 0 0 0 0	No operation
SKPBN	1 1 1 0 0	Skips next sequential word (NSW) if Busy is 1
SKPBZ	1 1 1 0 1	Skips NSW if Busy is 0
SKPDN	1 1 1 1 0	Skips NSW if Done is 1
SKPDZ	1 1 1 1 1	Skips NSW if Done is 0

The following Busy/Done mnemonics can be appended only to the NIO instruction.

Busy/Done Mne.	Bits 8 9	Effect
S	0 1	Clears Done and sets Busy, starting device (If device = 77 or CPU, sets Interrupt On Flag)
C	1 0	Clears Done and Busy, idling device (If device = 77 or CPU, clears Interrupt On flag)
P	1 1	Sets Done and Busy, pulsing I/O bus control line (If device = 77 or CPU, has no effect)

Figure 3-2. Assembly of an I/O Instruction without Accumulator

I/O Instructions with Accumulator

An input/output instruction with an accumulator field is implied by one of the following instruction mnemonics:

DIA DIB DIC
DOA DOB DOC

The format of the source program instruction is:

ioa-mnemonic *[busy/done]* □ac □device-code

where: ioa-mnemonic is one of the six semipermanent symbols listed above.

busy/done is an optional Busy/Done bit mnemonic.

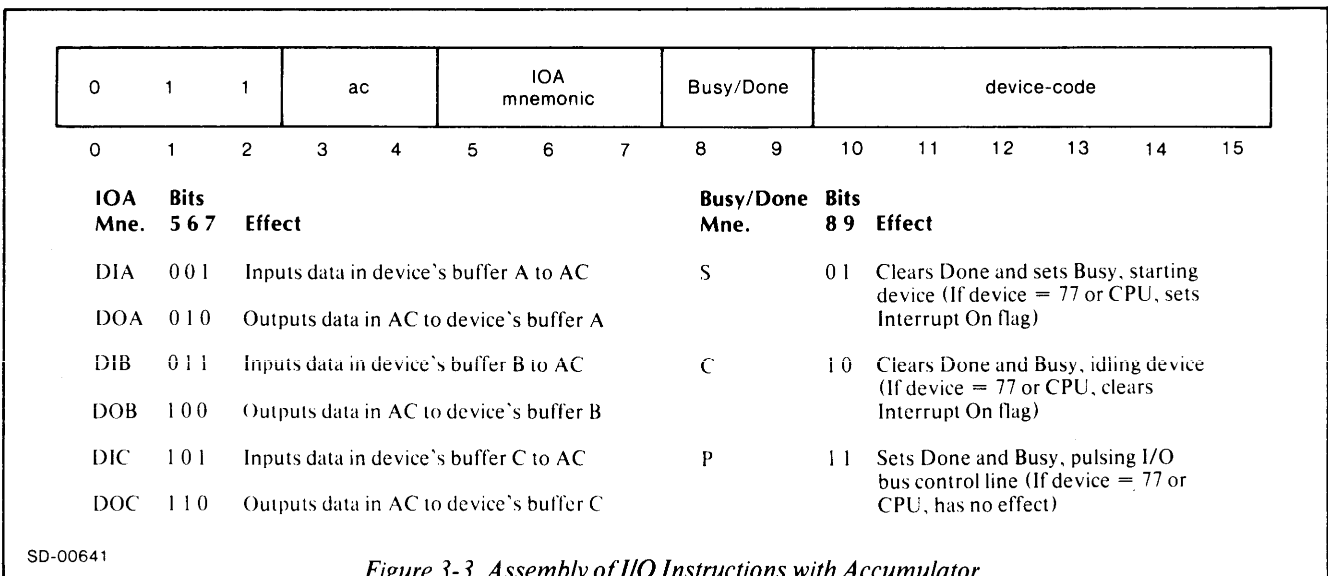
ac is a 0, 1, 2, or 3, indicating the accumulator to receive or supply the data.

device-code is any legal expression evaluating to an integer that specifies a device.

Figure 3-3 shows each assembled I/O instruction, its bit pattern, and the effect of the instruction and Busy/Done mnemonics.

Examples of I/O instructions with an accumulator field follow:

```
074477  DIA 3, CPU
070512  DIAS 2, PTR
063077  DOC 0, 77
```



IOA Mne.	Bits 5 6 7	Effect
DIA	0 0 1	Inputs data in device's buffer A to AC
DOA	0 1 0	Outputs data in AC to device's buffer A
DIB	0 1 1	Inputs data in device's buffer B to AC
DOB	1 0 0	Outputs data in AC to device's buffer B
DIC	1 0 1	Inputs data in device's buffer C to AC
DOC	1 1 0	Outputs data in AC to device's buffer C

Busy/Done Mne.	Bits 8 9	Effect
S	0 1	Clears Done and sets Busy, starting device (If device = 77 or CPU, sets Interrupt On flag)
C	1 0	Clears Done and Busy, idling device (If device = 77 or CPU, clears Interrupt On flag)
P	1 1	Sets Done and Busy, pulsing I/O bus control line (If device = 77 or CPU, has no effect)

Figure 3-3. Assembly of I/O Instructions with Accumulator

I/O Instructions without Device Code

Three common I/O instructions are defined with the CPU device code. These instructions require an accumulator field but have no device code field:

READS INTA MSKO HALTA (ECLIPSE only)

The format of the source program instruction is:

iac-mnemonic □ac

where: iac-mnemonic is one of the four semipermanent symbols listed above.

ac specifies which accumulator will receive or supply the data - 0, 1, 2, or 3.

The following I/O instructions are equivalent.

I/O Instruction Without Device Code	Equivalent Instruction
READS *accumulator*	DIA *accumulator*, CPU
INTA *accumulator*	DIB *accumulator*, CPU
MSKO *accumulator*	DOB *accumulator*, CPU
HALTA *accumulator*	DOC *accumulator*, CPU

Figure 3-4 shows each assembled I/O instruction, its bit pattern, and the effect of the instruction mnemonics.

Examples of I/O instructions without a device code follow:

```
074477   READS 3  ;Read console switch
                  ;positions into AC3.
061477   INTA 0 ;Read interrupt device
                  ;code into AC0.
062077   MSKO 0 ;Disable interrupts
                  ;from devices not
                  ;masked in AC0.
```

I/O Instructions without Argument Fields

Four common I/O instructions are defined as semipermanent symbols that require no argument field:

IORST INTEN INTDS HALT

The equivalent I/O instruction and effect of both instructions follow:

I/O Instruction Without Argument	Equivalent Instruction	Octal Value	Effect
IORST	DICC 0, CPU	062677	Clear all I/O devices and Interrupt On flag; reset clock to line frequency.
INTEN	NIOS CPU	060177	Set Interrupt On flag, enabling interrupts.
INTDS	NIOC CPU	060277	Clear Interrupt On flag, disabling interrupts.
HALT	DOC 0, CPU	063077	Halt the processor.

Figure 3-5. Assembly I/O Instructions without Argument Fields

Examples of these instructions follow:

```
062677   IORST
063077   HALT
060177   INTEN
```

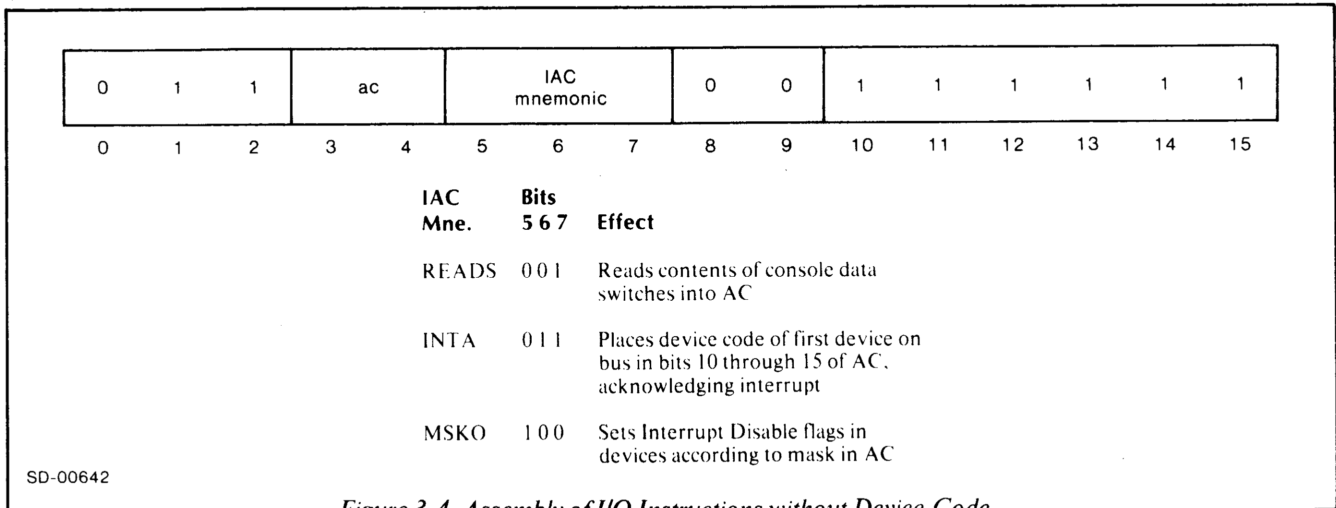


Figure 3-4. Assembly of I/O Instructions without Device Code

Memory Reference (MR) Instructions

MR Instructions without Accumulator

A memory reference (MR) instruction without an accumulator field is implied by one of the following instruction mnemonics:

JMP JSR ISZ DSZ

The format of the source program instruction is:

mr-mnemonic□address

or

mr-mnemonic□displacement□mode

where:

mr-mnemonic is one of the four semipermanent symbols listed above.

displacement is any legal expression evaluating to an 8-bit integer, ranging from -200_8 through $+177_8$.

mode is a 0, 1, 2, or 3, indicating a mode for forming an effective address (E). Mode 0 or 1 are implied by the format; you do not specify either explicitly. The assembler forms an effective address as follows:

Mode	Formation of Effective Address (E)
0	Address equals displacement.
1	Address is based on the contents of location counter: E=C(LC) +displacement and, therefore C(LC) $-200_8 < E < C(LC) + 177_8$
2	Address is based on the contents of AC2: E=C(AC2)+displacement and, therefore C(AC2)$-200_8 < E < C(AC2) + 177_8$
3	Address is based on the contents of AC3: E=C(AC3) +displacement and, therefore C(AC3)$-200_8 < E < C(AC3) + 177_8$

address is any legal expression evaluating to an 8-bit integer in one of the following ranges:

1. Page zero addressing: 0 through $+377_8$. Addressing is direct and E = address.
2. LC-relative addressing: C(LC)-200_8 through C(LC)$+177_8$. Address is based on the contents of the location counter and E=C(LC)+ address.

In addition, you can insert the atom @ in the source line address field as a break character. This atom assembles a 1 at bit 5, the indirect addressing bit. Thus, the effective address in the instruction is a pointer to another location, which may, in turn, contain an indirect address.

If only address is specified, the assembler determines if this address is in page zero (0 through 377_8) or within 177_8 words of the location counter. If the address is in page zero, bits 6 and 7 of the instruction word are set to 00 and the displacement field is set as follows:

1. If the address is absolute and fits in 8 bits, the displacement field is set to address.
2. If the address is page zero relocatable (that is, assembled with the .ZREL pseudo-op), the displacement field is set to address with page zero relocation, and the line is flagged with a dash (-) in column 16 of the source program listing.
3. If the address is an external displacement (that is, assembled with the .EXTD pseudo-op), the displacement is set to the assembler .EXTD number and the line is flagged with a $ in column 16 of the source program listing. (The assembler assigns each .EXTD a number, which RLDR uses to fill in the value of the external.)

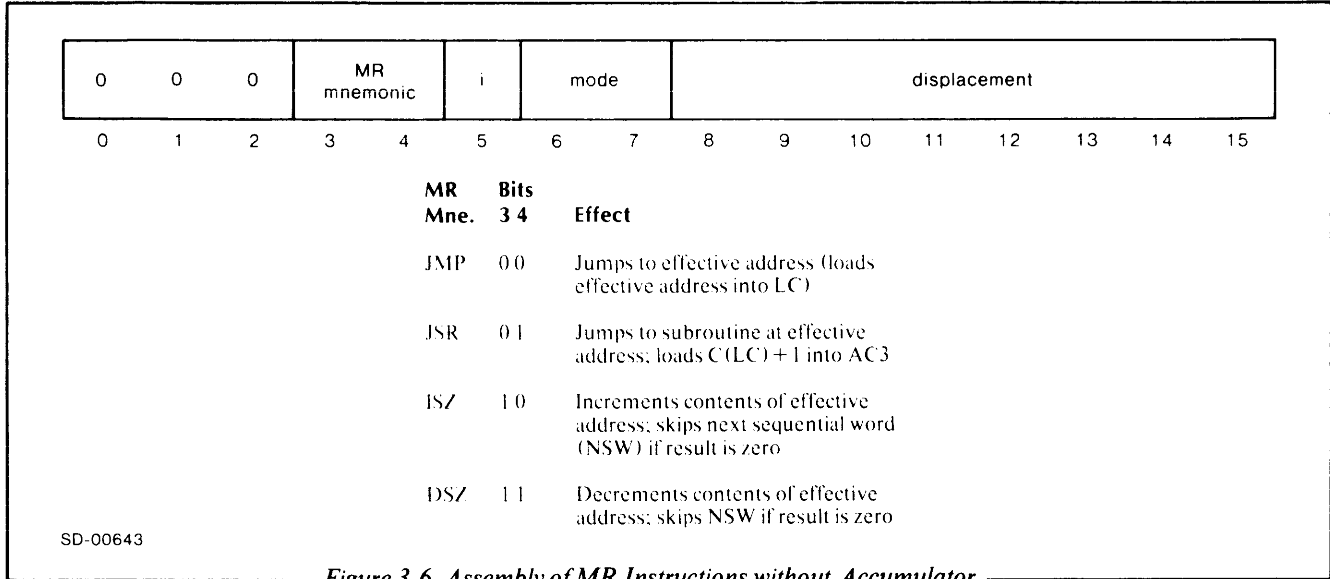
If address is within 177_8 words of the contents of the location counter, bits 6 and 7 are set to 01 and addressing is based on the current contents of the location counter (as in addressing mode 1). The displacement field of the instruction word is set to:

address - C(LC).

If address or the evaluation of displacement to an address does not produce an effective address within the appropriate range, an addressing error (A) is reported.

Figure 3-6 shows each assembled MR instruction, its bit pattern and the effect of the instruction mnemonics.

Figure 3-7 illustrates how effective addresses are formed.



MR Mne.	Bits 3 4	Effect
JMP	0 0	Jumps to effective address (loads effective address into LC)
JSR	0 1	Jumps to subroutine at effective address; loads C(LC) + 1 into AC3
ISZ	1 0	Increments contents of effective address; skips next sequential word (NSW) if result is zero
DSZ	1 1	Decrements contents of effective address; skips NSW if result is zero

Figure 3-6. Assembly of MR Instructions without Accumulator

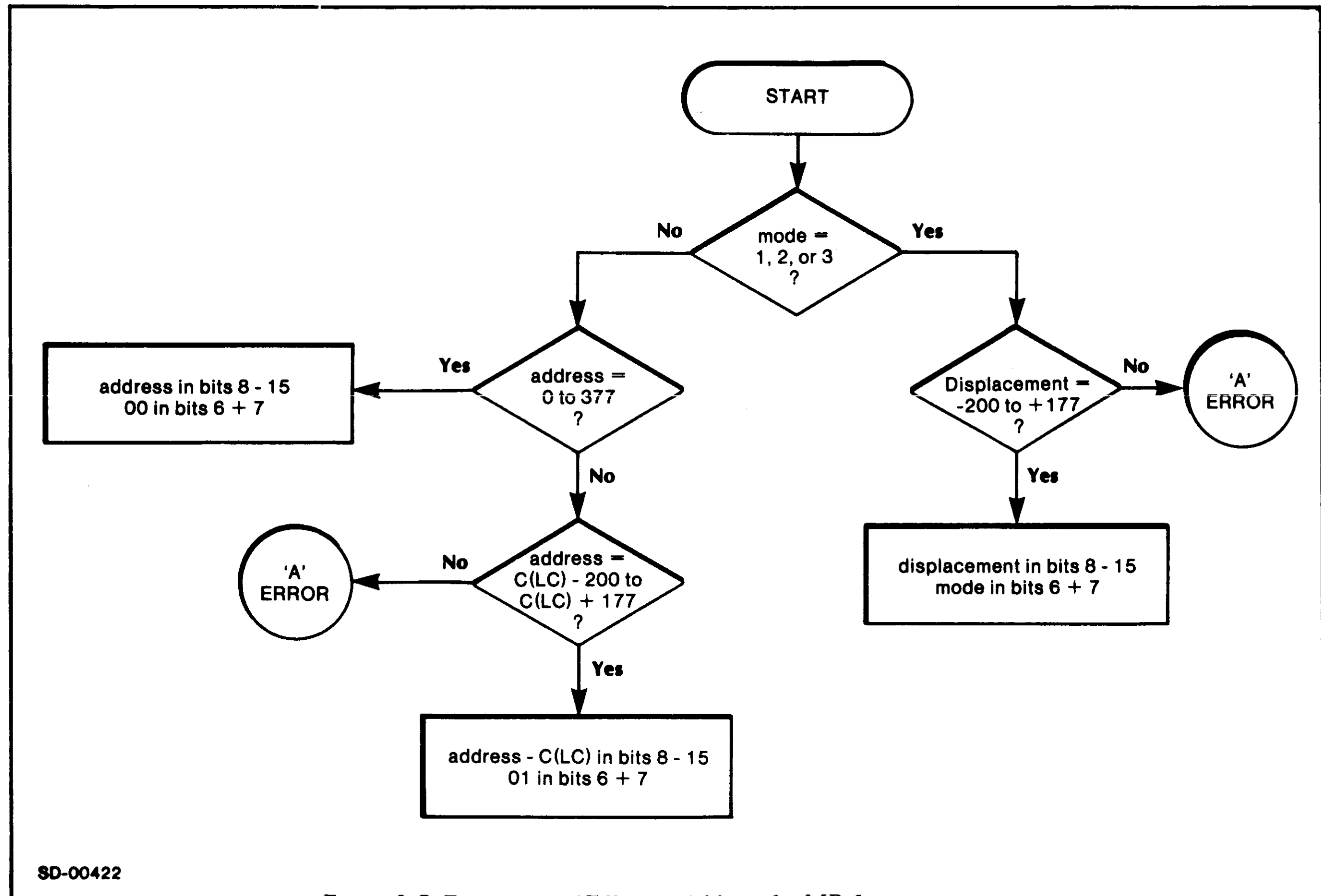


Figure 3-7. Formation of Effective Address for MR Instruction

Examples of MR instructions with their assembled addresses and values follow.

```
010012  SORT1: ISZ STAK
014013         DSZ COUNT
                •
004005         JSR PROC
                •
                •
054014  PROC:STA 3, SAV3
                •
                •
034014       LDA 3, SAV3
001400       JMP 0, 3
000000  STAK:0
000400  COUNT:400
000000  SAV3: 0
```

MR Instructions with Accumulator

A memory reference (MR) instruction with an accumulator field is implied by one of the following instruction mnemonics:

LDA STA

The format of the source program instruction is:

mra-mnemonic□ac□displacement□mode

or

mra-mnemonic□ac□address

where:

mra-mnemonic is a semipermanent symbol: LDA or STA.

accumulator specifies the accumulator to receive or supply the data: 0, 1, 2, or 3.

displacement, mode, and address are the same as MR instructions without an accumulator field.

The atom @ can be specified in the source line address field as a break character. This atom assembles a 1 at bit 5, the indirect addressing bit.

Figure 3-8 shows each assembled MR instruction, its bit pattern, and the effect of the instruction mnemonics.

