Data General

Software Documentation

RDOS/DOS Assembly Language and Program Utilities

RDOS/DOS Assembly Language and Program Utilities

069-400019-01

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. 069-400019

Revision 01, March 1984

NOTICE

RDOS/DOS
Assembly Language and
Program Utilities
069-400019

Revision History: Effective with:

Original Release - October 1983

First Revision - March 1984

CONTENT UNCHANGED

The content in this revision is unchanged from 069-400019-00. This revision changes only printing and binding details.

Preface

This manual consists of three parts each containing information about different programming utilities that run on the Real-time Disk Operating System (RDOS) and Disk Operating System (DOS). You should be familiar with assembly language.

* Part I describes the Extended Assembler (ASM) and the Macroassembler (MAC). These utilities process source files written in assembly language and produce relocatable binary files, called RBs by convention. After assembling a source file, you can generate an RB library with the Library File Editor (LFE).

* Part II of this manual explains the Library File Editor, which provides a means of updating and interpreting library files.

* Part III describes the Extended Relocatable Loader (RLDR). This utility processes and links any number of files and libraries to produce an executable save file. RLDR processes binary files produced by the Data General assemblers and libraries produced by the Library File Editor utility.

Manual Organization

We have organized this manual as follows:

Chapter 1 introduces the Assemblers. This chapter contains information on Assembler input and output and the macro facility within the Macroassembler (MAC).

Chapter 2 explains the contents and makeup of the source programs that are read by the assemblers. Each source program must conform to a given format containing characters within the assemblers' character set.

Chapter 3 describes the fundamental assembly tools used to input characters. The basic unit of assembly language, called an atom, is explained along with each of the five classes into which an atom may fall.

Chapter 4 describes an expression, which is made up of operators, symbols, and instructions. There are many different types of symbols and instructions, all of which are explained in this chapter.

Chapter 5 defines all pseudo-ops and value symbols. The pseudo-ops are described in categories, followed by a complete alphabetical pseudo-op dictionary

Chapter 6 explains the advanced features of MAC.

Chapter 7 describes the operating procedures of the Extended Assembler (ASM).

Chapter 8 describes the operating procedures of the Macroassembler (MAC).

Chapter 9 introduces the Library File Editor utility (LFE). This chapter contains general information about LFE and some terminology pertaining to this utility.

Chapter 10 describes the operation of the Library File Editor (LFE) and the commands used with it. The format of the LFE commands is also described.

Chapter 11 introduces and describes the RDOS/DOS Relocatable Loader. The command line is explained in this chapter, along with the different files and tables used with this utility.

Chapter 12 explains the RDOS Overlay Loader. It describes the command line as well as switches and includes a few examples.

Appendix A lists all valid pseudo-ops for the Extended Assembler and the Macroassembler.

Appendix B lists and describes the error messages for each utility.

Appendix C lists all ASCII characters along with their ASCII codes.

Appendix D describes the Relocatable Binary Block Format.

Related Manuals

The following manuals are a part of the series of books published on RDOS and DOS.

Introduction to RDOS (DGC No. 069-400011) describes the fundamentals of using RDOS and summarizes the features, utilities, and capabilities of the operating system.

Guide to RDOS Documentation (DGC No. 069-400012) describes all of the books that comprise the revised documentation set for RDOS, DOS, and RTOS and lists the previous books that each replaces.

How to Load and Generate RDOS (DGC No. 069-400013) explains how to load, generate, and maintain RDOS. Instructions cover preparing hardware, program loading, disk initialization, bootstrapping, tailoring, system backup, and system tuning.

How to Generate Your DOS System (DGC No. 093-000222) explains how to generate and maintain DOS. Instructions cover preparation of hardware, disk initialization, bootstrapping, tailoring, system backup, patching, and optimization.

RDOS/DOS Command Line Interpreter (DGC No. 069-400015) describes the user interface with the operating system. It covers the Command Line Interpreter (CLI) features and command mechanisms, and instructs you on how to use CLI commands. Features and operation of the Batch monitor are also presented.

RDOS/DOS Text Editor (DGC No. 069-400016) documents how to load, use, and operate the Text Editor (Edit) or Multi-user Text Editor (Medit) to create and edit text.

RDOS/DOS Superedit Text Editor (DGC No. 069-400017) introduces the commands and concepts of the Superedit Text Editor (Speed), which offers more powerful features for editing text.

RDOS System Reference (DGC No. 093-400027) describes RDOS system features, calls, and user device driver implementation for assembly language and high-level language programming.

DOS Reference Manual (DGC No. 093-000201) discusses all features of DOS for system programmers, including system control, memory management, and system calls.

RDOS/DOS User's Handbook (DGC No. 093-000105) summarizes the commands, calls, and error messages of the RDOS/DOS CLI, Batch monitor, and utility programs.

RDOS/DOS Debugging Utilities (DGC No. 069-400020) describes five utilities that assist you in editing, debugging, and patching programs—the Symbolic Editor (SEDIT), Symbolic Debugger (DEBUG), Disk Editor (DISKEDIT), and Patch (ENPAT and PATCH).

RDOS/DOS Sort/Merge and Vertical Format Unit Utilities (DGC No. 069-400021) offers functional information on Sort/Merge (RDOSSORT), which helps you manipulate records in data files, and the Vertical Format Unit (VFU), which enables you to define data channel line printer page formatting.

RDOS/DOS Backup Utilities (DGC No. 069-400022) presents the features and operation of the utilities that involve disk and tape backup. These are BURST/TBURST, DBURST/MBURST/RBURST, DDUMP/DLOAD, FDUMP/FLOAD, and OWNER.

Typesetting Conventions

We use these conventions for command formats in this manual:

COMMAND *required* [*optional*] . . .

Where	Means
COMMAND	Enter the command (or its accepted abbreviation) as shown. Upper-case letters indicate the command mnemonic.
required	Enter some argument (such as a filename) Sometimes, we use. *required*$_1$ | *required*$_2$ You can choose between the arguments listed Do not use the vertical bar; it merely separates the choices. Lower-case italic letters indicate an argument.
[*optional*]	Brackets mean that you have the option of entering the argument. (Command switches also appear in this format.) Do not include the brackets in your code; they only set off the choices.
. . .	Repeat the preceding entry or entries. The explanation will tell you exactly what to repeat.
. . .	The process has continued without incident, and you may now take the next action described.

Additionally, we use certain symbols in special ways:

Symbol	Means
⟨CR⟩	Press the RETURN key on your terminal.
□	Include a space at this point. (We use this to clarify in some cases. Normally, you can see where you should put spaces.)
⟨NL⟩	New Line. This convention appears in programs that are reproduced as examples.

All numbers are decimal unless we indicate otherwise; for example, 35_8.

The keys defined as DEL and RUBOUT perform the same function. Depending on the console you are using, you will find one of these keys on your keyboard In this manual, we use only DEL for that function.

The up arrow symbol (↑) is also executed by different keys, depending on your console. It can be executed by pressing either SHIFT-N or SHIFT-6. In this manual we reference SHIFT-6 to execute the up arrow symbol.

In examples of dialogue, we use.

THIS TYPEFACE TO SHOW YOUR ENTRY

THIS TYPEFACE FOR SYSTEM RESPONSES

R is the RDOS/DOS Command Line Interpreter prompt.

We welcome your suggestions for the improvement of this and other Data General publications. To communicate with us, use the postpaid comment form at the end of this manual.

Table of Contents

Figures

Tables

Part I

Assemblers

Chapter 1
Introduction to Assemblers

A language is a set of common characters and conventions that convey information in a well-defined way. Computer languages range from those tailored to computer hardware operations to those that 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 combinations of letters and numbers like English.

Assemblers process source programs written in assembly language, an ASCII character subset, to produce object files in machine language. To do this, the assembler substitutes a numeric code for each symbolic instruction code, and a numeric address for each symbolic address. Each line of symbolic instruction is translated into one line of numeric instruction by the assembler. Assembly language provides mathematical and logical operations for symbol manipulation, as well as a macro facility that permits your own character sequences to be expanded into different forms by the assemblers.

There are two assemblers available with the Real-time Disk Operating System (RDOS): the Extended Assembler (ASM) and the Macroassembler (MAC). Both are similar in their operation and are discussed together in this manual. Any differences are noted.

Assembly Language Processing

The assembler translates symbolic instruction codes (such as LDA 0,2) and symbolic addresses (such as LDA O,TEMP) into numeric codes and numeric addresses. The addresses may be either absolute (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 (RB) file or module. The computer cannot execute a source file (it is symbolic), nor can it execute a binary file directly. Each must be further processed to prepare it for execution. This process, called loading, turns a binary file into a save file that 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 it is needed during assembly of a source file.