Examples of MR instructions follow.

```
        040064  STA 0, FB11
        024063  LDA 1, FB10
        046020  STA 1, @20
                     •
                     •
000000  FB10:0
000000  FB11:0

;Indexed MR examples:

        035003  LDA 3, 3, 2
        031002  LDA 2, 2, 2
        021425  LDA 0, TEMP, 3
        000006  TEMP:6
```

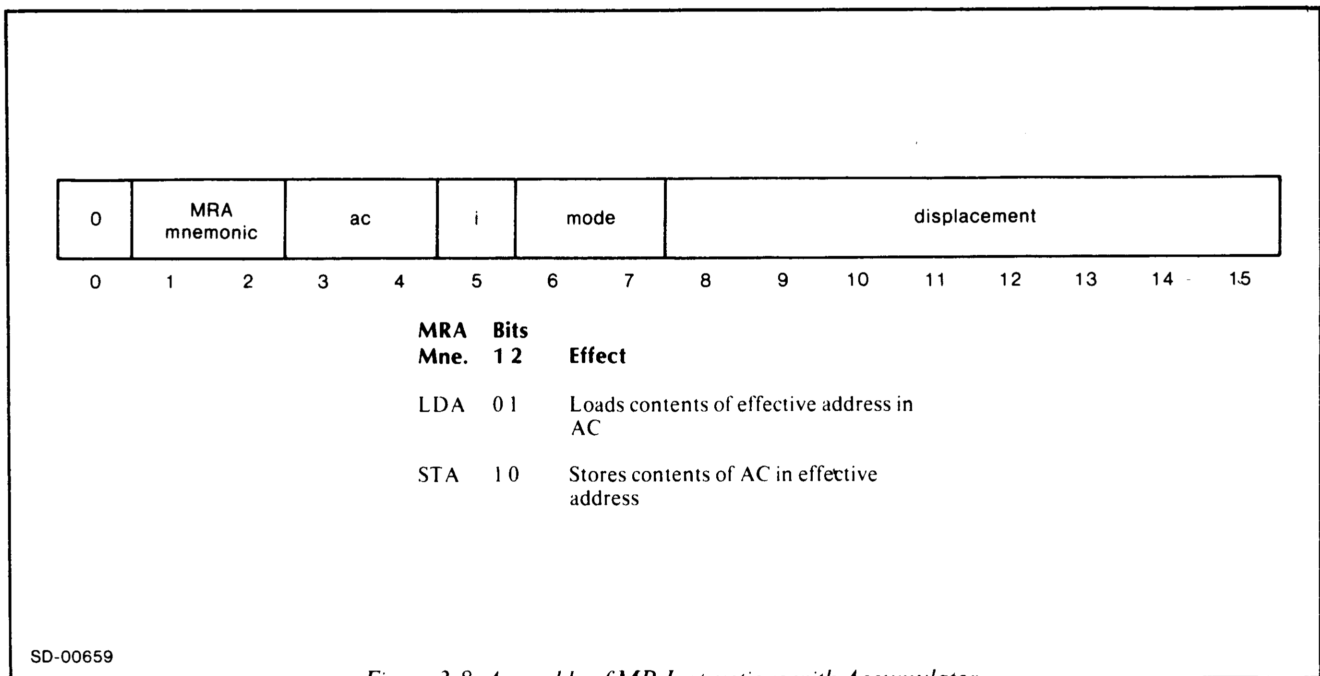


Figure 3-8. Assembly of MR Instructions with Accumulator

ECLIPSE Instructions

MAC recognizes certain instructions which will execute only on an ECLIPSE computer. The remainder of this chapter describes these instructions. If your programs will run on NOVAs only, skip to Chapter 4. The ECLIPSE-only insructions are:

Extended Memory Instructions (defined by pseudo-ops .DEMR and .DMRA)

EDSZ	EISZ	EJMP	EJSR
ELDA	ESTA	ELEF	

Commercial Instructions (defined by pseudo-op .DCMR)

ELDB ESTB

Floating-Point Instructions (defined by pseudo-op .DFLM and .DFLS)

You can code, assemble, and load these instructions on any Data General computer but the resultant save file will execute only on an ECLIPSE.

Extended MR Instructions

Extended memory instructions can reference any memory location in a full 32K address space. The extended memory instructions not requiring an accumulator are:

EDSZ EISZ EJMP EJSR

Those requiring an accumulator are:

ELDA ESTA ELEF

There are two formats for extended memory reference instructions: one specifies an index, the second specifies no index. The first format is:

instruction *[indirect]* □displacement□index for EDSZ, etc.,

or

instruction□ac□ *[indirect]* □displacement□index for ELDA, etc.

The second format is:

instruction *[indirect]* address for EDSZ, etc.,

or

instruction ac *[indirect]* address for ELDA, etc.

where:

instruction — is any extended memory reference instruction.

indirect — (@) represents an indirect address in the second word (bit zero) of this instruction.

ac — specifies accumulator 0, 1, 2 or 3. It must be given for ELDA, ESTA, or ELEF.

displacement — represents a displacement in the following ranges (r).

Index mode 0:	$0 < r < 100{,}000_8$
Index modes 1, 2,3:	$-40{,}000_8 < r < 40{,}000_8$

index — represents an index field whose value must be 0 0 (absolute addressing), 1 (PC relative), 2 (contents of AC2) or 3 (contents of AC3).

address — may specify any word in the full 32K address space.

Extended memory reference instructions require two words of memory. The first word specifies the instruction and index. The second specifies the displacement and whether the instruction is indirect.

The first format is used with the specified mode of indexing (0, 1, 2, or 3). When the second format is used, the assembler attempts to form the correct index mode and address representation. The assembled index mode will always be either 0 or 1. Rules for determining the assembled index mode in the second format are as follows:

1. Mode is 1 if the current program counter and addressed location have the same address type. The addresses must be both NREL, both ZREL, or both absolute.
2. Mode is 0 if the current program counter and the addressed location do not have matching address types.

3. Mode is 1 if the addressed location is external to the assembly. In this case, the assembler must make an assumption about the ultimate relocation of the destination symbol. The assembler assumes that the resolved address of an EDSZ, EISZ, EJMP, EJSR, ELDA, ESTA or ELEF will be determined by word relocation, not byte relocation. Only if the object of an ELEF is byte-relocatable could the assembler's assumption logically be false. In this case, force absolute addressing by using the first format with an index value of zero.

Commercial MR Instructions

There are two commercial memory reference instructions: extended load byte (ELDB) and extended store byte (ESTB). These instructions can reference any eight-bit byte in a full 32K address space. These instructions have the following formats:

comm'l-mnemonic□ac□displacement□index

or

comm'l-mnemonic□ac□address

where:	comm(ercia)l-mnemonic is either ELDB or ESTB;
ac	specifies accumulator 0, 1, 2 or 3 using any legal expression.
displacement	represents a displacement that must range from -37,777$_8$ through +37,777$_8$, or an from (0 through 7777$_8$ if *index* equals 0).
index	represents an index field whose value must be 0 (absolute addressing), 1 (PC relative), 2 (contents of AC2), or 3 (contents of AC3).
address	may specify any word in a 32K address space.

Each commercial extended MR instruction requires two words of memory. The first word specifies the insruction, accumulator field, and optional index; the second specifies the displacement.

Each of these insructions forms a bytepointer by either taking the value specified by index (PC, AC2 or AC3), multiplying it by 2, and adding the low-order 16 bits of the result to the value specified by displacement or, if absolute addressing is used, the bytepointer is simply the displacement. The byte addressed by ths bytepointer is placed in or stored from bits 8-15 of the specified ac.

The resolved address of an ELDB or ESTB instruction is assumed to be byte-relocatable. An example using a commercial extended MR instruction is:

```
        ELDB 1, ASC.A   ;Load ASCII
                        ;A into AC1,
                        ;no indexing.
                    .
                    .
TX:.TXT "AB"
   ASC.A = TX*2
   ASC.B = TX*2+1
```

Floating-Point Instructions

There are two general types of floating-point instructions:

Those which use a displacement, and those which do not, as follows.

instruction□fpac *[indirect]* displacement *[index]*

or

instruction□ac fpac

where:	
instruction	is any floating-point instruction; examples of those which use displacements (first format) are FLDD, FLDS, and FSTS. Examples of those without displacements (second format) are FLAS and FFAS.
fpac	is one of the four *floating-point* accumulators: 0, 1, 2, or 3.
indirect	(@) specifies an indirect address in the second word (bit 0) of the instruction.
displacement	is a value used to calculate the effective address (see below).

index is a value in bits 0 and 1, which the assembler used to calculate the effective address, as follows:

Index Value	Effective Address Determination
00	*displacement* is treated as an unsigned unsigned integer, which is the address of a word in memory.
01	*displacement* is treated as a signed, two's complement number, which the address of the word containing the displacement bits.
[index] 10,11	Index 2 or index 3 is used as an index register. The *displacement* is treated as a two's complement number which is added to the contents of the appropriate register to provide a memory address. The value of the sum cannot exceed 077777_8.

ac is one of the normal accumulators: 0, 1, 2 or 3.

Instructions of the first type (FLDS) require two words of memory while instructions of the second form (FLAS) require only one.

End of Chapter

Chapter 4
Pseudo-ops and Value Symbols

This chapter lists the Macroassembler pseudo-ops and value symbols, both by category (below) and alphabetically (on the yellow pages of this chapter).

The Macroassembler assembles a source program, and RLDR processes it into the assembler. All pseudo-ops are permanent; there is no pseudo-op or facility which can change or delete them. Symbols defined by pseudo-ops are semipermanent. That is, they can be deleted (.XPNGed), but ordinarily persist for the duration of an assembly. *Value Symbols* contain values during an assembly process. The value symbol *.ARGCT*, for example, has the number of arguments given in the most recent macro call.

Symbol Table Pseudo-ops

Symbol table pseudo-ops comprise the largest category of pseudo-ops; they define machine instructions, define user symbols, and expunge macro and symbol definitions. Certain symbol table pseudo-ops define instructions which will execute only on an ECLIPSE computer, although you can code, assemble, and load them on any Data General computer. These pseudo-ops are:

.DCMR	.DIAC
.DEMR	.DICD
.DERA	.DIMM
.DEUR	.DXOP
.DFLM	
.DFLS	

The symbol table pseudo-ops listed alphabetically are:

Pseudo-op	Instruction
.DALC	Define an ALC instruction or expression.
.DCMR	Define a commercial memory reference instruction or expression.
.DEMR	Define an extended memory reference instruction or expression, without accumulator.
.DERA	Define an extended memory reference instruction that requires an accumulator.
.DEUR	Define an extended user insruction or expression.
.DFLM	Define a floating-load or -store instruction or expression that requires an accumulator.
.DFLS	Define a floating load or store instruction or expression that requires no accumulator.
.DIAC	Define an instruction requiring an accumulator.
.DICD	Define a instruction requiring an accumulator and a count.

Pseudo-op	Instruction
.DIMM	Define an immediate-reference instruction requiring an accumulator.
.DIO	Define an I/O instruction that does not use an accumulator.
.DIOA	Define an I/O instruction having two required fields.
.DISD	Define an instruction with source and destination accumulators, no skip.
.DISS	Define an instruction with source and destination accumulators allowing skip.
.DMR	Define a memory reference instruction with displacement and index.
.DMRA	Define a memory reference insruction with two or three fields.
.DUSR	Define a user symbol without implied formatting.
.DXOP	Define an instruction with source, destination and operation fields.
.XPNG	Remove all semipermanent symbol definitions and macros.

All symbol table pseudo-ops (except *.DUSR* and *.XPNG*) name assembler instructions (such as LDA) which are described in the appropriate programmer's reference manual for your computer. Parameter files supplied by DGC employ these pseudo-ops to define assembler instructions; these definitions reside in disk file MAC.PS. MAC.PS is usually required for assembly of source programs, and it is further described in Chapter 6.

Symbol Table Pseudo-op Format

Symbol table pseudo-ops have the form:

pseudo-op□user-symbol = {instruction | expression}

where:

pseudo-op is a symbol table pseudo-op;

user-symbol is a symbol chosen by the programmer;

instruction and expression are as defined in Chapter 3.

In symbol table pseudo-ops, a user symbol is semipermanent; its value is the value of the instruction or expression following the equals sign.

Excepting .DUSR and .XPNG, each symbol table pseudo-op defines a certain type of instruction. After definition, the semipermanent symbol must be used with appropriate expressions. For example, the pseudo-op .DALC defines a symbol that is an implied arithmetic and logic mnemonic. Following the symbol are expressions entered into bit fields that represent the source and destination accumulators and the optional skip field in an ALC instruction. The format for .DALC definition of a symbol, and the format of the symbol as it would later be used are:

.DALC user-symbol = {instruction | expression}

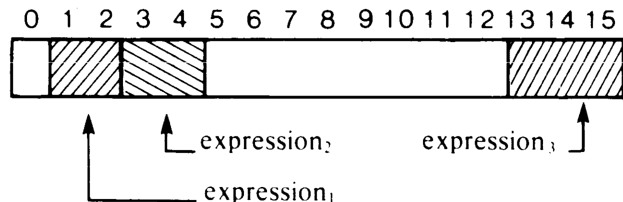
.

.

.

user-symbol $expr_1$ $expr_2$ *[$expr_3$]*

where: expr(ession)$_1$, expr(ession)$_2$, expr(ession)$_3$ are stored:



For example:

```
103120 .DALC   MULT4 = 103120
               ;MULT4 is defined as:
               ;1-000-011-001-010-000.

127120  MULT4, 1, 1  ;MULT4 must be used
        ;with 2 expressions that evaluate
        ;within the limits of an ALC
        ;instruction-- 2 bits for each AC.
        ;The assembled instruction is:
        ;1-010-111-001-010-000
```

If the field cannot contain the value of the added expression, an overflow error will occur. The field will be unaltered.

```
     123120   .DALC MULT4 = 123120
O    107120   MULT4   4, 1    ;Note overflow
                              ;error.
```

If the field is not zero, the expression to be added must evaluate to zero. otherwise, an overflow error will occur.

```
        123120   .DALC MULT4 = 123120
                  ;Bits 1-2 not zeroed.
00002   127120   MULT4 1,1   ;No overflow-

00003   103120   MULT4 0,0   ;Also accept-
                             ;able.
```

If the expressions following a semipermanent symbol do not fit the implied format, a format error (F) will result.

```
        103120  .DALC MULT4=123120
                ;MULT4 requires 2, option-
                ;ally 3, expressions.
FF              MULT4, 1 ;Format errors.
F00004 127121   MULT4, 1, 1, 1, 1
```

In summary, a symbol defined as semipermanent by a symbol table pseudo-op must meet the following conditions:

- As many expressions must follow the semipermanent symbol as are required by the implied format. Some formats permit optional expressions as well as required expressions. If the number of expressions following the semipermanent symbol does not meet the requirements of the implied format, a format error (F) will result.
- If an expression does not meet the requirements of the field, i.e., if

  expression > (2(supfield-width)-1)

  the field is unaltered and an overflow error (O) results.
- If the field in which the expression is to be stored does not equal zero, the expression must equal zero (0). Otherwise, the field is unaltered, and an overflow error (O) results.

A given *user-symbol* defined in one symbol table pseudo-op may be redefined in another symbol table pseudo-op. The last definition will be the one assigned to *user-symbol.* A redefinition of a permanent symbol will result in a multiple definition (M) error if the global /M switch was used.

Location Counter Pseudo-ops

Location counter pseudo-ops are used to reserve a block of memory locations and to specify a memory location or class of relocatable locations.

Pseudo-op	Instruction
.BLK	Reserve a block of storage locations.
.LOC	Set the current location counter.
.NREL	Specify NREL code relocation.
.ZREL	Specify page zero relocation.

The symbol "." (period) has the value and relocation property of the current location counter.

Intermodule Communication Pseudo-ops

Intermodule communication pseudo-ops allow symbols in one module to be referenced by modules after the modules are bound together. These pseudo-ops work by defining entries, external references and named and unlabeled common areas for communications:

Pseudo-op	Instruction
.COMM	Reserve a named common area.
.CSIZ	Reserve an unlabeled common area.
.ENT	Define an entry.
.ENTO	Define an overlay name.
.EXTD	Define an external displacement reference.
.EXTN	Define an external normal reference.
.EXTU	Treat undefined symbols as external displacements.
.GADD	Add an expression value to an external symbol.
.GLOC	Reserve an absolute data block.
.GREF	Add an expression value to an external symbol without affecting the sign bit.

When a source file defines a symbol which other source files will use, the defining file must declare this symbol with .ENT or .COMM at its beginning. The other source file(s) can then reference the defined symbol with .EXTN or .EXTD pseudo-ops. Symbols named by .ENT can be entry points, which can be called or jumped to, or they can be data available to their modules for external reference.

.ENTO identifies the number and node of an overlay so that it can be referenced by name.

Repetition and Conditional Pseudo-ops

These pseudo-ops perform two different functions. Source lines following the *.DO* pseudo-op are assembled a specified number of times. Source lines following conditional pseudo-ops will be assembled based on the evaluation of an expression provided to the pseudo-op. The following pseudo-ops are provided:

Pseudo-op	Instruction
.DO	Assemble the following source lines a specified number of times.
.ENDC	Define the end of a series of conditional-assembly source lines.
.GOTO	Suppress assembly of source lines until the specified symbol is encountered.
.IFE	Assemble only if expression equals zero.
.IFG	Assemble only if expression exceeds zero.
.IFL	Assemble only if expression is less than zero.
.IFN	Assemble only if expression is nonzero.

The .ENDC pseudo-op delimits source lines which are to be assembled conditionally.

Macro Definition Pseudo-op and Values

The .MACRO pseudo-op defines the start of a macro definition, and names that macro. Macros are described at length in Chapter 5.

Two symbol values are provided for use in macros: .ARGCT and .MCALL. .ARGCT has as a value the number of arguments specified by the most recent macro call. .MCALL indicates macro usage; its value is 1 if the macro in which it appears has been called previously in ths assembly pass. Its value is zero if this is the first call in the pass. Outside a macro call, the value of .MCALL is -1.

Stack Pseudo-ops and Values

The assembler maintains a last-in first-out stack onto which you may save the value and relocation property of any valid assembler expression. Expressions are pushed onto this stack by the .PUSH pseudo-op; they are removed from this stack by the .POP pseudo-op.

The .TOP pseudo-op returns the value and relocation property of the expression most recently pushed onto the stack.

Text String Pseudo-ops and Values

The assembler permits you to insert ASCII text strings within source programs in a variety of ways. Characters can have their most significant bit set to zero or one unconditionally, or to even or odd parity. Even strings can be terminated with two zero bytes or no zero byte; strings with odd numbers of bytes are always terminated with a single zero byte. The following string management pseudo-ops are available:

Pseudo-op	Instruction
.TXT	Set the leftmost bit to zero unconditionally.
.TXTE	Set the leftmost bit for even byte parity.
.TXTF	Set the leftmost bit to one unconditionally.
.TXTM	Set bytepacking to left/right or right/left.
.TXTN	Terminate an even bytestring with no zero bytes or two zero bytes.
.TXTO	Set the leftmost bit for odd byte parity.

Enclosing the .TXTN pseudo-op with parentheses, when it has no argument, returns the value of the most recent .TXTN expression. Likewise, .TXTM can return the most recent .TXTM value.

Listing Pseudo-ops and Values

The assembler provides several pseudo-ops to suppress portions of output listings. By default, all source lines are listed; this includes conditional areas, macro expansions, and lines lacking a location field. The following pseudo-ops can affect the listing of source lines:

Pseudo-op	Instruction
.EJEC	Begin a new listing page.
.NOCON	Omit or restore listing of conditional source lines.
.NOLOC	Inhibit the listing of source lines lacking location fields.
.NOMAC	Inhibit the lising of macro expansions.

You can override any of the suppression pseudo-ops by including the global /O switch in the MAC command line; you can also suppress the listing of an assembly by omitting the /L switch in the CLI command line.

Miscellaneous Pseudo-ops

These pseudo-ops perform miscellaneous assembly functions.

The .REV pseudo-op can be used with an argument to assign a numeric major and minor revision level to a save file.

.RDX specifies the number base to be used in evaluating all numeric expressions input to the assembler. .RDXO defines the radix to be used for numeric conversion output.

.COMM TASK can assign a number of tasks and I/O channels for the save file to use. At load time, you can override the values specified with RLDR local switches.

.TITL assigns a name to an object module. *.RB* names a relocatable binary file, and can be overridden by global /B. *.LMIT* causes the partial loading of an assembled binary file; loading of the remainder of the binary file ceases when the entry which you specified as an argument to .LMIT is detected.

The .PASS pseudo-op returns a value corresponding to the current pass of the assembler, either 0 (pass 1) or 1 (pass 2). An explicit end-of-file can be established for any source module by means of the *.EOF* pseudo-op. If this pseudo-op is missing from a source module, the system supplies an end-of-file for this module automatically. The *.END* pseudo-op specifies the end-of-file for the last module in the assembly. If you omit .END, the system automatically supplies an end-of-file. .END can also supply a starting address for the program file. At least one module loaded into each program file must specify a starting address with a .END statement.

(.)

Current Location Counter

Value:

The symbol . (period) has the value and relocation property of the current location counter.

Example(s):

```
                .NREL
00000'000003    3
      000003'   .LOC .+2
00003'020010    LDA 0, 10
```

.ARGCT

Number of Most Recent Macro Arguments

Syntax:

.ARGCT

Value:

.ARGCT has as a value the number of actual arguments given in the most recent macro call. If the symbol is used outside a macro, its value is -1.

Example(s):

```
              .NREL
              .MACRO X
              ^1+^2
              .ARGCT
              %
              X 4, 5   ;Macro call has
00000'000011 4+5      ;two args.
00001'000002 .ARGCT
```

.BLK

Reserve a Block of Storage

Syntax:

.BLK expression

Purpose:

This pseudo-op reserves a block of storage. expression is the number of words to be reserved. The current location counter is incremented by expression.

Example(s):

```
                          .NREL
00000'044403              STA 1, .F+1
00001'040403              STA 0, .F+2

00002'000004 .F:          .BLK 4
00006'000000 .F1:         0
00007'000000 .F2:         0
```

.COMM

Reserve a Named Common Area

Syntax:

.COMM user-symbol expression

Purpose:

This pseudo-op reserves a named common area for interprogram communication having the name user-symbol and the size in words as given by expression. This area will be reserved by the first routine loaded that declared the named user-symbol. The area is reserved at NMAX, immediately above all NREL code. Other programs bound with the defining program can share a .COMMon area, provided that they declare the same .COMMon size.

Since user-symbol is an entry in the program, it cannot be redefined elsewhere in the program. You can reference the user-symbol from other programs loaded together by using .EXTN, .EXTD, .GLOC, or .GADD pseudo-ops.

.COMM TASK defines the number of tasks and I/O channels which the entire save file will need to execute. You can also specify tasks and channels with local switches in the RLDR command line - this switch information overrides any .COMM TASK data.

.COMM TASK has the following format:

.COMM TASK,k*400+c

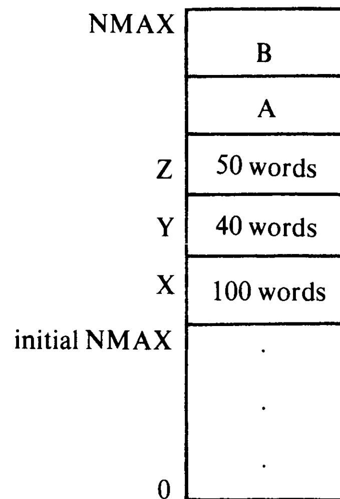
where k is the number of tasks, and c is the number of I/O channels, in octal.

Example (.COMM):

```
        .TITL A
000100  .COMM X, 100 ;100 words for X.
000040  .COMM Y, 40  ;40  for Y.
000050  .COMM Z, 50  ;50 for Z.
          .
          .
        .END

        .TITL B
000100  .COMM X, 100
          .
        .END
```

After loading, A and B would look like this when the program executes:



.CSIZ
Specify an Unlabeled Common Area

Syntax:

.CSIZ expression

Purpose:

This pseudo-op specifies the size in words of an unlabeled common area for interprogram communication.

The assembler evaluates expression and passes this value to the RLDR. More than one .CSIZ pseudo-op may appear in a program; the RLDR uses the largest value specified by all .CSIZ blocks.

Example(s):

```
        .TITL A
000020  .CSIZ 20        ;A allots 20 words.
           .
           .
        .END

        .TITL X
000050  .CSIZ 50        ;X allots 50 words.
           .
        .END
```

After assembly, you issue the command:

```
RLDR A X)
    A
    X
    NMAX     001037
    ZMAX     000050
    CSZE     000050
```

RLDR selects the largest .CSIZ value for USTCS(50).

.DALC

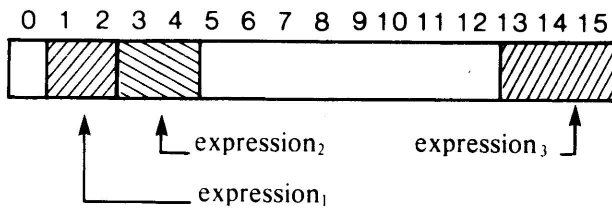
Define ALC Instruction

Syntax:

.DALC user-symbol = { instruction | expression }

Purpose:

The .DALC pseudo-op defines user-symbol as a semipermanent symbol having the value of instruction or expression. This symbol implies the format of an ALC instruction. At least two fields, and optionally three, are required. These fields are assembled as shown below.



The format for user-symbol is:

user-symbol expression₁ expr₂ *[expr₃]*

Example(s):

```
        103400  .DALC  ADD = 103000
  00021'103000  ADD 0, 0
  00022'103002  ADD  0, 0, SZC
  00023'133001  ADD 1, 2, SKP
FF              ADD 1
```

Notes:

You can insert the atom # before the arguments in the source line to specify no loading of the destination accumulator. If you use # in a source line without a skip field, the assembler will return a "Q" error (because such lines assemble as special ECLIPSE instructions).

A given user-symbol defined in one .DALC pseudo-op may be redefined in another .DALC pseudo-op The last definition will be the one assigned to user-symbol.

If you use this pseudo-op to define a three-character symbol which includes, or is followed immediately by, the letters Z, O, C, L, R, or S (or any combination of them), the Macroassembler will set bits 8-9 to 10-11 as follows:

Mnemonic	Bits 8 - 9	Bits 10 - 11
L	01	
R	10	
S	11	
Z		01
O		10
C		11

.DCMR
Define Commercial Memory Reference Instruction

Syntax:

.DCMR user-symbol = { instruction | expression }

Purpose:

This pseudo-op defines user-symbol as a semipermanent extended commercial memory reference symbol having the value of instruction or expression. This symbol implies the format of an instruction that requires an accumulator and displacement, and permits an optional index. The fields are assembled as shown below.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
			accumulator	accumulator		index	index								

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
displacement															

The format for using this semipermanent symbol is:

user-symbol displacement *[index]*

Example(s):

```
        000001 .TXTM 1
                .
        102170 .DCMR  ELDB=102170
               .NREL
000000'112170  ELDB 2, ASC.A   ;Get "A"
       001400"          ;in AC2.

000002'112170  ELDB 2, ASC.B    ;Get "B"
       001401"          ;in AC2.
               .
               .
000600'041101  ALPHA: .TXT "AB"
       000000
       001400" ASC.A=ALPHA*2 ,
       001401" ASC.B = ALPHA*2+1
```

.DEMR

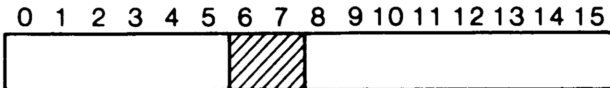
Define Extended Memory Reference Instruction

Syntax:

.DEMR user-symbol = { instruction | expression }

Purpose:

This pseudo-op defines user-symbol as a semipermanent extended memory reference symbol and gives it the value of instruction or expression. This symbol implies the format of an instruction that does not require an accumulator. One field is required; an index is optional. They are assembled as shown below.



The format for using the semipermanent symbol is:

user-symbol displacement *[index]*

The displacement and *index* fields are set according to the format used and the set of addressing rules described in Chapter 3.

Example(s):

```
                 .EXTN ADDR1, ADDR2, ADDR3
      102070     .DEMR  EJMP = 102070
                 .NREL

00000'102070     EJMP ADDR1
      077777
00002'102470     EJMP  .+3
      000002
000004'103470    EJMP  2, .+3   ;There is no AC
      000002
                                ;00004+3.

00006'103470     EJMP @TABLE, 3  ;Go to either
      100013'
                 .               ;ADDR1, ADDR3, or
                 .               ;ADDR3, based on
                 .               ;value in AC3.
00617'077777 TABLE:     ADDR1
00620'077777            ADDR2
00621'077777            ADDR3
```

Note:

You can use the @ atom in the address field as a break character, to specify indirect addressing.

.DERA

Define Extended Memory Reference Instruction with Accumulator

Syntax:

.DERA user-symbol = { instruction | expression }

Purpose:

This pseudo-op defines user-symbol as a semipermanent extended memory reference symbol having the value of instruction or expression. This symbol implies format of an instruction that requires an accumulator. Two fields are required and *index* is optional. They are assembled as shown.

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

accumulator (bits 3-4), index (bits 6-7)

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

@ (bit 0), displacement (bits 1-15)

The format for using this semipermanent symbol is:

user-symbol ac displacement *[index]*

The displacement and *index* fields are set according to the format used and the set of addressing rules described in Chapter 3.

Note:

You can specify @ in the address field to specify indirect addressing.

Example(s):

```
                                .TITL TESTS9
           122070       .DERA   ELDA = 122070
                        .NREL
    00034'133470        ELDA    2, .+5, 3
           000041'
    00036'122470        ELDA    0, .+3
           000002
    00040'132470        ELDA    2, .+4
           000003
FFO                     ELDA    .+3 ;AC needed.
```

.DEUR

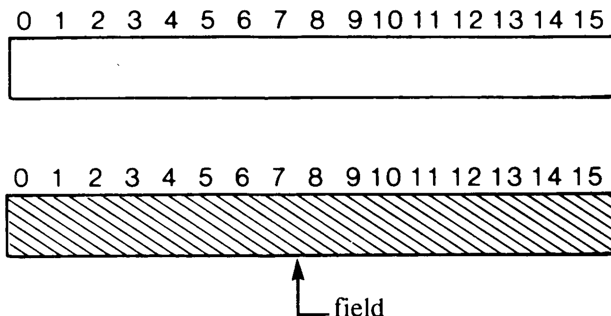
Define Extended User Instruction

Syntax:

.DEUR user-symbol = { instruction | expression }

Purpose:

This pseudo-op defines user-symbol as a semipermanent extended symbol having the value of instruction or expression. The expression can be either an expression, an external normal, or a external displacement. This symbol implies the format of a instruction that does not require an accumulator. One field is required and is assembled as shown.



The format for using this semipermanent symbol is:

user-symbol expression

Example(s):

```
       163710    .DEUR   SAVE = 163710
       061777    .DEUR   VCT = 061777
                 .NREL
                 .EXTN  SYMB
00042'163710     SAVE 4
       000004
00044'061777            VCT  SYMB
       077777
```

.DFLM

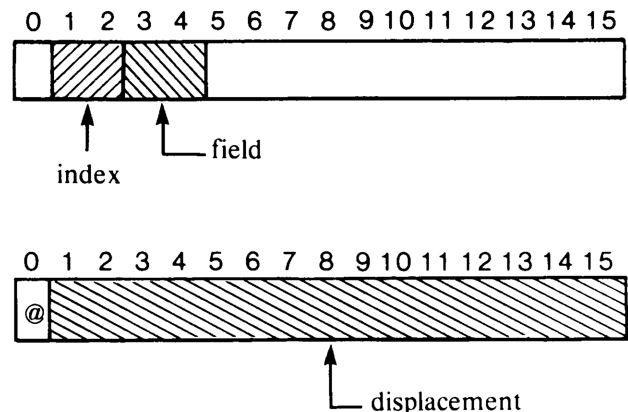
Define Floating Load or Store Instruction

Syntax:

.DFLM user-symbol = { instruction | expression }

Purpose:

This pseudo-op defines user-symbol as a semipermanent floating-point load or store memory reference symbol having the value of instruction or expression. This symbol implies the format of an instruction that requires an accumulator. Two fields are required and one is optional. They are assembled as shown.



The format for using ths semipermanent symbol is:

user-symbol fpac displacement *[index]*

Note that you can specify the atom @ in the address field as a break character to specify indirect addressing.

Example(s):

```
           102050    .DFLM FLDS = 102050
                     .NREL
   00046'122050      FLDS 0, .+2
           000001
   00050'146050      FLDS 1, .+3, 2
           000053'
FFO        FLDS   .+3    ;Format error-
                         ;AC needed.
```

.DFLS

Define Floating Load or Store Instruction

Syntax:

.DFLS user-symbol = { instruction / expression }

Purpose:

This pseudo-op defines user-symbol as a semipermanent floating load or store memory reference symbol having the value of instruction or expression. This symbol implies the format of an insruction that does not require an accumulator. One field is required; an index is optional. These are assembled as shown.

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

index

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

@

displacement

The format for using this symbol is:

user-symbol displacement *[index]*

Note that you can specify the atom @ in the address field as a break character to specify indirect addressing.

Example(s):

```
        123350   .DFLS FLST = 123350
                 .EXTD ADDR1

  00052'123350   FLST  ADDR1
        000005$
FF               FLST  ;Format error-
                ;displacement needed.
```

.DIAC

Define an Instruction Requiring an Accumulator

Syntax:

.DIAC user-symbol = { instruction / expression }

Purpose:

This pseudo-op defines user-symbol and gives it the value of instruction or expression. This symbol implies the format of an instruction requiring an accumulator. One field is required, and it is assembled as shown below.

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

expression

The format for using this symbol is:

user-symbol expression

Example(s):

```
       123370   .DIAC  XCT = 123370
                .NREL
       123370   XCT 1      ;ok.
 F00054'123370  XCT 0, 0 ;Illegal
FF              XCT        ;number of
                        ;expressions.
```

.DICD

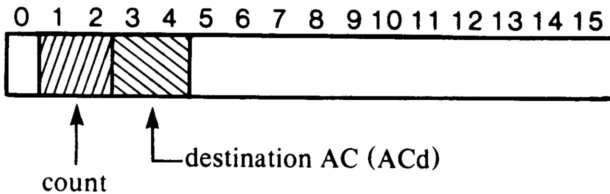
Define an Instruction Requiring an Accumulator and Count

Syntax:

.DICD user-symbol = { instruction / expression }

Purpose:

This pseudo-op defines user-symbol as a semipermanent symbol having count and destination fields. The symbol has the value of instruction or expression. This symbol implies the format of a instruction that requires an accumulator and a count from 1 to 4. Two fields are required and are assembled as shown.



The format for using this semipermanent symbol is:

user-symbol□count□destination-ac

Example(s):

```
        100010    .DICD ADI = 100010
                  .NREL
   00055'104010   ADI 1, 1
   00056'110010   ADI 1, 2
   00057'160010   ADI 4, 0
  000060'104010   ADI 5, 1 ;Overflow
        ;error- count field too large.
FF                ADI 1  ;Format
        ;error- 2 fields needed.
```

Note:

counts are entered in the instruction as "(specified value-1)". Thus the range of permitted values is 1 - 4.

.DIMM

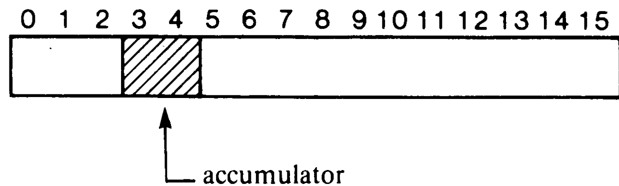
Define an Instruction Requiring an Accumulator and an Immediate Word

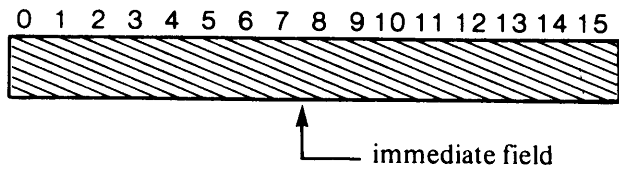
Syntax:

.DIMM user-symbol = { instruction / expression }

Purpose:

This pseudo-op defines user-symbol as a semipermanent immediate-reference symbol having the value of instruction or expression. This symbol implies the format of an instruction that requires an accumulator and a 16-bit immediate word. Two fields are required and are assembled as shown.





The format for using this semipermanent symbol is:

user-symbol immediate value destination-ac

Example(s):

```
          163770    .DIMM ADDI = 163770
                    .NREL
   00061'173770     ADDI 1002, 2
          001002
FF                  ADDI 0  ;Format error-
                        ;2 fields needed.
```

.DIO

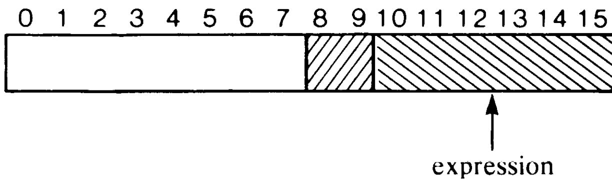
Define an I/O Instruction Without an Accumulator

Syntax:

.DIO user-symbol = { instruction / expression }

Purpose:

This pseudo-op defines user-symbol as a semipermanent symbol having the value of instruction or expression. This symbol implies the format of an I/O instruction without an AC field. One field is required; it is assembled as shown below.



user-symbol is used in this format:

user-symbol expression

Example(s):

```
        063400  .DIO SKION = 063400
        063402  SKION 2    ;ok.
FF              SKION  ;1 field needed.
 F      063402  SKION 2, 3 ;Too many
                           ;fields.
```

Note:

If you use this pseudo-op to define a three-character symbol which is followed immediately by the letters S, C, or P, the Macroassembler will assume that letter to be an optional expression and will set bits 8-9 of the instruction word as follows:

Mnemonic	Bits
S	01
C	10
P	11

.DIOA

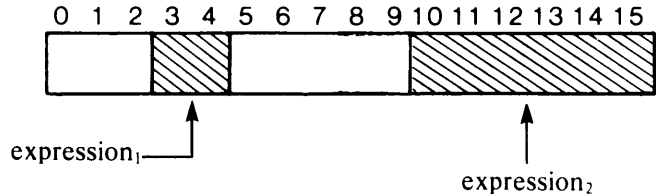
Define an I/O Instruction With Two Required Fields

Syntax:

.DIOA user-symbol = { instruction / expression }

Purpose:

This pseudo-op defines user-symbol as a semipermanent symbol having the value of instruction or expression. This symbol implies the format of an I/O instruction with two required fields. The fields are assembled as shown below.



user-symbol is used in this format:

user-symbol $expression_1$ $expr_2$

Example(s):

```
060400    .DIOA DIA = 060400
          .NREL
070410     DIA 2, TTI
070610    DIAC 2, TTI
```

Note:

If you use this pseudo-op to define a three-character symbol which is followed immediately by the letters S, C, or P, the Macroassembler will assume that letter to be an optional expression and will set bits 8-9 of the instruction word as follows:

Mnemonic	Bits 8 - 9
S	01
C	10
P	11

.DISD

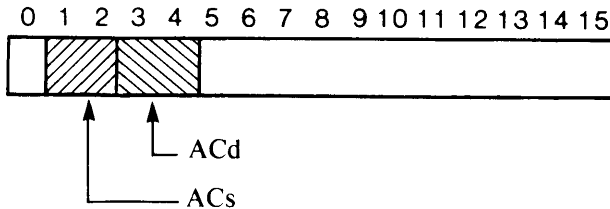
Define an Instruction With Source and Destination Accumulators

Syntax:

.DISD user-symbol = { instruction / expression }

Purpose:

This pseudo-op defines user-symbol as a semipermanent reference symbol with source and destination fields; it does not allow the no-load flag or skip conditions. The instruction cannot cause a skip. The symbol has the value of instruction or expression. This symbol implies the format of an instruction that requires a source and a destination accumulator. Two fields are required and are assembled as shown.



The format for using ths semipermanent symbol is:

user-symbol source-ac destination-ac

Example(s):

```
000001   .TXTM 1
           .
102710   .DISD  LDB = 102710
         .NREL
030402   LDA 2, .PTR
146710   LDB 2, 1    ;AC1 loads
                     ;byte "A".
           .
           .
000162" .PTR:.+1*2
041101   .TXT "ABCDE"
042103
000105
```

.DISS

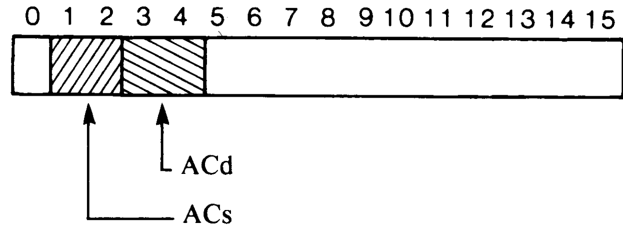
Define an Instruction With Source and Destination Accumulators Allowing Skip

Syntax:

.DISS user-symbol = { instruction / expression }

Purpose:

This pseudo-op defines user-symbol as a semipermanent reference symbol with source and destination fields. The no-load flag cannot be used and no skip condition can be specified, but the instruction may cause a skip to occur. The .DISS symbols differ from the .DISD symbols only in that .DISS symbols may cause a skip and .DISD symbols never cause a skip. The symbol has the value of instruction or expression. This symbol implies the format of an instruction that requires a source and a destination accumulator. Two fields are required and are assembled as shown.



The format for using this semipermanent symbol is:

user-symbol source-ac desination-ac

Example(s):

```
      101010    .DISS  SGT = 101010
                .NREL
      131010    SGT 1, 2
FF              SGT  1  ;Format error-
                        ;2 fields needed.
```

.DMR

Define a Memory Reference Instruction With Displacement and Index

Syntax:

.DMR user-symbol = { instruction | expression }

Purpose:

This pseudo-op defines user-symbol as a semipermanent symbol having the value of the instruction or expression. This symbol implies the format of an MR instruction with either one or two required fields (an address or a displacement and index). The fields are assembled as shown below.

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

(bits 6–7: index; bits 8–15: displacement)

The format for using this symbol is:

user-symbol displacement *[index]*

The displacement and *index* fields are set according to the format used and the set of addressing rules described in Chapter 3.

Example(s):

```
      000000    .DMR  JMP = 000000
                .NREL
00000'000402    JMP .+2
00001'003001    JMP @1, 2
00002'001401    JMP 1, 3
```

Note:

You can use the atom @ in the address field as a break character to specify indirect addressing.

.DMRA

Define a Memory Reference Instruction with Two or Three Fields

Syntax:

.DMRA user-symbol = { instruction | expression }

Purpose:

This pseudo-op defines user-symbol as a semipermanent symbol having the value of instruction or expression. This symbol implies the format of an MR instruction with either two or three required fields. The first field specifies an accumulator. Where there are two fields, the second is an implied address. Where there are three fields, the second and third fields are displacement and index respectively. The fields are assembled as shown below.

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

(bits 3–4: $expression_1$; bits 6–7: index; bits 8–15: displacement)

user-symbol is used in one of these formats:

user-symbol $expression_1$ $expr_2$

user-symbol $expression_1$ $expr_2$ $expr_3$

The displacement and *index* fields are set according to the format chosen and the set of addressing rules described in Chapter 3.