A macro facility works as follows:

1. You write a set of symbolic instructions, called a macro definition, and give it 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 only in 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.

Intermodule Communication

Intermodule (interprogram) communication facilities define data addresses and constants in one program and reference them in another. By using the interprogram communication pseudo-ops, programmers can write related subprograms without concern for the absolute locations of data and addresses shared by these programs at run time. See Chapter 5 for more information on intermodule communication.

Conditional Assembly

Conditional assembly facilities provide you with the capability to include portions of source code in the assembly process. Depending on the evaluation of an absolute expression, the assemblers either assemble or bypass a section of source code. The pseudo-ops necessary for controlling the conditional assembly are explained in Chapter 5.

Assembler Input and Output

An assembler accepts as input one or more source files written in assembly language. The output listing may contain any source program errors, as well as listings of the source program itself and the relocatable binary file. All of this output is optional. The program and error listings give you information, while the binary file is processed by the Relocatable Loader (RLDR) to make it executable. Figure 1.1 illustrates the input to and possible output from the Assemblers.

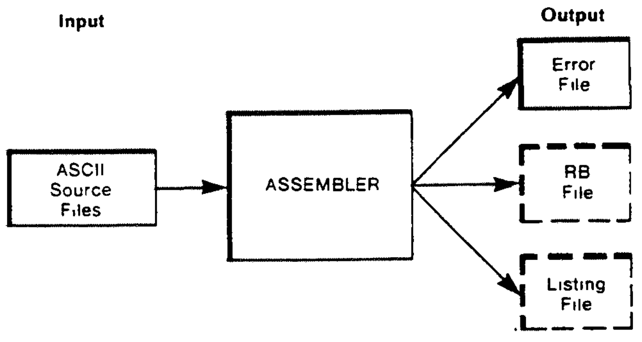


Figure 1.1 Assembler input and output SD-00420

Assembler Input

The source program input to the assembler consists of characters which are a subset of the ASCII character set. The assembler reads this input line-by-line and translates it into machine language binary code. 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 (⟨CR⟩) or a Form Feed.

2. The assembler ignores the null, Line Feed and RUBOUT characters.

3. It replaces characters that have incorrect parity with a 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 source program listing file.

3. An error listing.

Relocatable Binary File

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 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 the relocatable loader. The assembler gives each number an address, which may be a relative address that RLDR will relocate. The assembler also produces as part of the binary file the information which RLDR needs to map each address and its contents.

A relocatable binary file can have three different sections of code: absolute, page zero relocatable (ZREL), and normal relocatable (NREL) 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.

You may choose not to output a binary file.

Source Program Listing

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

Columns 1-3	Contain a two-digit line number followed by a blank space if the assembler finds no errors in the input. 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 through 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.

An error flag is a single letter indicating the type of error that appeared in the source line. For example, a parity error on input would produce the flag I in column 3 of the program listing line. Up to three error flags may appear on a line.

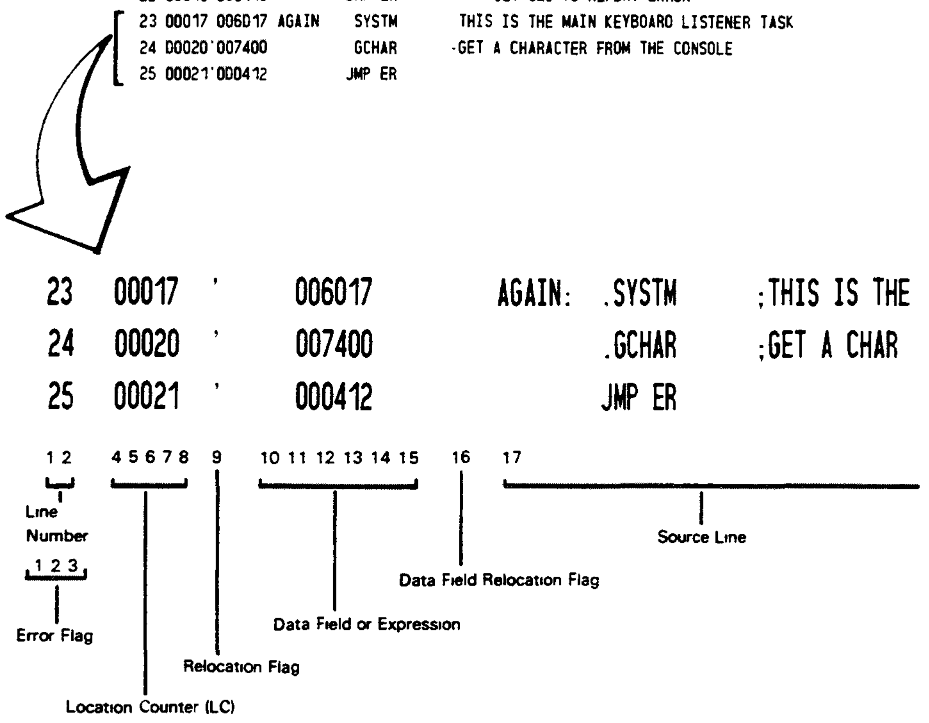
The 5-digit location counter (LC) assigned by the assembler to an instruction is displayed in columns 4 through 8. The LC is immediately followed by a single-character flag indicating the relocation mode of the address:

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

```
0001 EXAMP MACRO REV 06 00              15 11 42 07/26/77
                         TITL EXAMPLE
02                       NREL
03      000001           TXTM 1          PACK  TXT BYTES LEFT-TO RIGHT
04                       ENT START ER TASK1  AGAIN    DEFINED HERE
05                       EXTN  TASK  PRI  TOVLO   GET MULTITASK HANDLERS
06
07 00000'006017 START    SYSTM           SYSTEM  GET A FREE
08 00001 021052          GCHN            CHANNEL NUMBER  PUT IN AC2
09 00002'000776         JMP START       .ON ERROR  TRY AGAIN
10 00003'050427         STA 2  CHNUM    .STORE CHANNEL NUMBER IN "CHNUM"
11 00004'020433         LDA 0  NTTO      POINTER TO CONSOLE OUTPUT NAME
12 00005'126400         SUB 1  1         USE DEFAULT DISABLE MASK
13 00006'006017          SYSTM           SYSTEM  OPEN CONSOLE OUTPUT
14 00007'014077          OPEN 77         ON CHANNEL NUMBER IN AC2
15 00010'000423         JMP ER           ON ERROR  GET CLI TO REPORT
16 00011 020432         LDA 0  P4        GET NUMBER "4"
17 00012'077777          PRI             CHANGE YOUR PRIORITY TO 4
18 00013 020431         LDA 0  IOPRI     GET NEW TASK S ID AND PRIORITY
19 00014 024431         LDA 1  TASK1     START NEW TASK AT THIS ADDRESS
20 00015 077777          TASK            CREATE NEW TASK  WHICH GAINS CONTROL
21     ,                                 IMMEDIATELY  SINCE ITS PRIORITY IS 3
22 00016'000415         JMP ER           GET CLI TO REPORT ERROR
23 00017 006017 AGAIN    SYSTM       THIS IS THE MAIN KEYBOARD LISTENER TASK
24 00020'007400          GCHAR      -GET A CHARACTER FROM THE CONSOLE
25 00021'000412         JMP ER
```



```
23  00017  '   006017       AGAIN:  .SYSTM     ;THIS IS THE
24  00020  '   007400               .GCHAR     ;GET A CHAR
25  00021  '   000412                JMP ER
```

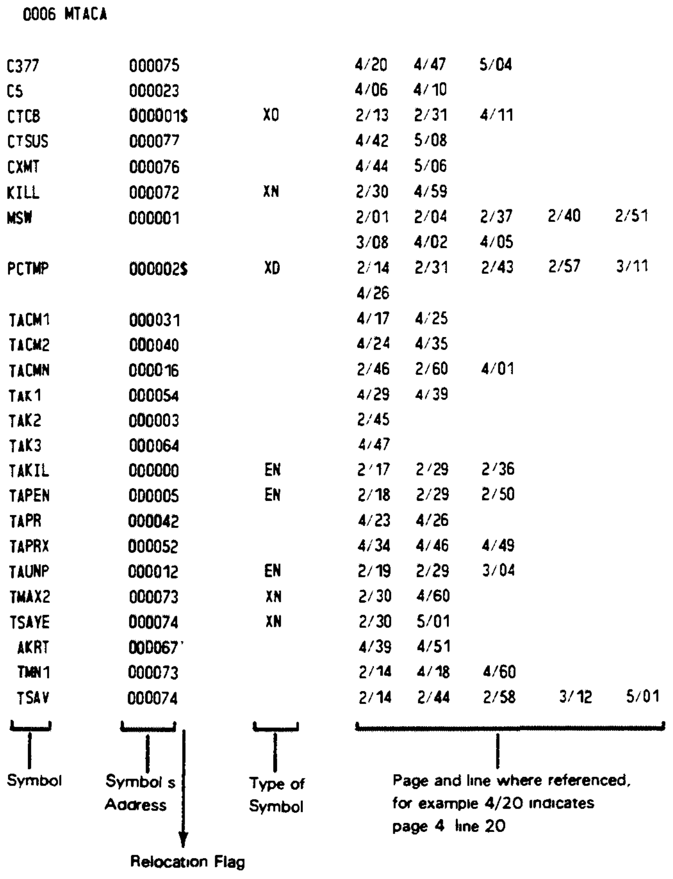
Figure 1.2 Program listing

DG-025168

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.

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



```
 0006 MTACA

C377        000075               4/20  4/47  5/04
CS          000023               4/06  4/10
CTCB        000001$      XO      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
TSAYE       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
Type of Symbol
Page and line where referenced, for example 4/20 indicates page 4 line 20
Relocation Flag

Figure 1.3 Cross-reference listing DG-25169

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 program and all source lines that have been flagged with an error code. If a source program listing is output, the error listing is optional. During the second pass, all errors in the source program are included as part of the program listing. If a program listing is not output, a separate error listing is produced automatically. The error listing output on the second pass lists only those lines containing errors. The format of the lines is the same as that of the source program listing 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

Relocatability

Both MAC and ASM are relocatable assemblers, which means that they assign to each storage word a relative location counter value RLDR takes each relative value and gives it an absolute memory address

The assemblers produce output that is placed by RLDR for execution in either the absolute, the ZREL, or the NREL sections of memory

The assemblers assign relative values by using three counters that they maintain for each type of relocatability one each for absolute, ZREL, and NREL code. The ZREL and NREL counters are initially zero and are increased by one for every storage word generated When a program has been completely assembled, it has used *x* ZREL words and *y* NREL words These words have been assigned relative addresses ZREL 0 to (*x*-1) and NREL 0 to (*y*-1)

The role of RLDR 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, using its own counters Like the assemblers, RLDR maintains three counters (for absolute, ZREL, and NREL code). Using these counters, RLDR establishes an absolute address for each relative address in the modules it processes. It initializes the ZREL counter to 50_8 and initializes the NREL counter(s) according to the length of the UST, the number of tasks specified, and the size and number of overlay nodes in the program. RLDR assigns to 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 the action of RLDR 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 information about tasks, system tables or overlays, consult the RDOS or DOS system reference.

Operation

The relocatable loader tape is in binary format and is loaded using the binary loader Once loaded, the relocatable loader will self-start and type.

SAFE=

on the terminal. This queries the user about the octal number of words to be preserved at the high end of memory.

The default response is a Carriage Return, which will cause the loader to save 200 octal words – enough to preserve both bootstrap and binary loaders.

Otherwise, the user may input an octal number, terminated by a Carriage Return, giving the number of octal words to be saved. The user response may be up to five octal digits long, and must be within the limits of memory.

An error on input will cause the loader to repeat the query, SAFE=. The error cases are

1. A character other than an octal digit or a Carriage Return is input.
2. More than five octal digits are input
3. The number specified is too large for the user's core configuration; that is, no load space remains.

When the SAFE= query has been correctly answered, the value specified is fixed for the duration of the loading process. The loader then prompts a user response by displaying an asterisk (*) on the terminal.

Relocatable loader action is initiated by single-digit responses to the loader-prompt, *. The possible loader-mode responses are tabulated in Table 1.1 and are described throughout this manual. Each time the loader has completed its response to a user request, it will type an asterisk and await a new request. If you enter an illegal response the loader prints

?

and awaits a legal response. You can end user-loader dialogue by responding to a prompt with the digit 8 (terminate loading process and prepare for execution)

To reinitialize the loading process, if the process was terminated by a fatal load error, you can issue the digit 7 in response to the * query.

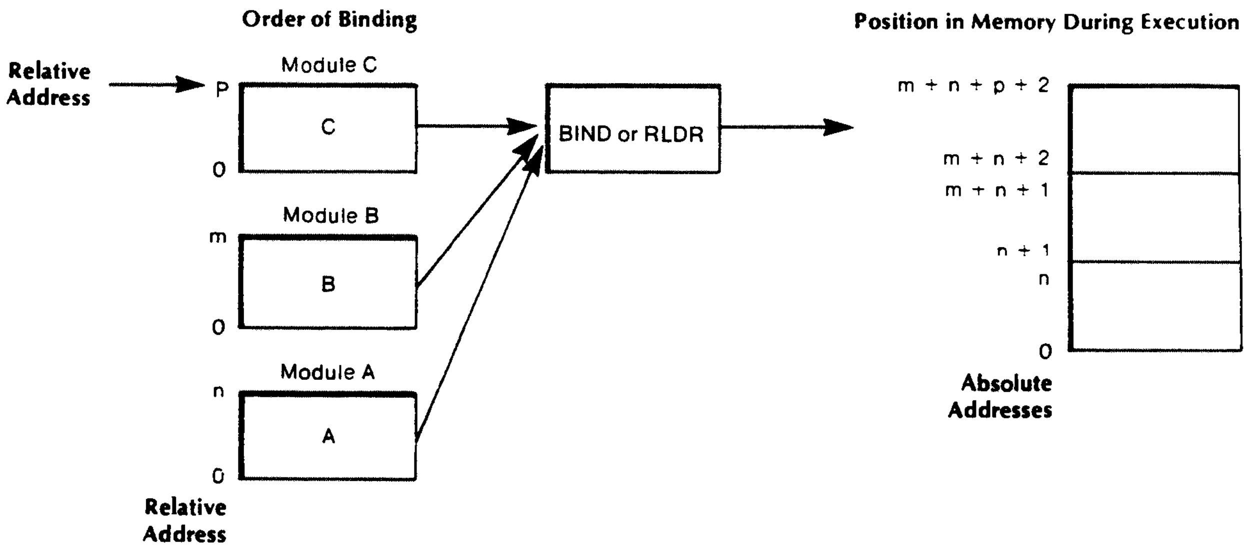


Figure 1.4 How RLDR operates on binary modules

SD-00637

Response	Effect
1	Load a relocatable binary or a library tape from the teletype reader
2	Load a relocatable binary or a library tape from the paper tape reader
3	Force a loading address for normally relocatable code
4	Complement the load-all-symbols switch
5	Print current memory limits
6	Print a loader map
7	Reinitialize the loader
8	Terminate the load process to prepare for execution
9	Print all undefined symbols

Table 1.1 Responses to asterisk loader prompt

Restart Procedures