Example(s):

```
      020000    .DMRA  LDA = 20000
                .NREL
00000'035400    LDA 3, 0, 3
00001'030402    LDA 2, .B+1
00002'000100    .BLK 100
00102'000004 .B: .BLK 4
```

Note:

You can use the atom @ in the address field as a break character to specify indirect addressing.

.DO
Assemble Source Lines Repetitively

Syntax:

.DO expression

Purpose:

.DO assembles all source program lines between itself and its corresponding .ENDC, the number of times given by expression.

Note that nondisk devices, like a card reader or mag tape, cannot execute a .DO loop more than once, because they cannot "back up" to the beginning of the loop. We recommend that all files you input to the assembler be on disk; if any are not, transfer them to disk with the CLI command XFER or LOAD, then assemble the disk file.

Example(s):

```
            ;Source program:

            I=0
            .DO 4   ;Assemble loop
                    ;4 times.
            1BI
            I=I+1
            .ENDC

            ;Listing:

      000000  I=0
      000004  .DO 4 ;Assemble loop
                    ;4 times.
00001'100000  1BI
      000001  I=I+1
              .ENDC
                    ;4 times.
00002'040000  1BI
      000002  I=I+1
              .ENDC
                    ;4 times.
00003'020000  1BI
      000003  I=I+1
              .ENDC
                    ;4 times.
00112'010000  1BI
      000004  I=I+1
              .ENDC
```

Notes:

.DOs may be nested to any depth, the innermost .DO corresponding to the innermost .ENDC, etc.

Chapter 5 describes the handling of .DOs and other conditionals within macros.

.DUSR
Define User Symbol Without Formatting

Syntax:

.DUSR user-symbol = { instruction / expression }

Purpose:

This pseudo-op defines user-symbol as a semipermanent symbol having the value of the instruction or expression following the = sign. Unlike other semipermanent symbols, a symbol defined by .DUSR is merely given a value and has no implied formatting. It may be used anywhere a single-precision operand would be used.

Symbols defined by .DUSR do not become part of the RLDR symbol table.

Example(s):

```
              .ZREL
      000025  .DUSR  B = 25
      000250  .DUSR  C = B*10
              .NREL
00001'177555  B-C
00002'006712  B*C+2
```

.DXOP

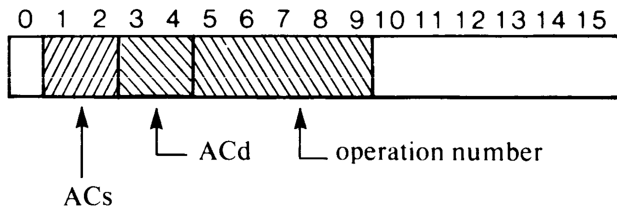
Define an Instruction With Source, Destination, and Operation Fields

Syntax:

.DXOP user-symbol = { instruction / expression }

Purpose:

This pseudo-op defines user-symbol as a reference symbol with source and destination fields and an optional operation number field. The symbol has the value of instruction or expression. This symbol implies the format of an instruction that requires a source and a destination accumulator. Two fields are required; a third is optional. The fields are assembled as shown.



The format for using this semipermanent symbol is:

user-symbol source-ac destination-ac operation no.

Example(s):

```
      100030    .DXOP XOP=100030
                .NREL
00000'130630    XOP 1, 2, 6
00001'154030    XOP 2, 3  ;Operation no.
                          ;is optional.
```

.EJEC

Begin a New Listing Page

Syntax:

.EJEC

Purpose:

To begin a new page in the listing output.

Example(s):

Source code

```
         ;Source program:

           .EXTD SYM
             .
           MOVS 1, 2
           .EJEC
           LDA 0, SYM
             .
           .END

------------------------
;Listing page 1:

         .EXTN SYM
           .
 131300  MOVS 1, 2
         .EJEC
------------------------
------------------------
;Listing page 2

 020001$ LDA 0, SYM
           .
         .END
------------------------
```

.END
End-of-Program Indicator

Syntax:

.END *[expression]*

Purpose:

This pseudo-op terminates a source program, providing an end-of-program indicator for RLDR. The *expression* is an optional argument specifying a starting address for execution. RLDR intializes the first TCB PC to the last address, if any, specified by an assembled binary. Execution of the save file begins at this address. (If RLDR finds no starting address among modules bound, an error message is given.)

Example(s):

```
        .TITL  GTSET
102440  GTSET: SUBO 0, 0 ;Initialize
                         ;for startup.
          .
        .END GTSET
```

.ENDC
Specify the End of Conditional Assembly

Syntax:

.ENDC *[user-symbol]*

Purpose:

If you omit *user-symbol*, .ENDC terminates lines for repetitive assembly (lines following .DO) or lines whose assembly is conditional (lines following .IFE, .IFG, .IFL, or .IFN).

If your syntax is .ENDC *user-symbol*, this pseudo-op both terminates assembly of lines following the last .DO or .IFx and suppresses the assembly of lines following .ENDC until the scan encounters another *user-symbol* enclosed in square brackets.

Example(s):

```
                  .
                  .
   000001  .IFN HDR   ;Assemble only if
   000000  0          ;HDR is nonzero.
           .ENDC LABEL  ;Skip to LABEL
           1            ;if HDR is nonzero.
                  .
                  .
                  .
                  .
000022 [LABEL]   22
                  .
```

.ENT

Define a Program Entry

Syntax:

.ENT user-symbol$_1$ *[user-symbol$_n$]*

Purpose:

This pseudo-op declares each *user-symbol* as a symbol that is defined within this source file and that may be referenced by separately-assembled programs.

A *user-symbol* appearing in a .ENT pseudo-op must be defined as a user symbol within the program. This symbol must be unique from external entries defined in other binaries loaded together to form a save file. If it is not unique, RLDR will issue a message indicating multiply-defined entries.

You can reference *user-symbol(s)* from separately-assembled programs by using one of the following pseudo-ops:

.EXTD
.EXTN
.GADD
.GLOC
.GREF

Example(s):

```
                .TITL A
                .ENT B, .C
                .EXTN C
                .ZREL
00000-077777    .C: C
                .NREL
00000'006000-   B: JSR @ .C
                .END
```

.ENTO

Define an Overlay Entry

Syntax:

.ENTO overlay-name

Purpose:

The .ENTO pseudo-op enables you to assign a symbolic name to a file which will eventually be an overlay. (If you omit .ENTO, you must reference the overlay later by memory node number and overlay number - which is a nuisance.)

You can then access the overlay from the root program by overlay-name, which must be declared as an .EXTN in the root program. Caution: overlay name cannot appear elsewhere in the file whch declares it as an overlay name, since its value is assigned at load time.

Example(s):

```
                .TITL METER   ;This is the
                .ENTO METER   ;overlay.
                .ENT PROC1
)0001           .TXTM 1
                .NREL
)2520 PROC1:    SUBZL 0, 0
                  .
                .END

                .TITL ROOT ;Root program.
                .EXTN METER
000001          .TXTM 1
                .ZREL
077777          .METER: METER
               .NREL
020411 START:  LDA 0, .OFILE ;Get name
006017          .SYSTM  ;and open ROOT.OL
012004          .OVOPN 4 ;on channel 4.
                  .
020000-         LDA 0, .METER ;Pointer.
126000          ADC 1,1 ;Uncond. load.
006017          .SYSTM
020004          .OVLOD 4 ;Load METER.
                  .
                  .
000024"         .OFILE: .+1*2
051117          .TXT "ROOT.OL"
047524
027117
046000
```

.EOF, .EOT
Explicit End-of-File

Syntax:

.EOF
.EOT

Purpose:

Either pseudo-op indicates the end of an input file but not the end of an input source; it provides an explicit end-of-file for any source module but the last one in a series for assembly. If .EOF pseudo-ops are missing from source modules, the assembler supplies them implicitly.

Example(s):

```
.TITL  MPROG
    .
    .
.EOF
```

Note that .EOF could be omitted in the example, with no effect on the assembly.

.EXTD
Define an External Displacement Reference

Syntax:

.EXTD user-symbol$_1$ *[...user-symbol$_n$]*

Purpose:

This pseudo-op declares each user-symbol as a symbol which may be referenced by this program but which is defined in some other program. The user-symbol must be declared by an .ENT pseudo-op in the program which defines it.

Any .EXTD user-symbol may be an address or displacement of a memory reference instruction. It can also specify the contents of a 16-bit storage word.

If used as a page zero address or as a displacement, the value of the entry must meet these specific requirements:

0 < page-zero-address < 377
-200 < displacement < 200

Note:

Because the displacement field of memory reference instructions must fit in 8 bits, RLDR will usually report an error if the user-symbol referenced has an address displacement of more than 8 bits. RLDR will not check the symbol, and thus not report an error, if the left byte of the instruction word is 0 (as it would be for a JMP instruction with an index mode of 0). Therefore, you should make sure that any JMP instruction without an index, which uses an .EXTD address, can be resolved in a displacement within 8-bit bounds.

Example(s):

```
        .TITL C
        .ENT  LIST
        .LOC 100
LIST:   LIST1
        LIST2
        LIST3
          .
          .
         .END

        .TITL D
        .EXTD LIST
        .NREL
         LDA 2, C2
         LDA 0, LIST, 2 ;Pick up 3rd
              ;LIST entry, LIST3.
          .
C2:      2
        .END
```

.EXTN
Define an External Normal Reference

Syntax:

.EXTN user-symbol$_1$ *[...user-symbol$_n$]*

Purpose:

This pseudo-op declares each user-symbol as a symbol that is externally defined in some other program but may be referenced by the current program. The user-symbol must be declared using an .ENT pseudo-op in the program which defines it.

An .EXTN user-symbol specifies only the contents of a 16-bit storage word. The value at load time must therefore be a number from 0 through 65,535.

Example(s):

```
               .TITL B
               .EXTN C
               .ZREL ;Put pointer in ZREL.
00000-077777   .C: C
               .NREL
                .
00000'006000-  JSR @ .C
```

.EXTU
Treat Undefined Symbols as External Displacements

Syntax:

.EXTU

Purpose:

This pseudo-op causes the assembler to treat all symbols that are undefined after pass 1 as if they had appeared in an .EXTD statement. Use .EXTU carefully; if you forget to define each .EXTU symbol elsewhere, RLDR will detect each undefined symbol.

Example(s):

```
          .TITLE A13
          .EXTU
'020001$  LDA 0, MURKO
           .
          .END
```

.GADD

Add an Expression Value to an External Symbol

Syntax:

.GADD user-symbol expression

Purpose:

.GADD (global add) generates a storage word whose contents is resolved at load time. The value of user-symbol is sought and, if found, its value is added to expression to form the contents of the storage word. If the user-symbol is not found, an RLDR error will result and the storage word will contain just the value of expression.

The user-symbol must be a symbol defined in some separately-assembled binary in conjunction with a .ENT, .ENTO, or .COMM pseudo-op.

Note:

To resolve .GADD user-symbol, RLDR requires that user-symbol be defined in a preceding relocatable binary. The file which defines user-symbol must precede any file(s) which use .GADD user-symbol, in the RLDR command line. If, after assembling the example below, you issued the command RLDR Y X, the value shared would be 207. If you transposed the loading order (RLDR X Y), then RLDR would resolve the value 7, and an error message would result.

Example(s):

```
                .TITLE Y
                .ENT A
       000200   .LOC 200
           A:              ;Value of A is 200.
                .END

                .TITLE X
                .EXTN A
       000100   .LOC 100
00100 000007    .GADD A, 3+4    ;Assembler
                           ;assigns value of 7.
                .END
```

.GLOC

Reserve an Absolute Data Block

Syntax:

.GLOC user-symbol

Purpose:

This pseudo-op tells RLDR to load the following block of data starting at the location assigned to user-symbol. You can terminate the absolute block by a .LOC, .NREL, .ZREL or .END pseudo-op.

Within the .GLOC block, there can be no external references, no label definitions, and no label references.

.GLOC reserves locations in memory, and these locations may impact on binaries loaded earlier or later in the RLDR command line.

Note:

One source file cannot both define user-symbol and use it in a .GLOC statement. You must define user-symbol via the .COMM (or .ENT) pseudo-op in one source file before you can use .GLOC user-symbol in another source file(s). The .GLOC file(s) must declare user-symbol external (.EXTN), or an assembler U error will occur. The defining binary must precede the .GLOC binary(ies) in the RLDR command line or a fatal RLDR error will occur.

Example(s):

```
                 .TITLE A
        000003   .COMM MYAREA, 3
                 .
                 .
                 .END

                .TITLE B  ;Program  B
                .NREL     ;will initialize
                .EXTN MYAREA  ;prog A's
                     ;named common area.
                .GLOC MYAREA
00000'000001    1
00001'000002    2
00002'000003    3
                .
                .END
```

.GOTO

Suppress Assembly Temporarily

Syntax:

.GOTO user-symbol

Purpose:

This pseudo-op suppresses the assembly of lines until the scan encounters another user-symbol enclosed in square brackets.

Example(s):

```
                .GOTO LABEL
                LDA 0,0,2 ;Don't assemble
                    ;this instruction.
040001 [LABEL] STA 0, TEMP   ;Start
               ;assembling again here.
000000  TEMP:  0
```

.GREF

Add an Expression Value to an External Symbol (OBO)

Syntax:

.GREF user-symbol expression

Purpose:

The .GREF (global reference) pseudo-op functions exactly like the .GADD pseudo-op except that when RLDR resolves the contents of the storage word (i.e., adds the value of the symbol and the value of the expression), a carry out of the low order fifteen bit positions is never allowed to alter bit zero.

Example(s):

See .GADD

.IFE, .IFG, .IFL, .IFN
Perform Conditional Assembly

Syntax:

.IFE expression
.IFG expression
.IFL expression
.IFN expression

Purpose:

The lines following these pseudo-ops will be assembled if the condition defined in the pseudo-op is met. You must always terminate the conditional lines with an .ENDC. The pseudo-ops define the following conditions:

.IFE expr	Assemble if expr equals 0.
.IFG expr	Assemble if expr is greater than 0.
.IFL expr	Assemble if expr is less than 0.
.IFN expr	Assemble if expr is not equal to 0.

The value field of the listings is 1 if the condition is true and 0 if the condition is false.

Example(s):

```
      000000    A=0
      000000    B=A
                .NREL
      000000    .IFE B-2
                LDA 0,A   ;Expression eval-
                .ENDC     ;uates to false
00000'020000    LDA 0, B ;in these cases,
            ;so the LDAs aren't assembled.
      000001    .IFL B-2  ;Expr evaluates
00001'020000    LDA 0, A  ;to true in
                .ENDC     ;these cases,
      000001    .IFN B-2  ;so the LDAs
00002'020000    LDA 0, A  ;are assembled.
                .ENDC
```

Notes:

.IFs may be nested to any depth, with the innermost .IF corresponding to the innermost .ENDC, etc. Note that all .IF conditions are degenerate forms of .DOs. For example: .IFG A is equivalent to .DO A > 0.

Chapter 5 describes handling of .IFs and .DOs within macros.

.LMIT

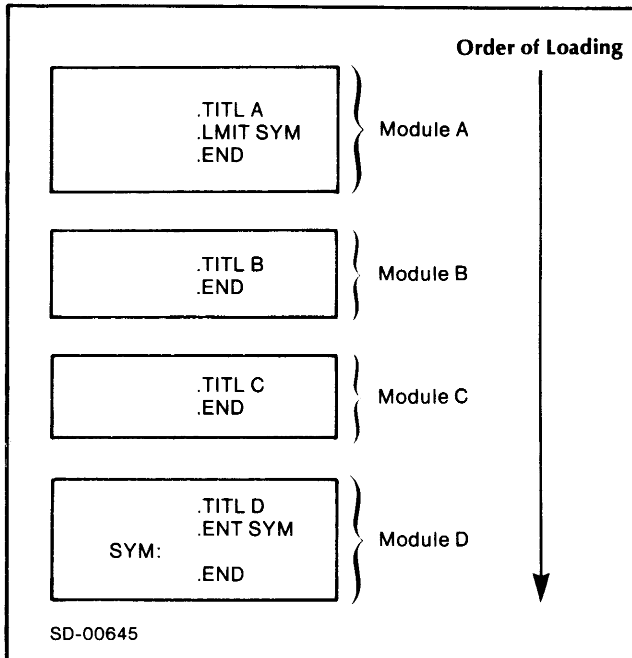
Load Part of a Binary Module

Syntax:

.LMIT user-symbol

Purpose:

This pseudo-op specifies partial loading of the assembler's binary output by RLDR. A .LMIT pseudo-op in one RB file will cause an RB later in the loading process to be partially loaded. user-symbol must appear as an .ENTry point in the later-loaded RB file. At load time, the RB will be bound only as far as the first occurrence of user-symbol; e.g.,



In this example, Module D contains the entry point SYM that corresponds to the user-symbol SYM appearing in the .LMIT pseudo-op in Module A. RLDR will bind D up to but not including the line identified by SYM.

The limiting symbol, in this case SYM, must be declared an entry point in the module to be partially loaded. If the limiting symbol is in NREL, all of Module D ZREL will be loaded and Module D NREL will be loaded up to the limiting symbol. If the limiting symbol is in ZREL, NREL will be completely loaded and ZREL will be loaded up to the limiting symbol. A module may be limited in NREL and in ZREL by two different symbols. If two symbols limit either NREL or ZREL, the lower symbol in value will be the limiting symbol. There are no restrictions on the number of limiting symbols that you may use.

If the limited module is in a library, the module will be loaded up to its limiting symbol, even if the module would otherwise not have been loaded (i.e., even if there is no undefined external to cause the library to be loaded.

If there is an undefined external that references an entry point in the unloaded part of the module, the module will still be only partially loaded as indicated by the limiting symbol.

All of the entry points of a partially loaded module will appear on the load map as though the corresponding parts of the module were actually loaded. Any references to them will be resolved, but, of course, will actually point into the succeeding module.

Note:

The .LMIT mechanism enters user-symbol in RLDR's symbol table as an entry point with a value of 100000_8.

After defining a .LMIT symbol, you must use it *only* as a limiting symbol in the program, because RLDR assumes that any later reference to this symbol is a .LMIT instruction, and doesn't flag it as a multiply-defined symbol.

.LOC

Set the Current Location Counter

Syntax:

.LOC expression

Purpose:

This pseudo-op sets the current location counter to the value and relocation property given by expression. The default value is absolute zero.

Exception:

If .LOC is .PUSHed to the assembler variable stack (see "Stack Pseudo-op and Values") and is subsequently used to restore the location counter, e.g..

```
.PUSH       .LOC

.LOC        .POP
```

then the *value* is ignored and only the relocation property is changed. This allows you to save the current relocation mode within a macro and restore it correctly, without affecting the relative location counter value which may have been altered within the macro.

Example(s):

```
 00000 000000  A:0
               .NREL
 00000'000000  NO:0
               .ZREL
 00000-000000  Z:0
       000100  .LOC 100
 00100 000000  A
U00101 000000  B  ;Undefined.
U00102 000000  C  ;Ditto.
       000001  .LOC A+1
 00001 000000  A
       000001- .LOC Z+1
 00001-000000  A
       000003- .LOC .+1
 00003-177777- Z-1   ;Minus 1.
 00004-000000- Z
```

.MACRO

Name a Macro Definition

Syntax:

.MACRO macro-name

Purpose:

This pseudo-op defines macro-name as the name of the macro definition that follows. Any line or lines that follow are part of the macro definition up to the first % character encountered.

After definition, macro-name calls the macro.

Example(s):

```
               .MACRO TEST ;Macroname.
               ^1       ;Macro
               ^2       ;defini-
               ^3       ;tion.
               %
        ;Call macro with arguments 4,5,6:
               TEST 4,5,6
00000 000004   4        ;Macro
00001 000005   5        ;defini-
00002 000006   6        ;tion.
        ;Change radix; call macro with
        ;arguments 0A, 0B, 0C.
      000024     .RDX 20
               TEST 0A,0B,0C
00003 000012   0A       ;Macro
00004 000013   0B       ;defini-
00005 000014   0C       ;tion.
```

.MCALL

Indicate Macro Usage

Syntax (in macro):

[conditional-or-.DO] .MCALL *[expression]*

Value:

.MCALL has value 1 if the macro containing it has been called on ths assembly pass, and value 0 if this is the first call on this pass. If used outside a macro, its value is -1.

Example(s):

```
.MACRO TEST2
.DO   .MCALL>0
JSR @ .X   ;JSR if not 1st call.
.ENDC
.DO   .MCALL==0    ;If 1st call, gen-
                   ;erate subroutine.
.PUSH .LOC         ;Save location counter.
.ZREL
.X: X          ;Pointer to subroutine.
.LOC .POP      ;Restore loc. counter.
JSR X          ;Call  X.
JMP XEND       ;Jump past X on rtn.
X: 'HI'        ;Code which will
.              ;assemble only once.
JMP 0, 3       ;Return.
XEND: .ENDC
%
```

.NOCON

Inhibit or Re-enable Listing Condition Lines

Syntax:

.NOCON expression

Purpose:

This pseudo-op either inhibits or permits listing of those conditional portions of the source program that do not meet the conditions given for assembly. If the value of expression is not zero, listing is inhibited; if the value of expression equals zero, listing occurs. If you omit .NOCON, listing occurs.