To restart the loading process, press RESET, enter 000377 in the data switches, and press START.

Input to the Stand-Alone Loader

The input to the relocatable loader is in the form of relocatable binary tapes output from the relocatable assembler. Each tape is deviced into a series of blocks and must contain at least a title block and a start block. The order of blocks input to the loader is shown below. Each block type is described in detail later in this manual

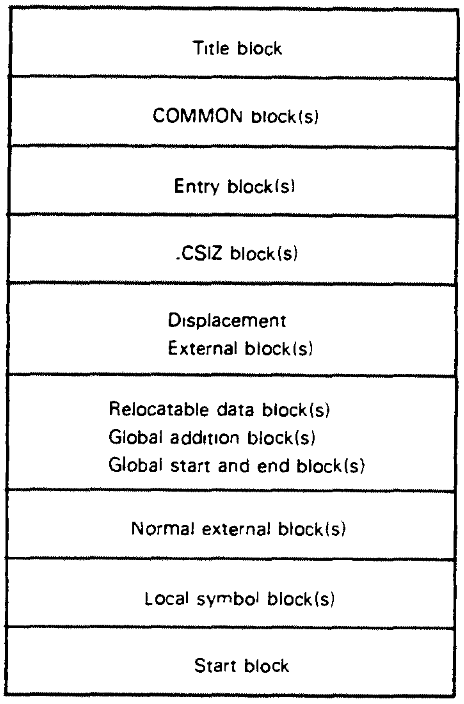


Figure 1.5 Blocks on binary tape caption ID-00119

To load a single relocatable binary tape, the user responds to the asterisk prompt with

1 Input from teletype reader, or

2 input from paper tape reader

The binary tape will be loaded, and the loader will respond with an asterisk after the start block has been processed. The user can then input another relocatable binary tape or give one of the other responses to the prompt.

Relocation Variables ZMAX and NMAX

The load addresses input to the relocatable loader are in three modes: absolute, page zero relocatable, and normal relocatable Absolute data is loaded to the assembler at the locations specified For relocatable data, the loader is initialized to assume two relocation variables called NMAX and ZMAX.

ZMAX has an initial value of 50_8, where 50_8 is the first location to be loaded with page zero relocatable data. As each location is filled, ZMAX is updated to reflect the next location available to receive page zero relocatable data.

NMAX has an initial value of 440_8, the first location available to load normal relocatable data. As each location is filled with normal relocatable data, NMAX is updated to represent the next available location.

Determining the Current Values of ZMAX and NMAX

The relocatable loader maintains a symbol table (also called a loader map) which is built down in core from the saved area (response to SAFE =). The current values of ZMAX and NMAX are given in the loader map. The user can obtain the current table (SST and EST) and CSZE (unlabeled COMMON size) by responding to the asterisk prompt with a 5.

The first two values given are NMAX and ZMAX in the formats:

NMAX *nnnnnn*
ZMAX *nnnnnn*

where *nnnnnn* represents the 6-digit current octal value of each variable.

Forcing a Value of NMAX

Before input of a relocatable binary tape, the user can force NMAX to a given value, thus determining the absolute load address for normally relocatable data.

The user can force NMAX to a given value by

1. Entering the desired octal value in the console data switches, bits 1-15.

2. Responding to the * prompt with the digit 3

The Symbol Table (Loader Map)

The symbol table is constructed downward in core from the first address below the saved area, determined by the SAFE= query.

At the top of the symbol table are entries for NMAX, ZMAX, CSZE, EST, and SST. Below these are all entry symbols, undefined externals, and local symbols (if the load locals switch was set by a response of 4 to the asterisk prompt).

A defined symbol is represented by three words. The first two words contain the symbol name (in radix 50, using 27 bits). There is a six-digit octal value in the third word.

For an entry symbol, its value is an absolute number – either the core address of the word for which the symbol was a label or the value of the symbol as defined by an equivalence.

For undefined external normals, the number is the absolute address of the last of a chain of references to the symbol. If the number is −1, there were no references. Each reference to the symbol has been replaced by the absolute address of the previous reference, the first reference having been replaced by −1.

For undefined external displacements, there may be more than one reference chain. The value printed is the absolute address at the last reference in the first such chain. The actual symbol table entry has the two-word symbol and the end addresses of *n* chains, where the first *n* − 1 have bit 0 set and the last does not, signifying the end of the symbol table entry. Within a chain, references are linked by 8-bit relative displacements, contained in the right half of each storage word. Each chain is terminated by a word having 377_8 in its right byte. Thus, if two consecutive references are more than 376_8 words apart, a new chain must be started, as follows.

At termination of the load, undefined external normals are resolved by the relocatable loader to the value −1. Each occurrence of that symbol is replaced with a −1.

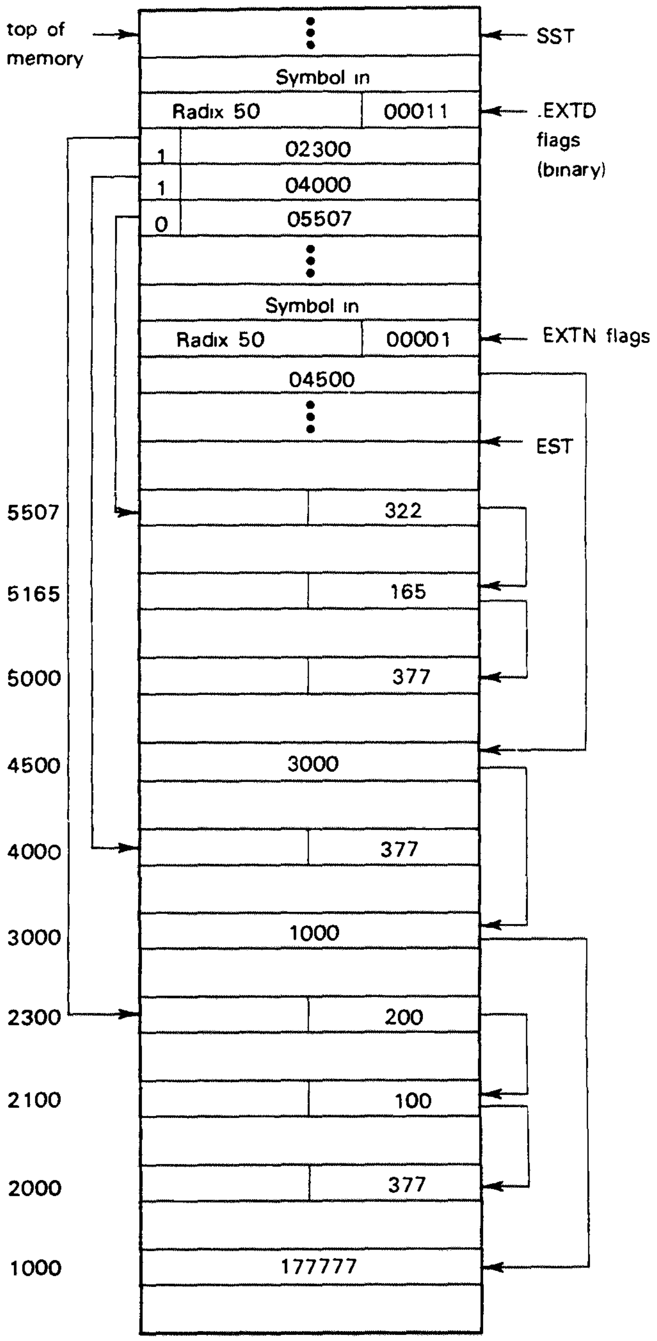


Figure 1.6 Memory map of load phase of RLDR

ID-00120

The symbol may be flagged on the loader map with one of two letters. If the symbol is an unresolved external (one for which no entry has yet been defined) a U appears to its left.

An M to the left of the symbol means that the symbol is defined in two or more entries or .COMM statements.

An example of part of a symbol table follows EST is the lowest word of the symbol table, − 1. SST is the highest word within the symbol table CSZE is the size of unlabeled Common.