.NOCON does not affect conditional portions of the source program that would be assembled.

Value:

The value of .NOCON is the value of the last expression.

Example(s):

```
                 ;Listing:     Source Program:

        000003  A=3        ;   A=3
        000000 .NOCON 0    ;   .NOCON 0
        000000 .DO 4==A    ;   .DO 4==A
               5           ;   5
               3           ;   3
               .ENDC       ;   .ENDC

        000001 .DO 4==(A+1);.DO 4==(A+1)
00007'000005 5             ;   5
00010'000003 3             ;   3
               .ENDC       ;   .ENDC

        000001 .NOCON 1    ;   .NOCON 1
                           ;   .DO 4==A
                           ;   5
                           ;   3
                           ;   .ENDC

        000001 .DO 4==(A+1);.DO 4==(A+1)
00011'000005 5             ;   5
00012'000003 3             ;   3
               .ENDC       ;   .ENDC
```

.NOLOC

Inhibit or Re-enable Listing Source Lines Without Location Fields

Syntax:

.NOLOC expression

Purpose:

This pseudo-op either inhibits or permits listing of lines which lack a location field. If the value of expression does not equal zero, listing is inhibited; if the value of expression equals 0, listing occurs. If you omit .NOLOC, listing occurs.

Value:

The value of .NOLOC is the value of the last expression.

Example(s):

```
;Source Program:

.TXTM 1
.NREL
.NOLOC 0
.TXT "ABCDEFGHIJKL" ;Won't print.
.NOLOC 1 ;Nor will this print.
.TXT "ABCDEFGHIJKL" ;Prints.
LDA 0, .TMP     ;LDA prints
                ;because it has a
                ;location field.
.LOC .+10       ;This won't print.
.TMP: 0         ;This prints.
.END            ;.END won't print.

 ;Listing:

000001 .TXTM 1
       .NREL
000000 .NOLOC 0
040502 .TXT "ABCDEFGHIJKL"
041504
042506
043510
044512
045514
000000
040502 .TXT "ABCDEFGHIJKL" ;Prints.
020411 LDA 0, .TMP     ;LDA prints
000000 .TMP: 0         ;This prints.
```

.NOMAC

Inhibit or Re-enable Listing Macro Expansions

Syntax:

.NOMAC expression

Purpose:

This pseudo-op either inhibits or permits the listing of macro expansions. If expression evaluates to zero, macro expansions will be listed; otherwise, macro expansions are inhibited. .NOMAC can also be used within a macro definition to inhibit listing selectively. If you omit .NOMAC, expansions are listed.

Value:

The value of .NOMAC is the value of the last expression.

Example(s):

```
                    .MACRO OR
                    COM ↑1, ↑1
                    AND ↑1, ↑2
                    ADC ↑1, ↑2
                    %

             000001 .NOMAC 1 ;Do not expand.

                    ;Call macro with args 1 and 2:
      00000 124000  OR [1,2]


             000000 .NOMAC 0      ;Resume expanding.

                    ;Call macro with args 3 and 0:

      00003 174000  OR [3,0]              COM 3, 3
      00004 163400  AND 3, 0
      00005 162000  ADC 3, 0


                    ;Here is another macro:
                    .MACRO TEST4
                    5
                    6
                    .NOMAC 1 ;You can inhibit
                       ;or re-enable expansions
                       ;at any time in a macro.
                    7
                    10
                    %
                    ;Now, call TEST4:
                    TEST4
      00006 000005  5
      00007 000006  6
```

.NREL

Specify NREL Code Relocation

Syntax:

.NREL

Purpose:

This pseudo-op assembles the following code using NREL code relocation.

Example(s):

```
                .NREL
00000'000123  EXMP1:   123
00001'000456           456
```

.PASS

Number of Assembly Pass

Syntax

[conditional-or-.DO] .PASS *[expression]*

Value:

.PASS has a value of zero on pass 1 and a value of one on pass 2 of assembly.

Example:

The following macro, HIFND, picks the largest argument from a list of positive numbers and assembles it into a location at the end of pass 1:

```
.MACRO HIFND
.IFE  .PASS
  I=2  ;Init counter.
  ^1=0
  .DO  .ARGCT-1
  .IFG  ^1-^1
     ^1=^I
  .ENDC
  I=I+1
  .ENDC
.ENDC
^1
%
```

The calling sequence for this macro is as follows:

HIFND temp-sym $value_1$ ..., $value_n$

HIFND uses temp-sym to hold the current highest value and the resulting highest value in the argument list. That value is then put into a storage word at the current location counter.

.POP

Pop the Value and Relocation of Last Item Pushed onto Stack

Syntax:

[expression] .POP

Value:

The value of .POP is the value and relocation property of the last expression pushed onto the variable stack (.PUSH pseudo-op). In addition .POP pops the value and relocation property.

If there are no values on the variable stack, .POP has a value of zero (i.e., absolute relocation) and an overflow flag (O) will be given to the line in which it is used.

Example(s):

```
       000025   A=25
 00000 000025   A
       000025   .PUSH A
       000015   A=15
 00001 000015   A

       000025   A=.POP
 00002 000025   A
000003 000000   .POP ;Overflow
                     ;error.
                .END
```

```
                .NREL
00000'000100    .BLK 100
      000100'   A=.
00100'000100'   .
      000100'   .PUSH A
      000101'   A=.
00101'000101'   A
      000100'   A=.POP
00102'000100'   A
                .END
```

.PUSH

Push a Value and its Relocation Property onto a Stack

Syntax:

.PUSH expression

Purpose:

This pseudo-op allows you to save the value and relocation properties of any valid assembler expression on the assembler stack. Additional expressions may be pushed until the stack space is exhausted. The stack is referenced by the permanent symbols .POP and .TOP. As with any push down stack, the last expression pushed is the first expression to be popped.

Example(s):

The current value of the input radix may be saved, its value altered, then restored, by the following statements:

```
000010   .RDX 8
000010   .PUSH .RDX
000012   .RDX 10
000010   .RDX .POP
```

.RB

Name a Relocatable Binary File

Syntax:

.RB filename

Purpose:

This pseudo-op names the relocatable binary file, filename, that is the output of MAC assembly. If more than one .RB pseudo-op occurs in the source, the .RBs will be flagged with an M (multiply-defined symbol) and the binary file will have the name given in the first pseudo-op encountered.

If you insert the global /N switch in the MAC command line, denoting that no binary file is to be created, the .RB pseudo-op is ignored. If you use the local /B switch, the .RB pseudo-op will be overridden and the binary file will have the name given preceding the /B switch. The precedence for naming the relocatable binary file is thus:

Precedence	Binary Name
Highest	/B name
	.RB name
Lowest	Default name (first name in MAC line)

One of the primary uses of .RB is in conditional assembly code when alternative file names are to be used, depending upon the type of assembly; for example, in mapped and unmapped versions of an .RB file.

Example(s):

```
000001   .IFE MSW ;MSW means
         ;mapped switches- See file
         ; "*RDOS.SR".
         .RB SYSTEM.RB     ;Name binary
         .ENDC             ;"SYSTEM.RB".
000000   .IFN MSW          ;Name binary
         .RB MSYST.RB      ;"MSYST.RB"
         .ENDC             ;(mapped).
```

.RDX

Radix for Numeric Input Conversion

Syntax:

.RDX expression

Purpose:

This pseudo-op defines the radix to be used for numeric input conversion by the assembler. expression is evaluated in decimal and the range of expression is:

2<expression<20

The default radix is 8.

Value:

The numeric value of .RDX is the current input radix.

Note:

Input and output radices are entirely distinct. Setting the input radix does not affect the listing radix.

Syntax:

(.RDX)

Example(s):

(Assuming a listing output radix of 8:)

```
      000010   .RDX 8
00000 000123   123
      000012   .RDX 10
00001 000173   123
      000020   .RDX 16
00002 000443   123
00003 000020   (.RDX) ;Current value of
                      ;input radix.
```

.RDXO
Radix for Numeric Output Conversion

Syntax:

.RDXO expression

Purpose:

This pseudo-op defines the radix to be used for number conversion by the assembler. The expression is evaluated in decimal and the range of expression is

8 < expression < 20

The default output radix is 8.

Value:

The numeric value of .RDXO is always expressed as "10". .RDXO is printed as "(.RDXO)".

Example(s):

```
      000012   .RDX 10   ;Input radix 10.
       00010   .RDXO 10  ;Output radix 10.
00004  00077   77        ;Decimal listing.
00005  00022   22
00006  00045   45
       00008   .RDX 8    ;Input radix 8.
      000010   .RDXO 8   ;Output radix 8.
00007 000077   77
00010 000022   22
00011 000045   45
      000020   .RDX 16   ;Input rdx 16.
        0010   .RDXO 16  ;Output rdx 16.
0000A   0077   77        ;Hex listing.
0000B   0022   22
0000C   0045   45
      000010   .RDXO 8   ;Output rdx 8,
00015 000167   77    ;input stays 16-
00016 000042   22        ;octal listing,
00017 000105   45        ;hex input.
       00010   .RDXO 10 ;Output rdx 10-
00016  00119   77        ;Decimal listing,
00017  00034   22        ;Hex input.
00018  00069   45
       00010   .RDX 10   ;Input rdx 10,
        0010   .RDXO 16 ;Output rdx 16.
00013   004D   77        ;Dec. input,
00014   0016   22        ;hex listing.
00015   002D   45
00016   0010   (.RDXO) ;Value of output rdx
                       ;always prints as 10.
        0008   .RDX 8    ;Restore radixes.
      000010   .RDXO 8
```

.REV
Set the Revision Level

Syntax:

.REV major-revision-no. minor-revision-no.

Purpose:

This pseudo-op identifies the revision level of a program; it may be placed anywhere in the source module. The major and minor revision levels are entered as numbers in the current radix. Revision levels are carried into the RB file and then into the save file. Both the major and minor revision levels have a numeric range of 0-99.

If two or more RB files containing revision numbers are to be loaded into a program, RLDR chooses the revision level for the file as follows:

- If the save file is to have the same name as any RB file, the revision in that RB will be carried over to the save file.
- Otherwise, revision level information will be selected from the first RB loaded that contains such information.
- If none of the modules being loaded contains revision information, the save file will be assigned major and minor revision number 00.00.

For example, assume that SCHED.RB, IODRIV.RB, and DISP.RB are to be loaded into SCHED.SV. If SCHED.RB contains revision information, that revision information will be passed to the program file. If SCHED.RB does not contains revision information, the revision information contained in either IODRIV.RB or DISP.RB will be passed depending upon which module is loaded first.

Use the CLI command REV to obtain revision information of a save file.

Example(s):

```
             .TITL MART
             .EXTN .TASK, .LIM
      000001 .TXTM 1
00422 005001 .REV 12, 1  ;Revision
             ;level information is
             ;in octal (default input
             ;radix.
```

.TITL

Entitle an RB file

Syntax:

.TITL user-symbol

Purpose:

This pseudo-op names a binary file. This title is required by the library file editor to distinguish binaries that have been grouped into a library. The title given is printed at the top of every listing page. The user-symbol need not be unique from other symbols defined by the program, but it should not be used as a macro name.

If you omit .TITL, the assembler supplies the default title: .MAIN.

Example(s):

```
         .TITL SYMB
000001   .TXTM 1
           .
```

.TOP

Value and Relocation of Last Stack Expression

Syntax:

.TOP

Value:

.TOP has the value and relocation property of the last expression pushed to the variable stack. .TOP differs from .POP in that the symbol does not pop the last pushed expression from the stack. If no expressions are pushed, zero (absolute relocation) is returned and the overflow flag (O) is given.

Example(s):

```
      000020   .PUSH 20
00027 000020   .TOP
00030 000020   .TOP
```

.TXT, .TXTE, .TXTF, .TXTO
Specify a Text String

Syntax:

.TXT *a* string *a*
.TXTE *a* string *a*
.TXTF *a* string *a*
.TXTO *a* string *a*

Purpose:

These pseudo-ops cause the assembler to scan the input following the character *a* up to the next occurrence of the character *a* in string mode. The character *a* may be any character not used within the string except null, line feed, or rubout; *a* delimits, but is not part of, the string. You may use RETURN or FORM FEED to continue the string from line to line or page to page, but these characters are not stored as part of the text string.

Every two bytes generate a single storage word containing ASCII codes for the bytes. Storage of a character of a string requires seven bits of an eight-bit byte. You can set the leftmost (parity) bit to 0, 1, even parity or odd parity as follows:

.TXT	Sets leftmost bit to 0 unconditionally.
.TXTF	Sets leftmost bit to 1 unconditionally.
.TXTE	Sets leftmost bit for even parity on byte.
.TXTO	Sets leftmost bit for odd parity on byte.

By default, bytes are packed left/right, and a null byte is generated as the terminating byte.

You can change the packing mode with the .TXTM pseudo-op. If an even number of bytes are assembled, you can suppress the null word following these packed bytes with the .TXTN pseudo-op.

Witin the string, you can use angle brackets (< >) to delimit an arithmetic expression. The expression will be evaluated, masked to seven bits, and the eighth bit set as specified by the pseudo-op. This is the only means, for example, to store a carriage return and/or line feed character as part of the text string. Note that *no* logical operators are permitted within the expression.

.TXT "LINE 1 <15> <12>")

Example(s):

```
      000001    .TXTM 1
00000 040502    .TXT "AB CD" ;Each example
      020103
      042000
00003 040502    .TXTE *AB CD* ;assembles
      120303
      042000
00006 140702    .TXTF 'AB CD' ;text string
      120303
      142000
00011 140702    .TXTO /AB CD/ ;"AB CD".
      020103
      142000
```

.TXTM

Change Text Byte Packing

Syntax:

.TXTM expression

Purpose:

This pseudo-op changes the packing of bytes generated using the text pseudo-ops, .TXT, .TXTE, .TXTF, or .TXTO. If expression evaluates to zero, bytes are packed right/left; if expression does not evaluate to zero, bytes are packed left/right. If you omit .TXTM, bytes are packed right/left.

Value:

The value of .TXTM, expressed as (.TXTM), is the value of the last expression.

Example(s):

```
      000000   .TXTM 0
00000 041101   .TXT "AB CD"
      041440
      000104
00003 000000   (.TXTM)
      000001   .TXTM 1
00006 040502   .TXT "AB CD"
      020103
      042000
00011 000001   (.TXTM)
```

.TXTN

Determine Text String Termination

Syntax:

.TXTN expression

Purpose:

This pseudo-op determines whether or not a string that contains an even number of characters will terminate with a word consisting of two zero bytes. (When the number of characters in the string is odd, the last word contains a zero byte in all cases.) If you omit .TXTN, the string terminates on a zero word.

If expression evaluates to zero, all text strings containing an even number of bytes will terminate with a full word zero. If expression does not evaluate to zero, any text string containing an even number of bytes terminates with a word containing the last two characters of the string.

Value:

The value of .TXTN, expressed as (.TXTN), is the value of the last expression.

Example(s):

```
      000000   .TXTN 0
00000 030462   .TXT "1234"
      031464
      000000
00003 000000   (.TXTN)
      000001   .TXTN 1
00004 030462   .TXT "1234"
      031464
00006 000001   (.TXTN)
```

.XPNG
Remove All Nonpermanent Macro and Symbol Definitions

Syntax:

.XPNG

Purpose:

This pseudo-op removes all macro definitions and all symbol definitions, except permanent, from the assembler's symbol table. .XPNG is used primarily as follows:

1. You write a program containing .XPNG followed by definitions of any semipermanent symbols.
2. The program is assembled using the global switch /S to the MAC command. This causes the assembler to stop the assembly after pass 1 and save the symbols in MAC.PS.
3. You can then use the MAC.PS with those semipermanent symbols defined in step 2 to assemble other files.

Chapter 6 further describes this mechanism.

Note that file NBID.SR begins with a .XPNG.

Example(s):

```
.TITL XP
.XPNG
.DMRA LDA=20000
.DMRA STA=40000
.END
```

After assembling and loading XP, you issue the CLI command:

MAC/S XP)

The assembler's symbol table now contains values for LDA and STA.

.ZREL
Specify Page Zero Relocation

Syntax:

.ZREL

Purpose:

This pseudo-op assembles subsequent source lines using page zero relocatable addresses (these addresses extend from 50_8 through 377_8. If ZREL mode is exited during assembly, the current .ZREL value is maintained and is used if .ZREL mode is entered again.

Example(s):

```
00064 000000    AL:0
                 .ZREL
00000-000000     Z:0
00001-000000    ZL:0
      000100     .LOC 100
00100 000064     AL
                 .ZREL
00002-000064     AL
```

End of Chapter

Chapter 5
Macros and Other Advanced Features

The Macro Facility

The macro facility allows a string of source characters, perhaps consisting of many lines, to be named and subsequently referenced by name. This string of source characters is the *macro definition* (or simply "macro"). While defining a macro, you assign it a symbol; this symbol will represent a *macro call* whenever it appears within your program. During assembly, the symbol expands to the original string. The original string may contain *formal arguments,* and the macro call *actual arguments;* the *actual arguments* replace the formal arguments as each macro call expands the macro symbol.

Macro Definition

You write a macro definition once, but use the macro as often as needed. Each macro definition has the form:

```
.MACRO□user-symbol)
macro-definition-string %
```

where:

user-symbol is the name which calls the macro. This name cannot exceed five characters.

macro-definition-string is a string of ASCII characters to be substituted for the macro call.

% terminates macro-definition-string and is not part of the definition.

Your user-symbol must conform to the standard rules for user symbols. Within macro-definition-string two characters (← and ↑) have special meanings. The back-arrow (←) stores the very next character without interpretation; the back-arrow is otherwise ignored. The back-arrow convention is generally used to render literally a character that would otherwise be interpreted (%, ↑, or ←). It can be used with any ASCII character: ← X and X will both be read as X. On some terminals, you enter a backarrow as SHIFT-O.

For example, if you want to use a percent sign (which terminates a macro definition) within a macro definition string, you would use the backarrow. For the definition string *ABC IS 15% OF D,* the format would be *ABC IS 15 ← % of D%.*

For a macro-definition-string containing a backarrow literally, such as:

```
X←YZ
```

the format would be:

```
X←←YZ
```

The second character with a special meaning is the uparrow (↑). An uparrow followed by an alphanumeric character specifies arguments for macro expansion. This convention has the following form:

↑ *n*	where *n* is a digit from 1 to 9.
↑ *a*	where *a* is a single letter from A to Z.
↑? *a*	where *a* is a single character from the following set: A-Z, 0-9, and ?

A digit following ↑ represents the position of an actual argument in the argument list of the macro call. The argument in position *n* will replace the formal argument *n* wherever *n* appears within the macro definition. For example, if the formal argument ↑3 occurs in the macro definition string, then ↑3 will be replaced by the third argument in the macro call, as described in the next section. (A zero following ↑ is unconditionally replaced by the null string.)

↑ *n* can be used only for arguments 0-9. To reference arguments 10-63, you must define symbol in the form of *a* or *?a* having the desired argument number in the range 10-63. By convention, we use ?0-?9 to reference arguments 10 through 19. For example:

```
        ?0=10.
          .
          .
        .MACRO A
        CB=^?0
        %
          .
          .
        Z=7
;Call macro with ten args:
        A 1 2 3 4 5 6 7 8 9 Z
;CB now has the value of the
;tenth arg, Z, which is 7.
          .
          .
```

An *a* or *?a* following ↑ is a symbol whose value the assembler looks up when it expands the macro. The value of the symbol indicates position of the macro argument that replaces it (as in ↑ *n*). The value of *a* or *?a* ranges from 1 through 63, since no macro can have more than 63 arguments.

The carriage return following user-symbol is required to distinguish user-symbol from the macro definition string.

Aside from the ↑, ←, and %, all characters return from macro expansion as they were written. Carriage returns are not inserted automatically into macro definitions; when a macro definition string contains more than one line of source language, you must terminate each line (except the last) with an explicit carriage return.

You must define each expression in a macro definition string within one line; an expression cannot be broken by a carriage return or comments. Thus each single expression can have a maximum of 132 characters, the line limit of the assembler.

If the macro definition string requires more than one line, you should use the % terminator as the last line. If the definition does not fill a single line, terminate the definition with a %.

Examples of macro definitions are:

```
.MACRO T            ;T is equivalent
LDA 0,0,3           ;to this two-
MOV 0,0, SNR        ;instruction sequence.
%

.MACRO EXP          ;EXP defines TEST as
TEST^1+^2 %         ;the sum of two
                    ;arguments.

.MACRO  COMM        ;COMM expands to a
;TEST FOR 95_% DONE.%  ;comment line.
```

The use of macros is illustrated further in the remainder of this chapter.

The definition of a macro may be temporarily terminated and then continued. This is especially useful when you use a first macro to define a second macro. The first macro may terminate definition of the inner macro temporarily, assign new values and continue. The macro VFD, described later, illustrates this continuation property.