```
TEMP
           NMAX      044723
           ZMAX      000244
           CSZE      000000
            EST      050131
            SST      052001

          AHESZ      000000
          BINAR      006207
          BUFFE      016711
          BUGIN      000000
          CHSW       000174
          CKSQ       000000
          COL01      035622
          COL02      035623
          COL03      035624
          COL04      035625
          COL05      035626
          COL06      035627
          COL07      035630
          COL08      035631
          COL09      035632
          COL10      035633
          COL11      035634
          COL12      035635
          COL13      035636
          COL14      035637
          COL15      035640
          COL16      035641
          COL17      035642
```

Figure 1.7 Symbol table load map DG-25170

To obtain a copy of the symbol table, respond to the loader prompt by typing a 6.

To obtain a copy of the undefined symbols in the symbol table only, respond to the asterisk prompt with the digit 9.

Execution of Loaded Programs

When you respond to the loader prompt (*) by typing an 8, you end the loading procedure. The programs previously loaded are then moved to the absolute addresses indicated by the loader map. Until the load process is terminated, the loader resides in low core, and all programs are loaded assuming a pseudo address for location 00000, which exists above the loader itself. Once you terminate the loading, the following steps occur.

- Location 377 is unconditionally initialized to 2406 (JMP @ . +6), providing a convenient restart address.
- The starting address of the loaded core image is set by the loader to location 405 of the User Status Table (UST). See the UST layout in this chapter.
- Memory is shuffled down to reflect the true addresses as shown in Figure 1.8.

The loader passes information to loaded routines that may be useful for their execution. This information is passed in the User Status Table, which starts at location 400.

Starting Address for Execution

After shuffling memory, the relocatable loader will stop (HALT). When you press CONTINUE, the loader will stop again if no starting address has been specified on any of the binary tapes.

If only one of the binary tapes loaded contains a starting address, that address will receive control, regardless of the order in which the tapes were loaded.

If more than one of the binary tapes loaded contains a starting address, the last starting address specified by a binary tape will receive control for execution.

Loading of Library Tapes

Library tapes are tapes containing a set of relocatable binaries that are preceded by a library start block and terminated by a library end block. These tapes are provided by Data General as part of the standard software packages.

Library tapes are loaded in the same way as relocatable binaries. You mount the library tape in the appropriate input device and respond to the loader asterisk query by typing either 1 or 2.

The library load mode is initiated when the loader encounters a library start block. The loader does not request a new load mode until after encountering the library end block.

The loader will load selected relocatable binary programs from the library tape. Programs in a library tape are loaded only if there is at least one entry symbol defined by that program which corresponds to a currently unresolved external in a previously loaded program.

For example, if programs A, B, and C are on a library tape and A calls B, which then calls C, none of those programs will be loaded unless A is called by a program loaded before the library tape. If A has an entry corresponding to a previously unresolved external, then programs A, B, and C will be loaded.

Loading Local and Title Symbols

Local and title symbols are normally loaded only when you intend to use the symbolic debugger, since the symbols will otherwise occupy symbol table storage space unnecessarily

The loader maintains a switch for local and title symbols which is set by default to inhibit loading of local and title symbols. You can complement the switch, altering the mode, if you respond to the loader asterisk query by typing a 4.

The loader responds with an S when the switch is set to load local and title symbols. You can complement the switch by issuing another 4, and the loader responds with an R, indicating that the switch has been reset.

Reinitialization of Loading

If the loading process is terminated by the fatal error (see the upcoming section on error detection), or if you wish to start the loading process again, the loader must be reinitialized. You can reinitialize loading by typing a 7 in response to the loader asterisk query.

The loader will then reset ZMAX and NMAX to 50 and 440 respectively, and will reinitialize the symbol table, eliminating all entries.

Determining Available Core

Total core available for program loading is dependent upon loader size, core configuration, the size of the SAFE area, and the number of symbols entered in the symbol table The following is an *approximate* formula for determining core available for program loading:

core available = *sc* - 2500 - SAFE - 3 **ne*)

where:

sc	is the core capability of the system configuration.
ne	is the number of entry points (plus the number of user symbols when in mode 4) defined by all relocatable programs to be loaded.

The quantities are given in octal.

You can obtain a printout of the current memory limits during loading by typing a 5 in response to the loader query

Error Detection

The loader detects two types of errors – fatal and nonfatal Fatal errors prevent further loader action unless the user reinitializes (response of 7). Nonfatal errors do not stop loading, but may change the intended state of your loaded system.

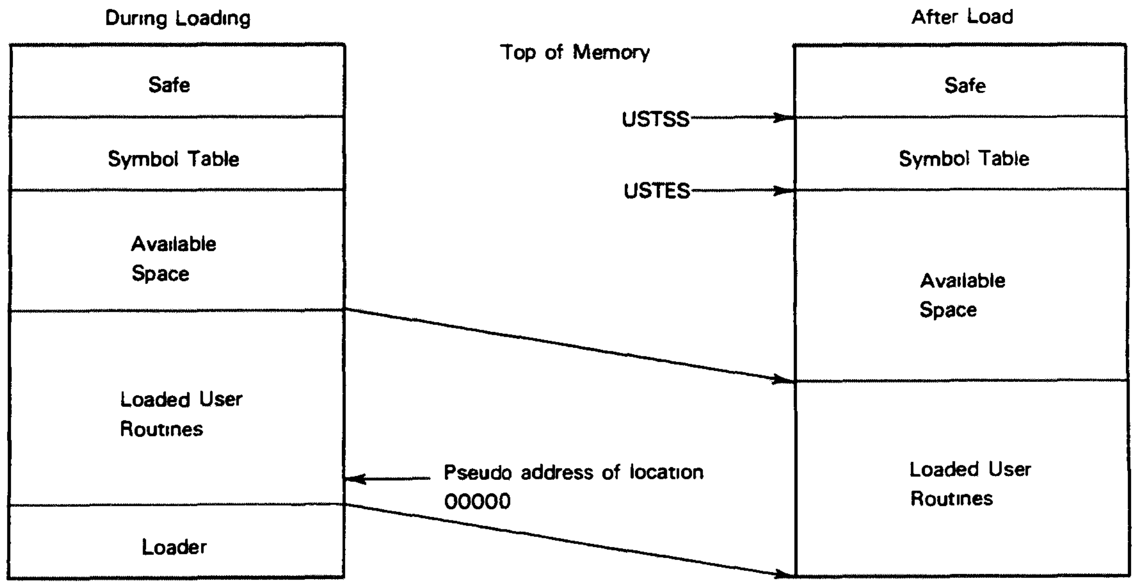


Figure 1.8 Memory map after loader phases

ID-00121

All errors are indicated by a two-letter code The code is printed at the terminal, followed by a symbol name, if needed, and by a six-digit octal number, if applicable The meaning of the octal number is defined later for each of the error codes. The message has the general form.

ee sssss nnnnnn

where:

ee	is the error code.
sssss	is the symbol name
nnnnnn	is the octal number.

Nonfatal Errors

The following table lists nonfatal errors

Codes	Description
DO	Displacement overflow
IB	Illegal block type
IN	Illegal NMAX
ME	Multiply defined symbol
TO	Input timeout
XE	External undefined in external expression

Table 1.2 Nonfatal Errors

DO Error

DO *nnnnnn*

If, while attempting to resolve an external displacement, the loader finds that the displacement is too large, a displacement overflow (DO) error results. The displacement is too large if:

- The index = 00 and the unsigned displacement is > 377.
- The index ≠ 00 and the displacement is outside the range −200 ≤ displacement < +200.

The location *nnnnnn* represents the absolute address at which overflow occurred. The displacement is left unresolved with a value of 000.

IB Error

IB *nnnnnn*

If an illegal relocatable block type is read, an illegal block (IB) error results. The octal number *nnnnnn* represents the block code of the illegal block. The loader will issue an asterisk query after issuing the error code. If the error was caused by an improper tape mounted in the reader, you should replace it with a relocatable binary or library tape and try loading again

IN Error

IN *nnnnnn*

If the user types a 3 in response to the loader prompt and the value in the switches is lower than the current value of NMAX, an IN error results. The octal number *nnnnnn* is the illegal value of NMAX. NMAX is unchanged.

ME Error

ME *sssss nnnnnn*

If an entry or named common (.COMM) symbol with the same name as one already defined is encountered during loading, a multiply defined entry (ME) error results. The name of the symbol *sssss* is followed by the absolute address *nnnnnn* at which it was originally defined.

TO Error

TO *nnnnnn*

If the time between input characters becomes excessive, a timeout (TO) error occurs. The usual cause of the timeout error is a binary tape without a start block or a library tape without an end block. The location *nnnnnn* represents the location in the loader where the timeout occurred. The loader issues an asterisk prompt when the error occurs.

XE Error

XE *sssss*

If a .GADD block is encountered referencing a symbol which has not yet been defined, an external error (XE) (defined in external expression) occurs. Zero is stored in the memory cell. The undefined symbol *sssss* is printed out following the error indicator.

Fatal Errors

The following table contains a list of fatal errors.

Code	Description
CS	Checksum error
MO	Memory overflow
NA	Negative address
NC	Named COMMON error
OW	Overwrite of memory
XL	External location undefined
ZO	Page zero overflow

Table 1.3 Fatal errors

CS Error

CS *nnnnnn*

If a checksum computed on any block differs from zero, a checksum (CS) error results The octal number *nnnnnn* represents the incorrectly computed checksum.

MO Error

MO *nnnnnn*

If the value of NMAX plus the loader size itself conflicts with the bottom of the loader's symbol table, a memory overflow (MO) error occurs The error implies that the user programs are too large to be loaded into the memory configuration. The octal number *nnnnnn* is the value of NMAX that caused the overflow

NA Error

NA *nnnnnn*

If bit 0 of an address word is set to 1, a negative address (NA) error occurs The assembler restricts addresses to the range.

$0 \leq \text{address} < 2^{15}$

A reader error, however, could cause bit 0 to be set. *nnnnnn* represents the negative address.

NC Error

NC *sssss nnnnnn*

where.

sssss gives the symbol name of the labeled COMMON and *nnnnnn* indicates the size of the labeled COMMON requested by the present .COMM.

A named common (NC) error results from either of the following conditions:

- Two programs have different sizes for a given area of labeled COMMON (defined by .COMM statements)
- The symbol table flags that are associated with the symbol are not 00010_2.

OW Error

OW *nnnnnn*

The loader prevents memory cells from being overwritten by subsequent data once they are loaded. If an attempt to overwrite is made, an overwrite (OW) error occurs. The absolute address at which the overwrite was attempted is given by *nnnnnn*.

XL Error

XL *sssss*

If a .GLOC block is encountered with data to be loaded at the address of a symbol that is as yet undefined, an external location undefined (XL) error results. The undefined symbol is given by *sssss*.

ZO Error

ZO *nnnnnn*

If in loading page zero relocatable code the code overflows the page zero boundary of 377, a page zero overflow (ZO) occurs The absolute address of the first word of the data block that caused the overflow is given by *nnnnnn*

User Status Table of Loading Information

The relocatable loader provides information concerning the loading process in a table called the User Status Table (UST) The UST originates at location 400. Following is a template with explanatory information.