Syntactically, when a macro of the same name as the last defined macro is encountered, the second and subsequent "definitions" are appended, in order, to the first definition. For example,

```
        .MACRO TEST
        I=0
        %

        .MACRO TEST
        J=0
        %

;These two macros are equivalent to:

        .MACRO TEST
        I=0
        J=0
        %
```

Macro Calls

You can place any number of macro calls for a given macro in your source program. A macro call consists of the user symbol in a macro definition followed by actual arguments to replace the formal arguments (if any) in the macro definition string.

A macro call has one of the following forms:

1. user-symbol
2. user-symbol □ $string_1$ *[□ string]*
3. user-symbol, *[□]* left-bracket *[□]* $string_1$)) *[string][...]* right-bracket

where:

user-symbol is the name you assigned to the macro definition.

Each string is an actual argument that will replace the appropriate formal argument during macro expansion.

left-bracket and right-bracket are ASCII brackets (we use these terms because the brackets themselves are notation conventions, meaning optional entries).

The first form of the macro instruction presumes that there are no formal arguments within the macro definition. Forms 2 and 3 presume that one or more formal arguments must be replaced by actual arguments. If an argument list extends to one or more additional lines, the carriage return character acts as an argument delimiter (and therefore should not be preceded by comma or space).

During macro expansion, $string_1$ will replace every occurrence of ↑1 (or replace ↑ *a* where *a* evaluates to 1), $string_2$ will replace every occurrence of ↑2, and so on. If more actual arguments are given by the call than were specified formally by the definition, they are ignored; if the definition specified none, all call arguments will be ignored. No macro can have more than 63_{10} arguments.

The following rules govern argument lists:

1. All leading spaces and tabs are ignored.

2. Multiple, contiguous imbedded spaces and tabs are treated as a single break character.

3. Multiple commas are treated as multiple null arguments, and a leading comma is treated as a null first argument.

Figure 5-1 shows how spaces, tabs, and commas in macro calls expand.

```
.MACRO EXAMP     ;Define the macro:
;
;A1=^1   A2=^2   A3=^3   A4=^4   A5=^5
;***********************************
%

EXAMP A B C      ;1 space between args.
;
;A1=A   A2=B   A3=C   A4=   A5=
;***********************************

EXAMP  A  B  C  ;2 spaces between args.
;
;A1=A   A2=B   A3=C   A4=   A5=
;***********************************

EXAMP   A       B       C ; 1 tab.
;
;A1=A   A2=B   A3=C   A4=   A5=
;***********************************

EXAMP           A B C  ;2 tabs,1 space.
;
;A1=A   A2=B   A3=C   A4=   A5=
;***********************************

EXAMP   B               C ;1 tab,2 tabs.
;
;A1=B   A2=C   A3=   A4=   A5=
;***********************************

EXAMP ,B,C       ;Leading comma.
;
;A1=   A2=B   A3=C   A4=   A5=
;***********************************

EXAMP ,,B,C      ; 2 leading commas.
;
;A1=   A2=   A3=B   A4=C   A5=
;***********************************

EXAMP ,,,B,C     ; 3 leading commas.
;
;A1=   A2=   A3=   A4=B   A5=C
;***********************************

.END
```

Figure 5-1. Macro Calls and Expansions

The list of arguments in a macro call may either be enclosed in square brackets (form 3), or not (form 2). Form 2 calls are terminated with a carriage return before the first byte of macro expansion, whereas form 3 calls are not. To replace the index field of an instruction, use a form 3 call.

In form 2, you use a return to terminate the argument list. In form 3, you use a right bracket (]) to terminate the argument list. Therefore, if you have more arguments than you can write on one line, use form 3. Remember that RETURN acts as an argument delimiter. For example:

```
ABC[1,2)
3,4]
```

calls macro ABC with 4 arguments. The call

```
ABC[1,2,)
3,4]
```

calls macro ABC with 5 arguments. The third argument (which follows the second comma) is a null.

Figure 5-2 shows a simple macro, form 2 and 3 calls to the macro, and a consequent macro expansion.

Macro Definition

```
.MACRO D
TEMP ↑1%
```

Macro Calls and Their Expansions

```
D2+3            ;FORM 2 MACRO CALL

TEMP2+3         ;MACRO EXPANSION

STA 3,D [2]     ;FORM 3 MACRO CALL
                ;(BRACKETS ARE LITERAL
                ENTRIES).
STA 3, TEMP2    ;MACRO EXPANSION.
```

Figure 5-2. Forms 2 and 3 Macro Calls

The action performed by the two asterisks atom (**) is unique in form 3 calls. If the first line in a form 3 macro definition starts with two asterisks, the last line of arguments will not be printed but the macro will assemble correctly.

Argument strings, like text strings, may consist of any characters. You can separate argument strings by a space, comma, horizontal tab, or return; but a leading comma indicates a null first argument.

In a macro call, the assembler stops scanning arguments when it encounters a semicolon (;). To include arguments whch follow a semicolon, insert a backarrow immediately before the semicolon. For example, in the call

```
MYMAC ARGA ARGB ; ARGC
```

ARGC would be dropped, but in the call

```
MYMAC ARGA ARGC ←; ARGC
```

ARGC would be included.

Listing of Macro Expansions

Macro definitions replace macro calls in the binary file and in listings of macro expansions. However, the listing output showing the expanded source text differs from the macro expansion that generates RB file output. The listing shows both macro calls and macro expansions, while the file contains only the RB codes of the macro expansions with their appropriate actual arguments. An example is:

```
.MACRO MYMAC
^1%

LDA 0 MYMAC [121] 3  ;Line from source
                     ;program.
LDA 0 MYMAC [121]  121 3   ;Expanded
                           ;line from
                         ;listing file.
LDA 0  +121  3  ;Expanded line from
                ;RB binary file.
```

You can suppress the listing of macro expansions with the pseudo-op .NOMAC. If suppressed, the load instruction above would appear on the listing exactly as it appears in the source listing line. The following example shows the results of suppressing macro expansion listings.

```
        .MACRO Z
        5
        LDA   ^1,^2
        %

;By default, expansions are listed:

        Z         0,4       ;Call Z.
        5
        LDA   0,4

; .NOMAC suppresses expansion in
; listings, unless it precedes an
; expression which evaluates to 0.

        .NOMAC 1

        Z 0,4    ;Call Z.

;Re-enable listing of expansions:

        .NOMAC 0

        Z         0,4       ;Call Z.
        5
        LDA 0,4

        .END
```

.DO Loops and Conditionals

In any macro, you should terminate each .DO loop or .IF conditional with a .ENDC. If you omit .ENDC in a .IF conditional, the assembler supplies the .ENDC before the terminating %. It's good programming practice, however, to include the .ENDC. If you omit .ENDC from a .DO loop and terminate this .DO with %, the code will be assembled once (if the argument to .DO is more than 0), or not at all (if the argument to .DO is 0).

The following example shows a macro, X, which defines four storage words containing the values 1, 2, 3, and 4 respectively. This macro contains a .DO statement which is not terminated by a .ENDC statement. Since it goes unterminated, the .DO is ignored (in other words, the macro is not written as it should be, but is presented here to make the point that .DO statements must be terminated by .ENDC statements or they will be ignored).

The calling sequence contains a .DO statement and a terminating .ENDC statement. Since this application of the .DO facility is proper, it causes the macro and storage words "7" and "3" to be repeated three times. Note that in the assembly listing the line ".DO 2" appears three times since it is part of the macro X.

However, since that .DO statement in X has no .ENDC terminator, it has no effect in the macro expansion even though it does appear in the listing.

In this example, no K error occurs because the .ENDC always terminates the .DO 3, not the .DO 2 in the macro. The .DO 2 loop is never repeated because it is discarded before it hits a .ENDC.

```
                  ;The macro definition of X is:

                  ;         .MACRO X
                  ;         1         ;First arg in macro.
                  ;         2
                  ;         .DO 2     ;This DO does nothing.
                  ;         3
                  ;         4         ;Last arg in macro.
                  ;%

                  ;The macro call to X is part of a properly-
                  ; terminated DO loop, as follows:

                  ;         .DO 3
                  ;         7         ; All args in this DO
                  ;         X         ; loop, and in the macro,
                  ;         3         ; repeat 3 times because
                  ;         .ENDC     ; it is closed by ENDC.

                            .MACRO X
                            1         ;First arg in macro.
                            2
                            .DO 2     ;This DO does nothing.
                            3
                            4         ;Last arg in macro.
                  %

       000003               .DO 3
00000  000007               7
                            X
00001  000001               1         ;First arg in macro.
00002  000002               2
       000002               .DO 2     ;This DO does nothing.
00003  000003               3
00004  000004               4         ;Last arg in macro.
00005  000003               3
                            .ENDC
00006  000007               7
                            X
00007  000001               1         ;First arg in macro.
00010  000002               2
       000002               .DO 2     ;This DO does nothing.
00011  000003               3
00012  000004               4         ;Last arg in macro.
00013  000003               3
                            .ENDC
00014  000007               7
                            X
00015  000001               1         ;First arg in macro.
00016  000002               2
       000002               .DO 2     ;This DO does nothing.
00017  000003               3
00020  000004               4         ;Last arg in macro.
00021  000003               3
                            .ENDC
```

Macro Examples

A number of macro examples follow in Figures 5-3 through 5-6. Note the use of the recursive property in the macro, FACT, and the use of macro continuation and the special character ← within VFD.

The first example is a macro to compute the logical OR of two accumulators. Its call takes form similar to an ALC instruction, i.e.,

OR source-ac □ destination-ac

The source accumulator is unchanged by the call. Note also that actual arguments replace formal arguments within the comments.

If you have an ECLIPSE computer, you could use the IOR instruction to OR the accumulators. Treat this macro as an example.

```
      ;Logical OR macro.

      ;Macro definition:
            .MACRO OR
            COM    ^1,^1       ;Clear "ON" bits
            AND    ^1,^2       ;of AC^1.
            ADC    ^1,^2       ;OR result to AC^2.
            COM    ^1,^1       ;Restore AC^1.
            %

      ;The macro call has the form:
              ;  OR   acs,acd
      ;and the expansion is:
              ;  acs  .OR.  acd

    ;The following calls to OR:
    ;      OR 1,2
    ;      OR 0,1
    ;produce this expansion.  Note affect on
    ;AC comments:

                    OR     1,2
    00000'124000    COM    1,1 ;Clear "ON" bits
    00001'133400    AND    1,2 ;of AC1.
    00002'132000    ADC    1,2 ;OR result to AC2.
    00003'124000    COM    1,1 ;Restore AC1.
                          OR      0,1
    00004'100000    COM    0,0 ;Clear "ON" bits
    00005'107400    AND    0,1 ;of AC0.
    00006'106000    ADC    0,1 ;OR result to AC1.
    00007'100000    COM    0,0 ;Restore AC0.
```

Figure 5-3. Logical OR Macro

FACT is a factorial macro, and illustrates the recursive property of macros. Its input consists of an integer, *i*, and a variable, *v*, and it computes the value:

$v = i!$

using the recursive formula

$i! = i * (i-1)!$

First, FACT checks for a negative number and either a 0 or 1; then, it expands the macro as follows:

Until the input integer becomes 1, the second conditional expands and recursively calls FACT. When the input becomes 1, the first conditional expands and terminates its expansion. This begins the succession of returns to each level at which a recursive call was made, in the process computing *i!*. All these recursive calls stack up until ↑ 1 becomes 1; they are then executed. After the final call, the macro will return the string:

↑2 = (↑1) *↑2

and terminate.

```
            ;The macro definition is:

                  .MACRO FACT
            **    .DO ^1<0              ;If negative,
            **    ^2=0                  ;return 0.
            **    .ENDC DONE

            **    .DO ^1<=1             ;If arg 1 is 0
            **    ^2=1                  ;or 1, return 1.
            **    .ENDC DONE

                  FACT ^1-1,^2          ;Else call yourself recursively.
                  ^2=^1*^2

                  [DONE]
                  %
            ;the macro call is FACT 6,I:

                  FACT 6,I


                  FACT 6-1,I            ;Else call yourself recursively.

                  FACT 6-1-1,I          ;Else call yourself recursively.

                  FACT 6-1-1-1,I        ;Else call yourself recursively.

                  FACT 6-1-1-1-1,I             ;Else call yourself recursively.

                  FACT 6-1-1-1-1-1,I           ;Else call yourself recursively.

                  FACT 6-1-1-1-1-1-1,I         ;Else call yourself recursively.
                  I=6-1-1-1-1-1*I

                  [DONE]
000002                   I=6-1-1-1-1*I
                  [DONE]
000006                   I=6-1-1-1*I
                  [DONE]
000030                   I=6-1-1*I
                  [DONE]
000170                   I=6-1*I
                  [DONE]
001320                   I=6*I
                  [DONE]
```

Figure 5-4. Factorial Macro

A macro to output "packed decimal" is given next. It illustrates a number of useful techniques.

In packed decimal, each decimal digit is represented as a 4-bit binary nibble. The sign of the number always occupies the least significant nibble. The translation of decimal to 4-bit binary is:

decimal	**4-bit binary nibble**
+	0011 (same bit pattern as "3")
-	0100 (same bit pattern as "4")
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

The input to PACK is the decimal string of digits, separated by spaces, followed by an explicit sign (+ or -) and the precision in 16-bit *words*. The macro outputs the *least* significant word first, and the number sign is stored in the rightmost nibble of this word.

Some further explanation is necessary:

1. The input radix within the macro must be *decimal*. Therefore, it is saved, set to decimal, and restored within the macro body.

2. To present the output as 4-bit nibbles, the output radix within the macro must be *hexadecimal*. Therefore, the output radix is also saved, set to hexadecimal, and restored. Note the order of the save for these radices is the opposite of the order of the restore. (See .PUSH and .POP descriptions in Chapter 4.)

3. Many statements are assembled with each macro call, but listing is inhibited except for the storage words assembled.

```
;The macro call has the form:
;       PACK  d d ... d  s,w
;where each d is a digit, s is the sign
;(+ or -) and w is the number of words.

;Here is the macro definition. We have
;shown it as part of the macro expansion,
;to avoid repetition.

          .MACRO   PACK
**        .PUSH    .NOMAC
**        .NOMAC   1
          .PUSH    .RDX
          .PUSH    .RDXO
          .RDX 10
          .RDXO 16
I=        .ARGCT
J=        I-1
B=        11
W=        3+("^J-"+/2)
J=        J-1
          .LOC     .+^I-1
          .DO      ^I
          .DO      B+1/4
W=        W+0^JBB
B=        B-4
          .DO      J<>0
J=        J-1
          .ENDC
          .ENDC
**        .NOMAC 0
          W
**        .NOMAC 1
W=        0
B=        15
          .LOC     .-2
          .ENDC
          .LOC     .+^I+1
          .RDXO    .POP
          .RDX     .POP
**        .NOMAC   .POP
%
```

```
;The following four calls to PACK
;produce the expansions below.

          .LOC 100
          PACK     1 2 3 4 5 +,3
          PACK     1 2 3 4 5 -,3
          PACK     6 5 4 3 2 1 -,4
          PACK     3 2 7 6 8 +,6

;(Normally, the macro definition would be
;repeated here.)  Expansions are:

            000100     .LOC 100
                       PACK     1 2 3 4 5 +,3
   00042    3453       W
   00041    0012       W
   00040    0000       W
                       PACK     1 2 3 4 5 -,3
   00045    3454       W
   00044    0012       W
   00043    0000       W
                       PACK     6 5 4 3 2 1 -,4
   00049    3214       W
   00048    0654       W
   00047    0000       W
   00046    0000       W
                       PACK     3 2 7 6 8 +,6
   0004F    7683       W
   0004E    0032       W
   0004D    0000       W
04 0004C    0000       W
05 0004B    0000       W
06 0004A    0000       W
```

Figure 5-5. Packet Decimal Macro

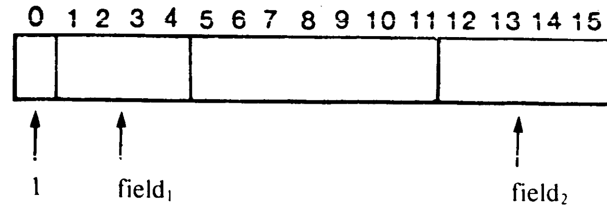
A powerful macro, used to associate a specified field layout with a given name, is shown below. The macro, VFD, defines a *new* macro named as the first argument in the call to VFD. Subsequent use of the name given in the VFD call generates a 16-bit storage word with a primary value to which fields are assembled as described in the call to VFD. The call has the form:

VFD□type-name□primary-value□$field_1$ -right-bit□↑)
$field_1$ -mask□...$field_n$ -right-bit□$field_n$ -mask...

The 3rd, 5th, ... arguments specify the rightmost bit positions of the 1st, 2nd, ... fields. The 4th, 6th, ... arguments specify the field masks for the 1st, 2nd, ... fields. To assemble the fields in the proper bit positions, with overflow and field zero checking, a call is made of the form:

type-name□$field_1$ □$field_2$ ...

The example that follows defines a type-name SPECL. This name is for words of the following layout:



The definition of VFD is:

```
;We define three macros in this figure: VFD, ERROR,
;and SPECL.  VFD actually uses ERROR to define SPECL.
;Note that spaces are critical in these macros- for example,
;SPECL 1,1 is NOT the same as SPECL 1, 1.  When you call VFD,
;you can use values other than we show-- on certain results,
;ERROR will print one of two error messages.

                  ;The macro definition for VFD is:

                           .MACRO VFD

                           I=4
                                   .MACRO ^1

                                   ** .PUSH   .NOMAC
                                   **.NOMAC 1
                                   VALUE=^2
                                   J=1
                  _%

                           .DO  .ARGCT/2-1
                                .MACRO ^1

                                .IFN ^I>=_^J
                                MASK=^I
                                DATA=_^J
                  _%
                           I=I-1
                                   .MACRO ^1
                                            .DO 15.-^I

                                            DATA=DATA*2
                                            .ENDC
                  _%
                           I=I+1
                                   .MACRO ^1

                                            .IFN VALUE&MASK
                                            ERROR [FIELD NONZERO]
                                            .ENDC

                                            .IFE VALUE&MASK
                                            VALUE=(VALUE&(-MASK-1))+DATA
                                            .ENDC
                           .ENDC

                           .IFE ^I>=_^J
                           ERROR [FIELD OVERFLOW]
                           .ENDC
                  _%
                           I=I+2
                           .MACRO ^1

                           J=J+1
                  _%
                           .ENDC
                                   .MACRO ^1

                                   **.NOMAC 0
                                   VALUE
                  _%               **.NOMAC  .POP
```

Figure 5-6 VFD, ERROR and SPECL Macros

```
;Here is the macro definition for macro ERROR:

                        .MACRO ERROR
                        ** .PUSH  .NOMAC
                        **.NOMAC  0
                ;***********************************

                ;        ^1 ^2 ^3 ^4 ^5 ^6 ^7 ^8 ^9
                ;***********************************
                        ** .NOMAC  .POP
                %


;The following call to macro VFD creates macro SPECL
;VFD uses ERROR in the process):

                        VFD  SPECL,100000,3,7,15.,17

        000004          I=4
                                .MACRO SPECL

                                ** .PUSH   .NOMAC
                                **.NOMAC 1
                                VALUE=100000
                                J=1
                %

        000002          .DO  .ARGCT/2-1
                             .MACRO SPECL

                             .IFN 7>=^J
                             MASK=7
                             DATA=^J
                %
        000003          I=I-1
                                .MACRO SPECL
                                        .DO 15.-3
                                        MASK=MASK*2
                                        DATA=DATA*2
                                        .ENDC
                %
        000004          I=I+1
                                .MACRO SPECL

                                        .IFN VALUE&MASK
                                        ERROR [FIELD NONZERO]
                                        .ENDC

                                        .IFE VALUE&MASK
                                        VALUE=(VALUE&(-MASK-1))+DATA
                                        .ENDC
                        .ENDC

                        .IFE 7>=^J
                        ERROR [FIELD OVERFLOW]
                        .ENDC
                %
        000006          I=I+2
                             .MACRO SPECL

                        J=J+1
                %
                        .ENDC
                             .MACRO SPECL
```

Figure 5-6 VFD, ERROR, and SPECL Macros (continued)

```
                              .IFN 17>=^J
                              MASK=17
                              DATA=^J
                  %
          000005          I=I-1
                                  .MACRO SPECL
                                       .DO 15.-15.

                                       DATA=DATA*2
                                       .ENDC
                  %
          000006          I=I+1
                                  .MACRO SPECL

                                       .IFN VALUE&MASK
                                       ERROR [FIELD NONZERO]
                                       .ENDC

                                       .IFE VALUE&MASK
                                       VALUE=(VALUE&(-MASK-1))+DATA
                                       .ENDC
                          .ENDC

                          .IFE 17>=^J
                          ERROR [FIELD OVERFLOW]
                          .ENDC
                  %
          000010          I=I+2
                          .MACRO SPECL

                          J=J+1
                  %
                          .ENDC
                                  .MACRO SPECL

                                  **.NOMAC 0
                                  VALUE


;Now, we can issue two calls to SPECL,
;with arguments 1,1 and 7,17.

                          SPECL 1,1

      00000 110001                VALUE
                          SPECL 7,17

      00001 170017                VALUE
```

Figure 5-6. VFD, ERROR, and SPECL Macros (continued)

Generated Labels

You can use the dollar sign ($) to generate unique labels within macros. In normal (nonstring) mode, each occurrence of the character $ is replaced by three characters from the set 0-9, A-Z. The three characters are determined by converting a count of the number of macro calls in radix 36 to ASCII. In nested macros, the replacement string for $ in the outer macro is saved and restored when the inner macro has been expanded.

When used in labels, $ should generally not be the first character, as the first replacement character may be a digit. If the number of macro calls on pass two differs from the number on pass one, the label will receive a different value on each pass, and produce phase errors when used.

Figure 5-7 shows the generation of label entries via the macro BKT.

```
;The macro definition is:

        .MACRO BKT
        DSZ      COUNT
  TR$=  .        ;Unique label.
%

;Now, call BKT 5 times with a .DO 5, BKT,
;.ENDC sequence, to produce 5 labels.

COUNT:  .
        .NREL
        .DO 5
        BKT
        .ENDC

        ;The expanded listing is:

   00000 000000 COUNT:  .
                        .NREL
         000005         .DO 5
                        BKT
   00000'014000         DSZ       COUNT
         000001'  TR$001=  .      ;Unique label.
                        .ENDC
                        BKT
   00001'014000         DSZ       COUNT
         000002'  TR$002=  .      ;Unique label.
                        .ENDC
                        BKT
   00002'014000         DSZ       COUNT
         000003'  TR$003=  .      ;Unique label.
                        .ENDC
                        BKT
   00003'014000         DSZ       COUNT
         000004'  TR$004=  .      ;Unique label.
                        .ENDC
                        BKT
   00004'014000         DSZ       COUNT
         000005'  TR$005=  .      ;Unique label.
                        .ENDC
```

Figure 5-7. Generating Labels

Literals

All memory reference instructions must specify an address field. This address is used to:

1. Access the contents of the memory location in the case of an LDA.
2. Modify the memory location in the case of an STA, ISZ, or DSZ.
3. Transfer control in the case of a JMP or JSR.

Often, however, you merely wish to specify the *contents* of a memory location and are not concerned about its address. Such a specification is called a *literal* reference (or simply a *literal*).

Literals are permitted for all memory reference instructions. The macro assembler dumps these literals and assigns relocatable memory locations using the first and subsequent .ZREL locations available after pass 1. Therefore, all literal references are directly addressable.

To conserve ZREL address space, define literals before you use them. The assembler assigns two storage locations to each forward reference to a location but only one to each backward reference. For example,

```
LDA 0,     =AFTER
           .
           .
           .
LDA 1,     =AFTER

AFTER:0
```

requires two locations - one for each reference to the literal. Whereas

```
BEFORE:0

LDA 0,     =BEFORE
           .
           .
           .
LDA 1,     =BEFORE
```

requires only one ZREL location.

The syntax of a literal reference is as follows:

memory-reference *[ac,]* = { expression | instruction }

Note that a literal may be any expression or instruction.

Frequently, literals are used to load an accumulator with some constant. For example,

```
LDA 1,=3
```

loads AC1 with the value 3.

Expressions are acceptable:

```
LDA 0,=1B0+"A/2
```

loads AC0 with the value 40040.

Instructions are also acceptable:

```
LDA 1,=SUBZ# 2,3,SNC
```

loads AC1 with the value 156433.

The previous examples give absolute expressions as literals. However, any relocatable expression is legal. For example:

```
           .NREL
           .
A:         .
           .
           LDA 2,=A
```

loads the value of "A" into index register 2. You can also use a literal to form a bytepointer to a text string labeled "TX":

```
           LDA 1,=2*TX)
           .
           .
           .
TX:        .TXT "TEXT STRING")
```

Literal labels permit communication with subroutines without concern for addressing errors. To call "SUB1"" (whether or not "SUB1" is directly addressable), the following creates a directly-addressable reference:

```
JSR @=SUB1
```

Generated Numbers and Symbols

The format:

\symbol

may appear anywhere in assembly code. When assembled, \symbol is replaced by a three-digit number, representing the current value of \symbol, truncated if necessary and in the current input radix.

\symbol may stand alone in the code to form an integer, or it may immediately follow characters that, together with the value of \symbol, will form a number or symbol. The number or symbol consists of any number of combined characters. For example:

```
;Source program:

.RDX 8
I = 1234
A\I     ;will become symbol A234
        ;(1 will be dropped).
BB\I    ;will become symbol BB234.
CCC\I   ;will become symbol CCC23.

450.\I ;will equal 450.234.


;Assembled listing:

000010 .RDX 8
001234 I = 1234
000000 A\I234       ;will become symbol A234
                    ;(1 will be dropped).
000000 BB\I234      ;will become symbol BB234.
000000 CCC\I234     ;will become symbol CCC23.

041434 450.\I234 ;will equal 450.234.
021676
```

\symbol will be printed in the assembly listing (but not in the cross-reference) even though it is suppressed in the generated relocatable code. For example,

Source Code:	**Listing:**
ONES=111	ONES=111
A\ONES:	A \ ONES111

\symbol can be incremented, using the .DO facility, to provide labels for a table. For example,

this source code:

```
          .RDX 8.
**        I=0
TABLE:    .DO 64.
A\I:      0
**        I=I+1
          .ENDC
```

assembles into this:

```
          .RDX 8.
TABLE:    .DO 64.
A\I000:   0
A\I001:   0
A\I002:   0
              .
              .
              .
A I077:   0
          .ENDC
```

Note that "\I", included in the labels "A\I000," etc., is not included in the actual symbol. It appears in the symbol table as "A000:".

End of Chapter

Chapter 6
How to Operate the Macroassembler

Assembler Files

Data General supplied the macroassembler along with your operating system in the following files:

Filename	Definition
MAC.SV	The assembler program.
MACXR.SV	Cross-reference file.

Often, during RDOS system generation, these files are placed in the master directory (the directory which holds the operating system). For DOS, you may need to LOAD these files, as well as the files listed under *microNOVA Systems* below, from your DG Utilities diskette.

Before you can use MAC, you must create a tailored version of the initial symbol table file, MAC.PS. You can do this with the assembler itself, by typing the appropriate command from the master directory.

For NOVA 3 systems:

MAC/S NBID, OSID, NSID, systype *[,PARU]* ⤶

For microNOVA systems:

MAC/S MBID, OSID, NSID *[,PARU,]* ⤶

For other NOVA systems:

MAC/S NBID, OSID, systype, *[,PARU]* ⤶

For ECLIPSE systems:

MAC/S NBID, OSID, NEID, *[NCID]* systype↑ ⤶
[,NFPID] [,PARU], ⤶

To find the systype, type LIST -DOS.SR⤶ from the master directory, and insert the name displayed. PARU is the system parameter file, which you'll need if you plan to use system mnemonics (like EREOF for end-of-file error). NFPID is the hardware floating-point instructions. Other files are defined later in this chapter. You can examine the contents of any of these files with the TYPE command. The names of all these source files end in *.SR;* thus PARU's full name is PARU.SR and OSID's full name is OSID.SR.

The proper MAC command above produces a tailored initial symbol table file, which the assembler will then access automatically for all future assemblies.

If you want to use the macroassembler from a different directory, you can create link entries to the assembler files in the master (or directory which holds the MAC files). For master directory Dxx, you'd type the following commands from the nonmaster directory:

```
LINK MAC.SV      Dxx:MAC.SV⤶
LINK MACXR.SV    Dxx:MACXR.SV⤶
LINK MAC.PS      Dxx:MAC.PS⤶
```

File LITMACS.SR

File LITMACS.SR, supplied with your system, contains a number of useful macros, already coded, for storing literals in NREL space. You may want to examine this file and copy the macros you like from it.

Operating Procedures

MAC can assemble source files input from a nondisk device, like a mag tape or card reader, but it works far more efficiently if the source files are on disk. The assembler cannot execute certain operators, like the iteration of .DO loops, from a nondisk file. We recommend that you code all your source files on disk with one of the text editor utilities, and that, if a file is not on disk, you transfer it to disk via the CLI command XFER or LOAD before trying to assemble it.

You invoke the macroassembler by typing the CLI command MAC, followed by one or more arguments. You can modify execution of a MAC command by inserting optional global switches, and modify an argument by inserting optional local switches.

The format of the Macroassembler command line is:

MAC *[globalswitches ...]* filename *[localswitch ...]*)

The MAC command line assembles one or more source files (filenames), and produces either an RB file, a listing file, or both. MAC assigns the extension .RB to filename; you must then process filename.RB with RLDR to make it executable.

Unless you specify otherwise with switches, the assembler output will receive the name of the first source program in the MAC command line; no listing will be produced, and assembly errors will be sent to the console.

You can include the following switches in the assembler command line:

Global Switches:

/A — Add semipermanent symbols to the cross-reference listing (used with global or local /L). By default these symbols are not included.

/E — Do not report assembly errors unless there is no listing file (global /L). In any case, error codes will always go to the console (unless you use both /E and /L). Error codes are described in Appendix A.

/F — Generate or suppress form feeds as required to produce an even number of assembly pages. This feature keeps the first page of successive listings on the outsides of paper folds, and makes refolding unnecessary. By default, a form feed is always generated at the end of a listing, whether the number of pages is odd or even.

/K — Keep the assembler's temporary symbol file (MAC.ST) at the end of the assembly. Since virtually no programs require the use of this file, it is deleted by default. Using the /L switch will give you a list of all symbols in this file used in the assembled program.

/L — Produce a listing of the assembly, including cross-reference. If you omit *local* /L, the listing will go to a disk file, named for the first source file in the command line, with the extension .LS.

/N — Produce no RB file. This switch is often used the first or second time that a file is assembled since there will probably be assembly errors (and the resulting binary would not be useful).

/M — Flag multiple-definition errors on pass one. Normally MAC flags these errors only if they remain at the end of pass two.

/O — Override all listing control pseudo-ops: .NOCON, .NOLOC, and .NOMAC. Also override the "**" listing suppression feature.

/S — Skip the second assembly pass (produce no .RB file) and save a version of the assembler's symbol table, MAC.PS. This procedure is described in detail below, under "Macroassembler Symbol Table Files".

/T Invoke the eight-character symbol feature. This instructs MAC to recognize symbol names of up to eight characters, and to store and output eight characters for each symbol name. Normally, MAC recognizes and stores only the first five characters of each symbol, although it prints longer symbol names in program listing. To allow for longer symbol names, the RB produced by a MAC/T command will be in *extended* RB format. If you plan to use global /T, be aware of the following restrictions when you write your source program:

1. No macro name can exceed five characters.
2. The first five characters of each macro name must differ from the first five characters of any other symbol name; e.g., .MACRO TEST1 and TEST100: MOVS 0,0 could not be used in one module. If MAC encounters such a symbol, it treats it as a macro call and tries to expand the symbol (e.g., it would try to expand TEST100 using the macro definition of TEST1.)
3. The cross-reference listing will show only the first five characters of each symbol name.

Eight-character symbols may also affect debugging, if you plan to use the debugger. See the Program Symbol Table section of the *Extended Relocatable Loaders* manual, for debugger restrictions associated with eight- character symbols.

4. A MAC.PS file created *without* a global /T switch will not work properly with files that need global /T. If you plan to use eight-character symbols in your sources, create a MAC.PS file specifically for assembly of these files. You'll use the MAC.PS created without global /T to assemble five-character symbol sources. See the local /T and the "Symbol File" sections of this chapter for more details.

If you omit global /T, MAC defaults to five-character symbol names and standard RB output.

/U Include local user symbols in the RB file. When the /U switch is also applied to the RLDR command line, then the debugger will be able to find local user symbols. This facilitates program debugging.

/Z For DGC personnel only: list the DGC proprietary license heading at the top of each assembly and cross-reference page. By default this heading is not listed.

Local Switches:

name/B Direct RB output to "name". Normally, the assembler places its output under the filename of the first source file in the MAC command line, with the RB extension, unless an RB module contains the .RB pseudo-op.

name/E Direct assembly errors to file "name", when a listing file has been specified.

name/L Direct assembly listing to file "name" (global /L is not required with this switch).

name/S Skip file "name" on the second pass of the assembly. File name must not define any storage words. Typical files that might be skipped include parameter definition files and macro definition files. Skipping such a file on the second assembly pass does not hinder the assembly of other files in the command line; it merely decreases the size of the output listing and reduces assembly time.

name/T This file holds the initial symbol table. If you omit "name/T", the assembler uses MAC.PS as the initial symbol table file.

Whether or not any filename in an assembly command line bears the source file extension, ".SR", the assembler will always search first for "name.SR" and only if this file cannot be found will the assembler search for filename without an extension. In every case, the assembler will name its output after the first source file in the command line (unless you specify /B, or insert the .RB pseudo-op). That is, the following commands,

MAC A B C) and MAC A.SR B.SR C)

each produce file A.RB. Error messages from these commands go to the console, and no listing file is produced. If, instead, the command was

```
MAC A B C $LPT/L
```

then A.RB would be produced as before, the assembly listing would go to the line printer, and error codes (and copies of the offending source code lines) would go to the console. You can give your source files any extensions you want (.SR is conventional), but avoid the extension .RB, because two identical filenames can't exist in the same directory.

You may not want a separate error file (local /E) since all assembler error messages consist simply of a letter code beside a bad line of source code and since all bad lines of source code are also flagged within an assembly listing.

The global /F switch is useful when you are performing a series of assemblies. For example:

```
MAC/F PROG<1,2,3,4> TFILE/L
```

assembles PROG1, PROG2, PROG3, and PROG4 (see the *CLI Reference Manual* for other uses of < >). TFILE will contain listings of the four files in the order they were assembled. The Macroassembler will insert either one or two form feeds between listings so that each separate listing starts on an even-numbered page. Before typing the command, position the paper in the printer so that the first page of TFILE will fold facing up. (The proper starting position will vary from one printer model to another.) This will make the first pages of the PROG2, PROG3, and PROG4 listings also fold facing up.

Macroassembler Symbol Table Files

The assembler maintains its symbol table and macro definition table in a disk file called MAC.PS. The symbol table is required to associate standard DGC machine instruction names (such as LDA) with their appropriate machine instructions (see the discussion of symbol table pseudo-ops in Chapter 4).

These machine instructions are provided in file NBID (NOVA Basic Instruction Definition). Additional instructions are contained in NSID.SR (NOVA Stack Instruction Definition, for NOVA 3 and microNOVA computers only), and NEID.SR (NOVA Extended Instruction Definition, for ECLIPSE machines). Additional instructions, for Commercial ECLIPSE computers, are defined in NCID.SR

Operating system call definitions are in file OSID.SR, and in another file whose name varies with your operating system (as described at the beginning of this chapter). The *macro definition* part of the table is used during assembly of your own macros.

At the start of each assembly, the permanent symbol table file, MAC.PS, is copied to create a temporary symbol file. Thus MAC.PS can be used to save symbol and macro definitions from assembly to assembly.

When the assembler detects the .XPNG pseudo-op, it deletes the symbol file and creates a new, empty symbol file. The /S function switch stops the assembler at the end of its first pass; the new symbol file then receives the name MAC.PS.

The requirements of MAC.PS will vary with your needs. If, for example, you will use operating system mnemonics for error codes, the system parameter definitions (found in PARU.SR) must be part of MAC.PS. You will probably find it convenient to build different versions of this file, and specify them at assembly time with the local /T switch. You could create such a file this way:

MAC/S sourcefile$_1$ ... sourcefile$_n$)

Then, RENAME the MAC.PS file to a useful name:

```
RENAME MAC.PS SYMBOLS3.PS)
```

Then, use the /T switch to specify your special file:

```
MAC FILEA SYMBOLS3.PS/T)
```

After each RENAME step, you'd need to recreate the original .PS file, as shown at the beginning of this chapter.

You can also change the original MAC.PS file by inserting global /S for a source file that begins with an .XPNG pseudo-op; or you could add to the retained symbols and macros by using global /S on a source file not containing .XPNG. These procedures allow you to define symbols or macros for one assembly and use their definitions for subsequent assemblies. You could also use .XPNG and the /S switch to assign new mnemonics to machine instructions (such as JUMP or some foreign language equivalent to the instruction that DGC names JMP). File NBID.SR, which is part of most MAC.PS files, starts with .XPNG.

Any MAC command has the following effects on the symbol table:

1. Copies MAC.PS into MAC.ST.

2. During pass 1, copies all user symbols, macros, and semipermanent symbols to MAC.ST (if it encountered .XPNG, it would have deleted the old table before adding these).

By including global /S, you effectively instruct the assembler to rename the MAC.ST to the MAC.PS; it will then use this MAC.PS file for subsequent assemblies unless you specify another file with the /T switch.

The symbol table portion of the symbol file can hold approximately 8,000 symbols. By default, MAC truncates symbols longer than five characters to five characters; if you include global /T, it truncates symbols to eight characters. It then stores the symbols in radix 50 representation to save space. Using smaller symbols will not increase the potential total beyond 8,000 symbols. The macro definition portion of the file can hold approximately 1/2 million characters or macro definition strings. Radix 50 representation and MAC's formats for different binary blocks are described in an appendix of the *Extended Relocatable Loaders* manual.

End of Chapter

Appendix A
Error Codes

MAC outputs two kinds of error messages. The first kind consist of one or more letter codes which MAC places beside a line of source code on the listing. These letter flags indicate assembly errors, which do not abort the MAC command. The second kind of error occurs when MAC cannot continue with the command (possibly because it lacks an essential file like MACXR.SV); this is a fatal error. This appendix describes first the alphabetical assembly error codes, then explains the fatal error messages.

Assembly Errors

Assembly error messages appear as single letter codes in the first three character positions of a listing line. The first error code appears in character position three of the line in which the error occurred. If there is a second error, the code is output in position two; for a third error, the code appears as the first character of the listing line.

Assembly errors are output as part of the assembly listing and to the console. If the listing is suppressed, the error listing is always output to the console.

The list of possible assembler error codes is as follows:

A	Address error.
B	Bad character.
C	Macro error.
D	Radix error.
E	Equivalence error.
F	Format error.
G	Global error.
I	Parity error on input.
K	Conditional or repetitive assembly error.
L	Location counter error.
M	Multiply-defined symbol error.
N	Number error.
O	Overflow error or stack error.
P	Phase error.
Q	Questionable line.
R	Relocation error.
U	Undefined symbol error.
V	Variable label error.
X	Text error.

Some typical causes of errors are given on the pages following. However, there is no way to pinpoint all the possible causes of assembly errors.

Addressing Error (A)

An A flags an error appearing in a memory reference instruction (MR) and indicates an illegal address. For example:

1. A page zero relocatable instruction references a normal relocatable (NREL) address.

   Example:

```
                  .NREL
    '000010 G:    10     ;NREL address.
                  .ZREL
 A  -040000       STA 0, G   ;ZREL
                             ;instruction.
```

2. An NREL address references an address outside the program location counter's relative address range: (.-200 < *displacement* < .+177).

   Example:

```
                  .NREL
 A   '020000      LDA  0, Y    ;Y is out
                               ;of MRI
                               ;instruction
     000420'      .LOC  .+416  ;range.
    '000002 Y:    2
```

Bad Character (B)

Error code B indicates an illegal character in some symbol. The line containing a symbol that has an erroneous character will be flagged with a B. A bad character error often leads to other errors.

Example:

```
                  .NREL
B00000' 024023   .A%: LDA 1,23     ;% IN LABEL
                                   ;SYMBOL
                                   CAUSES
                                   ;BAD
                                   CHARACTER
                                   ;ERROR.
```

Macro Error (C)

The macro error code C occurs under the following circumstances:

1. You attempted to continue the definition of a macro when it is not the last macro defined.

```
M        .MACRO A
         MOVZL 0, 0%
         .
         .
M        .MACRO A
         COM  1, 1%          ;Legal
         .                   ;continuation.
         .
         .MACRO B
         ADDZL 0, 0%
         .
         .
M        .MACRO A            ;Illegal to
         NEG 0, 0            ;continue any
                             ;macro but B.
                             ;Error goes to
                             ;all A defs.
```

2. If a macro exhausts assembler working space. However, this should only occur if the macro definition causes endless recursion.

3. If more than 63_{10} arguments are specified.

Radix Error (D)

Error code D occurs on a .RDX or .RDXO pseudo-op when .RDX contains an expression that is not in the range 2-20 or when .RDXO contains an expression that is not in the range 8-20, or when you used a digit that is not within the current input radix.

Examples:

```
D       000030   .RDX 4*6          ;Out of range.

        000002   .RDX 2
D       000013   B:      35        ;Not within
                                   ;current radix.
```

Equivalence Error (E)

Error code E occurs when an equivalence line contains an undefined symbol on the right-hand side of the equals sign. This may occur on pass one before the symbol on the righthand side has been defined or on pass two if the symbol is never defined.

Examples:

```
        .NREL
EUU     A=B        ;EUU on first pass.
UFU     A=B        ;UFU on second pass.
```

Format Error (F)

An F error results from any attempt to use a format that is not legal for the type of line and often occurs in conjunction with other errors.

When a format error occurs in an instruction, the code generated by the instruction reflects only those fields assembled before the error was detected.