```
UST     = 400     ;START OF USER STATUS AREA
USTPC   =   0     ;PROGRAM COUNTER
USTZM   =   1     ;ZMAX
USTSS   =   2     ;START OF SYMBOL TABLE
USTES   =   3     ;END OF SYMBOL TABLE
USTNM   =   4     ;NMAX
USTSA   =   5     ;STARTING ADDRESS
USTDA   =   6     ;DEBUGGER ADDRESS
USTHU   =   7     ;HIGHEST ADDRESS USED BY
                  LOAD MODULE
USTCS   =  10     ;COMMON AREA SIZE
USTIT   =  11     ;INTERRUPT ADDRESS
USTBR   =  12     ;BREAK ADDRESS
USTCH   =  13     ;NUMBER OF CHANNELS/TASKS
USTCT   =  14     ;CURRENTLY ACTIVE TCB
USTAC   =  15     ;START OF ACTIVE TCB CHAIN
USTFC   =  16     ;START OF FREE TCB CHAIN
USTIN   =  17     ;INITIAL START OF NREL CODE
USTOD   =  20     ;OVERLAY DIRECTORY ADDRESS
USTSV   =  21     ;FORTRAN STATE VARIABLE SAVE
                  ROUTINE
USTEN   = USTSV   ;LAST ENTRY
```

Table 1.4 User status table

Location 400 USTPC is the program counter. The loader initializes this word to 0, indicating that the program has never run.

Location 401 USTZM points to the first available location in page zero for page zero relocatable code.

Locations 402 and 403 USTSS and USTES point to the start and end of the symbol table respectively as shown in the diagram in Figure 1.7 The loader sets 402 and 403 to 0 if the debugger is not loaded.

Location 404 USTNM contains NMAX The loader sets the pointer to the first free location for further loading, or for allocation of temporary storage at run time

Location 405 USTA points to the program starting address, specified by the .END statement. If no starting address is specified by any loaded program, -1 is stored in 405 If several programs specify starting addresses, USTSA contains the address specified in the last program loaded. (Location 377 contains JMP @ . = 6, which transfers control to the starting address of the program Therefore, the user can conveniently restart his program at 377, assuming that he has specified a starting address.)

Location 406 USTDA points to the starting address of the debugger, or if the debugger is not loaded, 406 contains -1.

Location 407 USTHU is set to the value of NMAX at the termination of loading Since no operating system changes USTHU during program execution, it can be used to reset NMAX when a program is restricted

Location 410 USTCS contains the size of the FORTRAN unlabeled COMMON area, used when the binary relocatable programs being loaded contain .CSIZ blocks, such as those generated by the FORTRAN compiler.

Locations 411 and 412 USTIT and USTBR are set to 0 by the loader

Locations 413 through 416 and 420 through 421 These locations are compatible with RDOS.

Location 417 USTIN contains the address of the start of normally relocatable code (440_8).

Chapter 2

Source Programs

Assembly language source programs are composed of a series of lines. A line is composed of all characters scanned by the assembler until it reaches a Carriage Return or Form Feed. The assembler recognizes several types of lines; each source line must conform to a given structure, depending on its type. In addition, each line must contain only characters in the character set of the assembler.

Character Sets

The assemblers accept the following characters in a source program:

1. Alphabetics A through Z
2. Numerals 0 through 9
3. Special Characters:
   ! " # & * + , - . / : ; (=) @
4. Format control and line terminators:
   Carriage Return, Form Feed, Space, Tab

Appendix C contains a table of the Extended Assembler and Macroassembler ASCII character subsets and their octal equivalents. As shown in that table, the assemblers also accept lower-case alphabetics, but automatically translate them to their upper-case equivalents.

Three characters are unconditionally ignored by the assemblers:

Character	Octal Value
null	000
Line Feed	012
Rubout	177

Any character not in the ASCII character set is flagged with a B (bad character) on the assembly listing.

The assembler substitutes a backslash (\ ) for all source program characters which have an incorrect parity. This character is ignored by higher level processing; that is, L A is processed as LA.

Source Lines

ASCII characters are combined to form source lines. The majority of source lines affect the generation of a 16-bit value (with relocation properties) for each memory location at execution time. Any line of this type is said to produce a storage word. The storage word has a value, usually defined by an expression or instruction, and an address. At assembly time, the address assigned is the contents of the current location counter (LC). The generation of each 16-bit storage word causes the contents of the location counter (LC) to be incremented by one. Thus, in general, storage words are assigned to consecutively increasing LC values.

Several types of source lines produce storage words. Others are used to define symbols, control the assembly process, and provide instructions to the assembler. Assembly source lines must be one of the following types:

Data lines
Instruction lines
Pseudo-op lines
Equivalence lines

Data Lines

A data line is one of the simplest in the assembly language. It consists of a single numeric expression.

A data line generates either a 16-bit storage word (if the expression is a single precision integer) or a 32-bit storage word (if the expression is a double precision integer or a floating point number). In fact, data lines provide the only means for storage of double precision and floating point values.

The special character @ (explained in Chapter 3) can be used anywhere in a data line to generate a full word indirect address. After the expression is evaluated, the assembler places a 1 in bit 0 (the indirect addressing bit) of the storage word. Thus, for example, all of the following data lines have the same value:

```
102644
102644@
2644@
@1322*2
```

Instruction Lines

An instruction line consists of an instruction mnemonic, or op code, and any required or optional argument fields. Every instruction line generates a 16- or 32-bit storage word which provides an instruction to the assembler, directing it to load an accumulator, add two accumulators, or increment an accumulator. Instructions are described in Chapter 3.

Pseudo-Op Lines

A pseudo-op line must begin with a permanent symbol (except the symbol .) and may be followed by one or more required or optional arguments. Some pseudo-op lines (such as .NREL and .ZREL) are merely commands to the assembler and generate neither a storage word nor 16-bit value. Others (such as .RDX) generate a 16-bit value, but do not increment the location counter.

Pseudo-ops and pseudo-op line syntax are described in Chapter 5.

Equivalence Lines

One means of assigning a symbolic name to a numeric value is by equivalence. An equivalence line associates a value with a symbol; that symbol can then be used any time the value is required. An equivalence line has the form

usr-sym = exp

where:

usr-sym is a user symbol conforming to the rules for symbols given in Chapter 4.

exp is an expression or instruction.

The symbol to the left of the equal sign in an equivalence (*usr-sym*) must be previously undefined in pass 1. The expression to its right must be able to be evaluated in pass 1. Examples of equivalence lines are:

```
A   = 342

B   = A/2

INS = ADD# 0, 1, SKP
```

An equivalence line assembles as a 16-bit value, but does not affect the current location counter.

Labels

Any source program line can contain a label. A label allows you to name a storage word symbolically. By using such a label, you can reference the storage word without regard for its numeric address.

A label is simply a user symbol; it must appear at the beginning of a source line and must be followed by a colon (:). A label must conform to rules established for any symbol, as described in Chapter 4. Like other symbols, a label has a value which is that of the current location counter; that is, it is the address of the next storage word assembled. Since some source lines do not generate storage words, this definition is not necessarily associated with the statement in which it appears. The following source line is given the label LOOP:

```
LOOP: ADD#0, 1,SKP ⟨CR⟩
```

A source line can consist solely of a label. For example:

```
LAB: ⟨CR⟩
```

Any source line can have more than one label, provided all symbols are defined at the beginning of the line. For example:

```
LOOP:LAB1:LAB: ADD# 0.1, SKP ⟨CR⟩
```

Comments

An assembly language program can include comments to facilitate program checkout, maintenance, and documentation. A comment is not interpreted in any way by the assembler and cannot affect the generation of the object program. All comments must be preceded by a semicolon (;). Upon encountering a semicolon, the assembler ignores all subsequent characters up to a Carriage Return. The following source program lines illustrate the use of comments.

```
;THIS SUBROUTINE CALCULATES THE ABSOLUTE VALUE OF
A NUMBER IN AC0

      .TITL .ABSL
      .ENT .ABS
      .NREL

.ABS: MOVL# 0, 0, SZC      ;TEST SIGN
      NEG 0, 0             ;NEGATIVE IF NEGATIVE
      JMP 0, 3
      .END                 ;END OF ABSOLUTE VALUE SUBR.
```

Source Line Formatting

Within broad limits, you are free to determine the format of the source lines for a program. For example, all of the following lines have the same meaning to the assembler; they differ only in format.