Examples:

```
FFD           ADD  2             ;Not enough operands.

FD            STA 0, 10, 3, SNC  ;Too many operands
                                 ;and wrong operand for
                                 ;instruction type.

F             .ZREL-1            ;Pseudo-op does not
                                 ;allow argument.

    060612    .DUSR   C = DIAC 0, PTR
F   060612    C 1                ;Attempt to give argument
                                 ; to a symbol defined      L
                                 ; by .DUSR pseudo-op.

F   125005    MOV  1, 1, SNR     ;Instruction with skip
                                 ; field preceeds
FU 000000     ELDA 0, SYMB       ; two-word instruction.
```

Global Symbol Error (G)

A G error code results when there is an error in the declaration of an external or entry symbol.

Examples:

```
        .TITL Z
GU      .ENT FH          ;FH never defined.
          .
          .
        .END
```

Input (Parity) Error (I)

An I error code occurs when an input character does not have even parity. The assembler substitutes a back slash (\) for any incorrect charcter and flags the line containing the error with an I.

Conditional Assembly Error (K)

A K error code occurs when an .ENDC pseudo-op does not have a preceding .DO or .IF *x* pseudo-op.

Example:

```
        .DO   2
          .
          .
        .ENDC
K       .ENDC      ;No DO or IF for this ENDC.
```

Location Error (L)

The L code occurs when an error is detected in a line that affects the location counter.

Examples:

```
    L        177777          .LOC   -1
```

Here, the expression in a .LOC evaluates to less than zero or cannot be evaluated on the first pass of the assembler. If the expression is outside the range of locations or cannot be evaluated, the .LOC is ignored, and the location counter is unchanged.

```
    00436'000000 A:      0
  L        000443'       .BLK   .+100
```

Here, the expression in a .BLK statement cannot be evaluated on the first pass of the assembler or its value, when added to the current value of the program location counter, is less than zero. If an L error occurs, the .BLK statement is ignored and the location counter is unchanged.

Multiple Definition Error (M)

The M code flags a multiply-defined symbol. Within an assembly program a symbol appearing, for example, as a label cannot be redefined as another unique label. Any multiply-defined symbol will be flagged M at each appearance of the symbol.

Example:

```
            .NREL
M    A:     0        ;First pass.
PM   A:     0        ;Second pass.
```

Note that the second definition of A is also flagged as a phase error (P) on the second assembler pass. (See Phase Error.)

Number Error (N)

The N code is given when a number exceeds the proper storage limitations for the type of number; the N error occurs under the following conditions:

1. An integer is greater than or equal to 2^{16}. The number is evaluated modulo 2^{16}.

```
     000012       .RDX    10.
N    000003       65539
```

2. A double-precision integer is greater than or equal to 2^{32}. The number is evaluated modulo 2^{32}.

```
N    167153       40000000000.D
     024000
```

3. A floating point number is larger than $7.2*10^{75}$.

```
N    077777       7.3E75
     177777
```

Field Overflow Error (O)

A field overflow error occurs when:

- Variable stack space is exceeded; or
- When a .TOP or .POP is given with no previous .PUSH; or
- When an instruction operand is not within the required limits; e.g., 0-3 for an accumulator, 0-7 for a skip field, etc.

When overflow occurs in an instruction field, such as an accumulator field, the field will remain unchanged.

Examples:

```
O      000000      .POP

O      000000      .TOP

O      020775      LDA      5, .-3
```

Phase Error (P)

A phase error is caused when the assembler detects on pass 2 some unexpected difference from the source program scan on pass 1. For example, a symbol defined on the first pass which has a different value on the second pass will cause a phase error. If, as in the example, a symbol is multiply-defined, the M error flags each statement containing the symbol, while the phase error flags the second and any subsequent attempt to redefine the symbol.

Example:

```
M      B:     0      ;First pass.
PM     B:     1      ;Second pass.
```

Questionable Line (Q)

A Q error occurs when you have used a # or @ atom improperly, or a ZREL value where an absolute value is expected, or an instruction that may cause a skip immediately before a two-word instruction.

Examples:

```
        .ZREL
  FLD:  .BLK  10
        .NREL
G       LDA  0, FLD, 2   ;Assembler
                          ; expects ab-
                          ; solute for FLD.

G       MOV # 0, 1       ;No-load bit set
                          ; no skip field
                          ; given.
```

Relocation Error (R)

The R error indicates that an expression cannot be evaluated to a legal relocation type (absolute, relocatable, or byte relocatable as described in Chapter 3) or that the expression mixes ZREL and NREL symbols.

Example:

```
            .NREL
  000012 E: 10    ;Contents absolute.
  001130"   E+E   ; Contents NREL
                  ; byte relocatable.
R 001604'   E+E+E   ; Contents not
                    ; word or byte
                    ; relocatable.
```

Undefined Symbol Error (U)

The U error occurs on pass 2 when the assembler encounters a symbol whose value was never known on pass 1. The error occurs on pass 1 when the definition of a symbol (by equivalence) depends upon another symbol whose value is unknown at that point.

Example:

```
U       LDA 0, XX   ;XX was never defined.
```

See also the example given for equivalence error E.

Variable Label Error (V)

A V error occurs if anything other than a symbol follows the pseudo-op .GOTO.

Example:

```
FV            .GOTO 14
```

Text Error (X)

An error occurring in a string is flagged as a text error (X). A text error occurs if the expression delimiters < and > within a string do not enclose a recognizable arithmetic or logical expression. (Relational expressions cannot be used within text strings.)

Examples:

```
X  037131   .TXT "<X+  Y>" ;No spaces are
   000000
                           ; allowed in
                           ; expressions.

X  000000   .TXT "<+>"   ;Expressions must
                         ; have operands.

X  037067   .TXT "<X=>Y>"   ;Relational
   037131
   000000
                            ; operators
                            ; are not
                            ; allowed.
```

Fatal Errors

The following error messages abort the MAC command and return control to the CLI.

ATTEMPT TO POP LINKED ELEMENT WHEN NONE EXISTS

This results from an internal MAC error. An internal stack containing linkage information has become too small. When it tried to pop a link from the stack, MAC couldn't find any link. This error produces a BREAK.SV file (FBREAK.SV if MAC was running in the foreground). Please send a copy of your command line, source file(s), and (F)BREAK.SV to your local Data General Software Engineer.

COMMAND FILE ERROR

The CLI's command file (COM.CM or FCOM.CM), which it uses to communicate with MAC, has the wrong format. See the CLI manual for (F)COM.CM formats.

INSUFFICIENT MEMORY

The minimum configuration of MAC is too large to run in available memory.

LINKAGE STACK OVERFLOW

An internal stack containing linkage information has become too large for its allotted memory space.

MACRO DEFINITION OVERFLOW

Macro definitions have overflowed the maximum addressable size of the MAC.ST file. This is approximately a half-million bytes; see Chapter 6.

MACXR.SV DOES NOT EXIST

File MACXR.SV generates the cross-reference listing; it is required. You can either MOVE it to or LINK to it from the current directory (see the beginning of Chapter 6).

SYMBOL TABLE OVERFLOW

MAC can handle a maximum of about 8,000 symbols. If the file(s) in your command contain more than this number, you receive this message. Try assembling files in smaller groups.

End of Appendix

Appendix B
Permanent Symbols

Permanent Symbol	Pseudo-op (directive)	Value
.ARGT	No	Yes
.BLK	Yes	No
.COMM	Yes	No
.CSIZ	Yes	No
.DALC	Yes	No
.DCMR	Yes	No
.DEMR	Yes	No
.DERA	Yes	No
.DEUR	Yes	No
.DFLM	Yes	No
.DFLS	Yes	No
.DIAC	Yes	No
.DICD	Yes	No
.DIMM	Yes	No
.DIO	Yes	No
.DIOA	Yes	No
.DISD	Yes	No
.DISS	Yes	No
.DMR	Yes	No
.DMRA	Yes	No
.DO	Yes	No
.DUSR	Yes	No
.DXOP	Yes	No
.EJEC	Yes	No
.END	Yes	No
.ENDC	Yes	No
.ENT	Yes	No
.ENTO	Yes	No
.EOF	Yes	No
.EXTD	Yes	No
.EXTN	Yes	No
.EXTU	Yes	No
.GADD	Yes	No
.GLOC	Yes	No

Permanent Symbol	Pseudo-op (directive)	Value
.GOTO	Yes	No
.GREF	Yes	No
.IFE	Yes	No
.IFG	Yes	No
.IFL	Yes	No
.IFN	Yes	No
.LMIT	Yes	No
.LOC	Yes	Yes
.MCALL	No	Yes
.MACRO	Yes	No
.NOCON	Yes	Yes
.NOLOC	Yes	Yes
.NOMAC	Yes	Yes
.NREL	Yes	No
.PASS	No	Yes
.POP	Yes	Yes
.PUSH	Yes	No
.RB	Yes	No
.RDX	Yes	Yes
.RDXO	Yes	Yes
.REV	Yes	No
.TITL	Yes	No
.TOP	No	Yes
.TXT	Yes	No
.TXTE	Yes	No
.TXTF	Yes	No
.TXTM	Yes	Yes
.TXTN	Yes	Yes
.TXTO	Yes	No
.XPNG	Yes	No
.ZREL	Yes	No

End of Appendix

Appendix C
ASCII Character Subset

7-Bit ASCII Code	Character	7-Bit ASCII Code	Character	7-Bit ASCII Code	Character	7-Bit ASCII Code	Character	7-Bit ASCII Code	Character
011	HT	061	1	104	D	127	W	153	k
014	FF	062	2	105	E	130	X	154	l
015	CR	063	3	106	F	131	Y	155	m
040	SP	064	4	107	G	132	Z	156	n
041	!	065	5	110	H	133	[	157	o
042	"	066	6	111	I	134	\	160	p
043	#	067	7	112	J	135	]	161	q
045	%	070	8	113	K	136	↑	162	r
046	&	071	9	114	L	137	←	163	s
047	'	072	:	115	M	141	a	164	t
050	(	073	;	116	N	142	b	165	u
051	)	074	<	117	O	143	c	166	v
052	*	075	=	120	P	144	d	167	w
053	+	076	>	121	Q	145	e	170	x
054	,	077	?	122	R	146	f	171	y
055	-	100	@	123	S	147	g	172	z
056	.	101	A	124	T	150	h		
057	/	102	B	125	U	151	i		
060	0	103	C	126	V	152	j		

End of Appendix

Index

Within this index, the letter "f" following a page number means "and the following page"; "ff" means "and the following pages". *Italics* indicate the primary page reference (where applicable). Entries in CAPITAL letters are generally assembler pseudo-ops (if they begin with a period, e.g., .ARGCT), instructions (e.g., ADC), or filenames (e.g., MAC.PS).

) (CR) 1-2, 2-2, 5-5
□ (Space) 2-2, 5-3
! (OR, inclusive) 3-1f
" (quote) 2-4f
(number sign) 2-7
$ (generate label) 5-15
% (macro definition terminator) 5-1
& (AND) 3-1f
' (apostrophe) 2-5
( ) (parentheses) 2-2
* (multiply) 3-1f
** (asterisks) 2-8
+ (plus) 3-1f
, (comma) 2-2, *5-3*
. (current location) 4-5
. (decimal input) 2-5f
- (minus) 3-1f
/ (divide) 3-1f
: (colon) 2-2
; (semicolon) 2-1f
< > (relational operators) 3-1f
= (equals) 2-2
@ (commercial AT) 2-7
[ ] (brackets) 2-2, *5-3*
\ (backslash-generate a number or symbol) 5-17
↑ (dummy argument in macro definition) 5-1ff, 5-5
← (backarrow) 2-4ff, 5-1

A error 3-11f, *A-2*
absolute code 1-2, 1-6
ADC 3-7
ADD 3-7
address
  assembler, loader operations 1-6
  forming effective 3-11, 3-14f
  indirect 2-7
ALC instructions 3-7
AND
  & 3-1f
  instructions 3-7
.ARGCT 4-4, *4-6*
arguments
  to macros 1-1, 2-2, 5-1ff
  to pseudo-ops, see pseudo-ops
arithmetic operators 2-2, 3-1f
ASCII character subset C-1
assembler, see macroassembler
asterisks (**) 2-8
atom (special character)
  definition 1-2

B, see bit alignment
bit alignment (B) 3-1f
.BLK 4-6
break characters 2-2
busy/done mnemonics 3-8f

carriage return ()) 1-2, 2-2, 5-5
case of characters (upper/lower) 2-1
code, absolute 1-2, 1-6
COM 3-7
.COMM 1-4, *4-7*
.COMM TASK 4-7
comments 2-1f
commercial at sign (@) 2-7
commercial instructions 3-15, 6-4
communication between modules 4-3
conditionals 4-4, 5-6f
  also see .DO, .IFE, etc.
counter, location 1-3f, *1-6*, 4-3, 4-5, *4-29*
cross-reference listing 1-4f
.CSIZ 4-8

data field 1-3
.DALC 4-9
.DCMR 4-10
defining macros 4-29, 5-1ff
.DEMR 4-11
.DERA 4-12
.DEUR 4-13
.DFLM 4-13
.DFLS 4-14
DIA, etc. 3-9
.DIMM 4-15
.DIO 4-16
.DIOA 4-16
.DICD 4-15
.DISD 4-17
.DISS 4-17
.DMR 4-18
.DMRA 4-18
.DO 4-4, *4-9*, 4-27
.DO loops in macros 5-6f
DOA, etc. 3-9
documentation conventions iv
double-precision numbers 2-5f
DSZ 3-11ff
.DUSR 4-19
.DXOP 4-20

E (exponent, floating-point) 2-6
E-series ECLIPSE instructions 3-14f
eight-character symbols 6-2
.EJEC 4-20
.END 4-21
.ENDC
  in macro 5-6f
  pseudo-op 4-21
.ENT 1-4, 4-3, *4-22*
.ENTO 1-4, *4-22*
.EOF, .EOT 4-23
error
  assembly A-1 to A-5
  fatal A-6
  field in listing 1-3f
expressions
  general 3-1f
  relocation of 3-3f
.EXTD 1-4, 3-11, *4-23*
extended RBs 6-2
.EXTN 1-4, 4-3, *4-24*
.EXTD 4-24

F error A-3
fields in source code 1-3
files
  instruction definition 6-4
  macroassembler 6-1, 6-4f
floating-point
  constants 2-6f
  instructions (ECLIPSE) 3-15f
form feed 2-2

G error A-3
.GADD 4-25
generated
  labels 5-15
  numbers/symbols 5-17
.GLOC 4-25
.GOTO 4-26
.GREF 4-26

HALT 3-10
HALTA 3-10

I/O instructions
  with ac 3-9
  without ac 3-8
  without arguments 3-10
  without device code 3-10
.IFE, .IFG, .IFL, .IFN 4-27
INC 3-7
indexed addressing, 3-11f, 3-14f
indirect addressing 2-7
input to assembler
  ASCII subset C-1
  break characters 2-2
  comments 2-1
  macros, see macro normal code 2-1
  numbers 2-2 to 2-7
  strings 2-1
instructions
  ALC 3-7
  by type 3-6
  I/O, see I/O instructions
INTA 3-10
INTDS 3-10
integer storage 2-2ff
INTEN 3-10
IORST 3-10
ISZ 3-11ff

JMP 3-11ff
JSR 3-11ff

labels, generated 5-15
LDA 3-13
LFE (library file editor) iii
listing
  fields 1-3f
  pseudo-ops 4-5
literals 5-16
LITMACS.SR 6-1
.LMIT 4-5, *4-28*
.LOC 1-2 *4-29*
location counter 1-3f, *1-6*, 4-3, 4-5, *4-29*
logical operators 2-2, *3-1f*
lowercase characters 2-1

macro
  calls 2-2, 5-3ff
  definitions 5-1f
  .DO loops in 5-6f
  examples
    FACT 5-9
    PACK 5-10f
    OR 5-8
    VFD, ERROR, SPECL 5-12ff
  expansions 5-2, 5-5ff
  overview 1-1
  pseudo-ops 4-4
.MACRO 2-1, *4-29*
macroassembler
  errors, see error
  files 6-1
  input, see input to assembler
  language 1-1
  listing 1-3f
  macro overview 1-1, 5-1ff
  operating Chapter 6
  output 1-1f, also see RB
  switches 6-2f
  symbol
    capacity 6-5
    file, see MAC.PS
    table file 6-4f
MAC.PS
  building 4-39, *6-1*, 6-4
  /T switches 6-2
MAC.ST 6-5
.MCALL *4-30*
memory reference instructions 3-11 to 3-16
  commercial (ECLIPSE) 3-15f
  extended (ECLIPSE) 3-14f
  with ac 3-13ff
  without ac 3-11
mnemonics, busy/done 3-8f
modules, communication between 4-3
MOV 3-7
MSKO 3-10

NBID.SR 4-39, 6-1, 6-4
NCID.SR 6-4
NEG 3-7
NEID.SR 6-4
NIO 3-8
.NOCON 4-30
no-load instructions 2-7
.NOLOC 4-31
.NOMAC *4-31*, 5-6
normal-relocatable (NREL) code 1-2, 1-6
.NREL 1-2, *4-32*
NSID.SR 6-4
number sign (#) 2-7

numbers
  double-precision 2-5f
  general 2-2 to 2-7
  generated 5-17
  operators 2-2
  single-precision 2-2ff

operators 2-2, 3-1f
OSID.SR 6-4
OR 3-1f
organization of manual iii
output of assembler 1-1f
  also see RB

page zero (ZREL) code 1-2, 1-6
  conserving 5-16
  specifying 4-39
parity 1-2, 4-4
PARU.SR 6-4
.PASS 4-32
permanent symbols, summary C-1
.POP 4-33
program listings 1-3f
pseudo-ops Chapter 4
  alphabetically 4-5 to 4-39
  by category 4-1 to 4-5
  purpose of 3-5f
  summary B-1
.PUSH 4-29, *4-33*

radix 2-4, also see .RDX
RB
  extended, see extended RB
  format 6-5
  overview 1-1f
.RB 4-33
.RDX 2-4, 4-34
.RDXO 4-35
READS 3-10
relational operators 2-2, *3-1f*
relocatable binary see RB
relocation
  counters 1-6
  expressions 3-3f
  flags 1-3ff
  overview 1-6
  pseudo-ops 4-32, 4-39
removing symbols 4-39
related manuals iii
.REV 4-35
RLDR 1-6

single-precision numbers 2-2ff
skip (ALC instructions) 3-7
SKPBN, etc. 3-8
source files for definitions 6-4
space (□) 2-2, 5-3
special characters (@, #, **) 2-7f
STA 3-13
stack pseudo-ops 4-4, 4-33, 4-36
storage, integer 2-2ff
SUB 3-7
switches (MAC) 6-2f
symbols
  capacity 6-5
  eight-character 6-2
  file, see MAC.PS
  format of 2-7
  generated by MAC 5-17
  global 3-5
  local 3-5
  permanent *3-4f*, B-1
  removing 4-39
  semipermanent 3-5
  storage 6-5
  user, see user symbols
  value 4-1
symbol table pseudo-ops 4-1f

/T switches (MAC.PS) 6-2
tab 2-2, *5-3*
terminating a line, see break characters
text strings 2-1, 4-4
.TITL 4-36
.TOP 4-36
.TXT, .TXTE, etc. 4-37
.TXTM 4-38
.TXTN 4-38

U error A-5
uppercase (characters) 2-1
user symbols
  defining semipermanent, see pseudo-ops
  format of 2-7
  length 6-2
  overview 3-5

value symbols 4-1

.XPNG 4-39

.ZREL 1-2, *4-39*

DataGeneral

Software Documentation Remarks Form

How Do You Like This Manual?

Title ______________________________ No. ____________

We wrote the book for you, and naturally we had to make certain assumptions about who you are and how you would use it. Your comments will help us correct our assumptions and improve our manuals. Please take a few minutes to respond.

If you have any comments on the software itself, please contact your Data General representative. If you wish to order manuals, consult the Publications Catalog (012-330).

Who Are You?

- ☐ EDP Manager
- ☐ Senior System Analyst
- ☐ Analyst/Programmer
- ☐ Operator
- ☐ Other ____________

What programming language(s) do you use? ____________

How Do You Use This Manual?

(List in order: 1 = Primary use)

- ____ Introduction to the product
- ____ Reference
- ____ Tutorial Text
- ____ Operating Guide
- ____ ____________

Do You Like The Manual?

Yes	Somewhat	No	
☐	☐	☐	Is the manual easy to read?
☐	☐	☐	Is it easy to understand?
☐	☐	☐	Is the topic order easy to follow?
☐	☐	☐	Is the technical information accurate?
☐	☐	☐	Can you easily find what you want?
☐	☐	☐	Do the illustrations help you?
☐	☐	☐	Does the manual tell you everything you need to know?

Comments?

(Please note page number and paragraph where applicable.)

From:

Name ____________ Title ____________ Company ____________

Address ______________________________ Date ____________

FOLD DOWN FIRST FOLD DOWN

FIRST
CLASS
PERMIT
No. 26
Southboro
Mass. 01772

BUSINESS REPLY MAIL

No Postage Necessary if Mailed in the United States

Postage will be paid by:

Data General Corporation

Southboro, Massachusetts 01772

ATTENTION: Software Documentation

FOLD UP SECOND FOLD UP

SD-00742A

STAPLE