```
LAB: ADD,2,3,SZR#;SKIP IF SUM = ZERO

LAB:ADD,2,3,SZR#;SKIP IF SUM = ZERO
    LAB: ADD 2 3 SZR # ;     SKIP IF SUM = ZERO
```

The special character # can appear anywhere in a source line

A common practice in writing source programs is to divide each line into four columns by means of three tab settings, using the left column for labels, the second column for the beginning of the source line, the third for arguments, and the right for comments. If the listing device is not equipped with automatic tabbing, the assemblers simulate tabs by spacing to the nearest assembler-defined tab position (and always leaving at least one space between fields) Assembler-defined tab positions are set eight columns apart; that is, at columns 9, 17, 25, and so on.

Under the Stand-alone Operating System, programs for systems that do not use magnetic tape or cassette I/O are loaded using the Stand-alone Relocatable Loader, 091-000038, as described in Chapter 1. Relocatable loader 091-000038 is supplied in absolute binary.

Programs for SOS systems that have either magnetic tape or cassette I/O, however, are loaded by the relocatable loader 089-000120, which is supplied as part of the SOS cassette or magnetic tape system.

Operation

The SOS relocatable loader 089-000120 must be loaded by the core image loader in accordance with the procedures outlined in the Stand-alone Operating System User's Manual.

Once loaded, the relocatable loader will print the following prompt at the terminal:

```
RLDR
```

The user responds by typing a command line giving the names of files used as input to and output from the relocatable loader.

The user response to the RLDR prompt consists of a list of file names which may have local switches. The command causes the relocatable loader to produce from one or more .RB or .LB files, an executable core-resident program and a core image (save) file on magnetic tape or cassette. Both files start at address zero. The same file cannot be used for both input to and output from the relocatable loader. At least one input file and an output save file must be designated in the command line.

The SOS magnetic tape/cassette relocatable loader is compatible with the RDOS relocatable loader and builds a core resident program in much the same way:

The user program ZREL code starts at location 50 and builds upwards in page zero. The User Status Table is contained in locations 400-437. The User NREL code starts at location 440 and builds upward in memory.

The symbol table is retained in core only if the symbolic debugger, Debug III, is loaded. At termination of loading, the symbol table is moved down to the end of NREL code.

The maximum core size of each loaded program cannot exceed the maximum core address less 1325_8. The 1325 locations are required for the core image loader and pass 2 of the relocatable loader.

Upon completion of a successful load, the message "OK" is output on the terminal and the system halts with the loaded program in core.

Symbol Table

The symbol table is built in high core and moved down to the end of NREL code at termination of loading The symbol table is retained in core only if the symbolic debugger, Debug III is loaded. Debug III is supplied on relocatable binary tape 089-000073 and must be loaded as one of the input files in the RLDR command line if a symbol table is desired. The symbol table is similar to the one shown on page 1-7 for the Stand-alone Relocatable Loader.

User Status Table

Locations 400-437 contain the User Status Table (UST). The table is given below:

USTPC	0	
USTZM	1	ZMAX
USTSS	2	;START OF SYMBOL TABLE
USTES	3	;END OF SYMBOL TABLE
USTNM	4	;NMAX
USTSA	5	;STARTING ADDRESS
USTDA	6	;DEBUGGER ADDRESS
USTHU	7	;HIGHEST ADDRESS USED
USTCS	10	;FORTRAN COMMON AREA SIZE
USTIT	11	;INTERRUPT ADDRESS
USTBR	12	;BREAK ADDRESS
USTCH	13	;NUMBER OF CHANNELS/TASKS
USTCT	14	;CURRENTLY ACTIVE TCB
USTAC	15	;START OF ACTIVE TCB CHAIN
USTFC	16	;START OF FREE TCB CHAIN
USTIN	17	;INITIAL START OF NREL CODE
USTOD	20	;OVERLAY DIRECTORY ADDRESS
USTSV	21	;FORTRAN STATE VARIABLE SAVE

Command Line

When the prompt RLDR is output, the user responds on the same line with a list of input and output file names. Switches may be attached to one or more of the file names, and each space is separated from the next by at least one blank space. The general form of the command line is:

filename$_1$...*filename*$_n$

At a minimum, the command line must contain at least one input file name and one output save file name:

inputfilename outputfilename/S

where:

S is a switch indicating the save file.

A number of switches may be appended to the names of input and output files in the command line. They are as follows:

Output File Name Switches

/S	The /S follows the name of a cassette or magnetic tape file, indicating that that device will be used for output of the save file. If no save file is specified or if a file is incorrectly specified as a save file, an error message will result and the loader will reinitialize itself, displaying the prompt "RLDR."
/L	The /L follows the name of a device and causes a numerically ordered listing of the symbol table to the device. The output device for the listing cannot be the same as that used for the save file.
/A	This switch may be appended to the same device as that having the /L switch. It causes an alphabetic as well as numeric listing to result. The /L switch must be present.

Input File Name Switches

/N	NMAX, the starting address for loading a given input file may be changed from the default address by use of this switch. The /N follows an absolute address, given in octal, and precedes the name of the input file to be loaded beginning at the octal address. The octal address given must be greater than the current value of NMAX.
/P	Files to be loaded may be on different cassettes. /P following a file name causes a halt before the file of that name is loaded that allows the user to mount a new cassette containing the input file. When the loader halts, the message: PAUSE - NEXT FILE *filename* is printed, where *filename* is the name of the file that had the /P switch. When the new cassette is mounted, the user restarts loading by pressing any terminal key.
/U	/U causes local user symbols appearing within the file preceding the switch to be loaded.
n	*n* is a digit in the range 2-9. The input file preceding the switch is loaded the number of times specified by the switch.

Command Line Error Messages

Following are the command line error messages:

```
NO INPUT FILE SPECIFIED
NO SAVE FILE SPECIFIED
SAVE FILE IS READ/WRITE PROTECTED
```

The save file device must be either a cassette or magnetic tape and must premit both reading and writing.

I/O ERROR nn

where:

nn is one of the following RDOS codes:

160	Illegal file name
180	Attempt to read a read-protected file.
180	Attempt to write a write-protected file.
180	Non-existent file.

Examples of Command Lines

```
$TTO/L/A CT2:0/S $PTR CT1:6 16500/N CT1:0
```

Input files are the $PTR, CT1:6 and CT1:0. NMAX is reset for CT1:0 to 16500$_8$. The save file is written to CT2:0 and an alphabetically ordered listing is output.

If one of the input files, CT1:6, CT1:0 or the $PTR contains the debugger, a symbol table will be generated.

```
MT1:0/S MT0:1 MT0:2 $PTP/L
```

Input files are MT0:1 and MT0:2. The save file is output to MT1:0. A numeric listing is to the paper tape punch.

```
CT0:0/S CT1:2 CT1:0/P
```

Input files are on different cassettes, so the /P switch allows a pause for the user to change the cassette tape on unit 1. The save file is output to CT0:0.

Restart Procedure

The loader can be stopped and restarted at location 377 any time in Pass 1 (up until the end of the listing of the symbol table). Once Pass 2 starts, the loader must be reloaded from cassette or magnetic tape.

Error Messages

In addition to the command line error messages described on the previous page, the loader produces explicit error messages that are printed to the console. These include both fatal and non-fatal error messages. The error messages are followed by an appropriate identifying location, symbol, or both.

Non-fatal Errors

Non-fatal errors do not stop loading but may change the intended state of the output file. The non-fatal error messages are as follows.

DISPLACEMENT OVERFLOW nnnnnn

A displacement overflow error occurs if the loader finds the displacement is too large when attempting to resolve an external displacement. The displacement is too large if:

- The index = 00 and the unsigned *displacement* is 377.
- The index 00 and the displacement is not in the range: $-200 \leq$ *displacement* $< +200$

nnnnnn is the absolute address where overflow occurred. The displacement is left unresolved with a value of 000.

ILLEGAL BLOCK TYPE nnnnnn

The error message normally occurs if the input file is not a relocatable binary or library file. The file in error will not be loaded. Octal number *nnnnnn* is the block code of the illegal block.

MULTIPLY DEFINED ENTRY sssss nnnnnn

This error occurs when an entry symbol or named common (.COMM) symbol, *sssss*, having the same name as one already defined is encountered during loading. *nnnnnn* is the absolute address at which the symbol was originally defined.

EXTERNAL UNDEFINED IN EXTERNAL EXPRESSION sssss

This error occurs if a .GADD block is encountered that references an as yet undefined symbol, *sssss*. Zero is stored in the memory cell.

BINARY WITHOUT END BLOCK

This error occurs when a binary file has no end block. The file is loaded up to the point where the error is discovered.

ILLEGAL NMAX VALUE nnnnnn

This error occurs when the user attempts to force the value of NMAX to a value lower than the current value of NMAX, i.e., if the octal value following a /N local switch is lower than the current value of NMAX. *nnnnnn* is the illegal value. NMAX is unchanged.

NO STARTING ADDRESS FOR LOAD MODULE

This error occurs if at assembly time the user failed to terminate at least one of the programs to be loaded with a .END pseudo-op that was followed by a starting address for the save file. The starting address can be patched by the user into location 405 (USTSA) of the User Status Table.

EXTERNAL NORMAL/DISPLACEMENT CONFLICT sssss

This error occurs when a symbol *sssss* appears in a .EXTD block in one module to be loaded and in a .EXTN block in another module.

CAUTION OLD ASSEMBLY ssss

This error occurs when this program was assembled by an incompatible assembler.

Fatal Errors

If an error is fatal, the error message and the location at which it was discovered are followed on the next line by a second message:

```
**FATAL LOAD ERROR**
```

and the loader gives the prompt, RLDR. For example:

```
CHECKSUM ERROR nnnnnn **FATAL LOAD ERROR** RLDR
```

The fatal errors are as follows.

CHECKSUM ERROR *nnnnnn*

This error occurs if a checksum that is computed on some block differs from zero. *nnnnnn* is the incorrect checksum.

NEGATIVE ADDRESS nnnnnn

This error occurs if bit 0 of an address word is set to 1. The assembler restricts addresses to the range: 0 *address*; however, the error can be caused by a reader error. *nnnnnn* represents the negative address.

PAGE ZERO OVERFLOW nnnnnn

This error occurs in loading page zero relocatable data if the data overflows the page zero boundary 377_8. The absolute address of the first word of the data block that caused the overflow is given by *nnnnnn*.

NAMED COMMON ERROR sssss nnnnnn

This error occurs if two programs have different sizes for a given area of labeled COMMON (defined by .COMM statements), or if the symbol table flags that are associated with the symbol are not 00010_2. *sssss* gives the symbol name of the labeled COMMON and *nnnnnn* indicates the size of the labeled COMMON requested by the present .COMM.

SYMBOL TABLE OVERFLOW

This error occurs during loading if the size of the symbol table becomes so large that it would overwrite the loader in core.

EXTERNAL LOCATION UNDEFINED sssss

This error occurs if a .GLOC block is encountered with data to be loaded at the address of a symbol, *sssss*, that is as yet defined

MEMORY OVERFLOW nnnnnn

If the value of NMAX plus the loader size itself conflicts with the bottom of the loader's symbol table, a memory overflow error occurs. The error implies that the user programs are too large to be loaded in the memory configuration. The octal number *nnnnnn* is the value of NMAX that caused the overflow.

Chapter 3

Fundamental Assembly Tools

Character Input

You can input characters to the assembler in one of two modes:

string and normal.

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, for example.

   ```
   ;SET MASK BITS
   ```

   is terminated with a Carriage Return or Form Feed.

2. Macro Definition Strings (not macro calls)

   A macro definition string is valid only when you are using the Macroassembler. The 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 ⟨CR⟩
   LDA       0,2 ⟨CR⟩
   MOVZL     1,1 ⟨CR⟩
   %
   ```

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 $."
   .TXT $ THE PROGRAM RESPONDS, "WHAT?"$
   ```

Normal Mode

All other input is in normal mode, where the input string consists of characters in a subset of the ASCII character set. divided into lines. Each line of characters is terminated either by a Carriage Return or by a Form Feed In normal mode, the assembler recognizes the following ASCII codes:

All alphabetics,
Numerics,
Relational operators,
Most conventional punctuation,
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 give 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 the basic unit of assembly language recognized by its specific class. Atoms fall into five classes:

Operators
Break Characters
Numbers
Symbols
Special Characters

Operators

Operator characters are used with single-precision integers and symbols to form expressions. There are three classes of operators: arithmetic. logical. and relational. Table 3.1 shows the characters in each class.

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

Table 3.1 Operators

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; for example, 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 (=) immediately follows it. We use □ only where we must: normally, spaces are obvious in the formats.

Note that when a macro call references arguments, commas, spaces, and tabs do not produce the same results. For example, each of the macro calls

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

has two arguments and assembles the same way. On the other hand, when written in the following manner,

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

each has three arguments; the first a null. Also, a space with a comma in a macro call may produce a format error, because the assembler tries to use it to expand the macro.

See Chapter 6 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 enclose a symbol or an expression

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

```
MYMACRO [3,4 ⟨CR⟩
5,6]
```

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

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

⟨CR⟩ A Carriage Return terminates a line of source code, for example,

```
MOVZR0,0,SNR ,CHK FOR ODD NUMBER ⟨CR⟩
```

A Form Feed also terminates a line of source code

Numbers

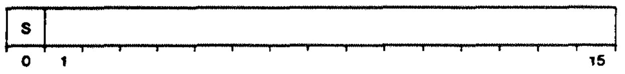
Three types of numbers are defined for the Macroassembler

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.

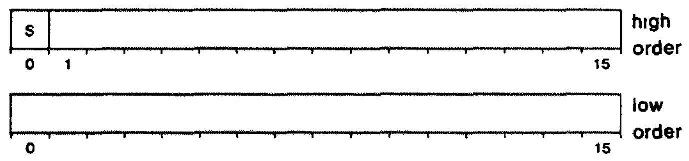
You can use single-precision integers both in expressions and in data statements, but you can use double-precision integers and floating-point numbers only in data statements

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 assigned, 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 through −32767.



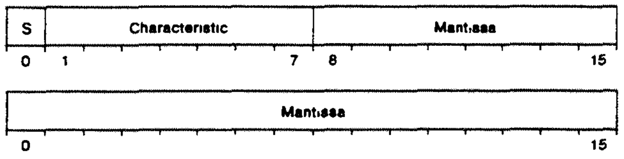
A double-precision integer is represented in memory in two contiguous words, where the first word is the high-order word. Using two's complement notation, a double-precision integer is represented as:



Bit 0 of the high-order word is the sign bit.
Bits 1 through 31 contain the magnitude.

Double-precision integers cannot be combined in expressions; they can be used only in data lines.

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. An unsigned integer is considered positive.

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.

$16^{-1} \langle$ mantissa $\langle (1\text{-}16^{-6})$

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:

$16^{-1} * 16^{-16} \langle$ floating-number $\langle (1\text{-}16^{-6}) * 16^{63}$

which is approximately

$5\ 4 * 10^{-79} \langle$ floating-number $\langle 7.2 * 10^{75}$

Most routines that process floating-point numbers assume that all nonzero operands are normalized, and that 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, there is 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:

$\begin{Bmatrix} [+] \\ [-] \end{Bmatrix}$ *d [d...d] [.] [break]*

Each *d* is a digit within the range of the current input radix The initial *d* must be in the range of 0 through 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

Table 3.2 Digit representation

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   OF        , Decimal 15.
000377   OFF       ; Decimal 255.
007777   OFFF      ; Decimal 4095.
177777   OFFFF     ; Decimal 65535.
```

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

```
Operators:          + - * / B ' &
                    == <> <= >= ><
Break characters    □ ) <CR> ;
```

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 or underscore conventions. This breaks the number string and is then ignored. The following example illustrates the character B as a digit and as a bit shift operator. We use the underscore rather than the backarrow in these examples. 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).

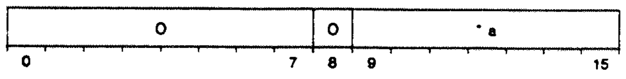
Only the single ASCII character immediately following the quotation mark is interpreted. The ASCII characters Null, Line Feed, and Rubout are invisible to the assembler, and cannot be input with this format. All other ASCII characters can be converted to single-precision integers

```
00101     "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, "<CR> assembles an octal 15 and also terminates the line The "a format character is packed right to left in the word·



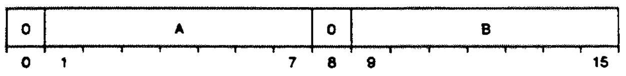
A second format, which uses apostrophes, can convert up to two ASCII characters to a single-precision integer. The format is:

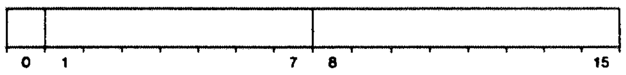
'string' or *'string'* <CR>

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 format character, are packed left to right in the word:





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:

$\begin{Bmatrix}[+]\\ [-]\end{Bmatrix}$ *d[dd..d] [.]* D *break*

Each *d* is a digit within the current radix. The first *d* must be in the range of 0 through 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 ⟨CR⟩) that indicates the end of the integer.

Do not terminate a double-precision integer with an operator, or you will receive a format error (F).

The radix of a double-precision integer may be in the range of 2 through 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 underscore or the backarrow convention:

```
000020    .RDX 16
00455     12D        ; D represents digit
                     ; 13 (decimal).
000000    12_D       ; 12 is a double-pre-
000022               ; cision 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 $\begin{Bmatrix}.\\ \text{E}\end{Bmatrix}$ *d break*

where *d* is a digit in the range of 0 through 9.

A single-precision floating-point number is represented in source format in one of the following ways.

± *d[d..d].d[d..d]*[E*[* ± *d]d]*

± *d[d..d]*E ± *d[d]break*

Each *d* is a digit 0 to 9. The mantissa and exponents are always converted to decimal (for example. 2E9 = ⟩ 2 * 10^9).

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

break is typically one of the terminating characters □ or ; or ⟨CR⟩.

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

```
041376    254.333
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 backarrow convention ( ) ASCII 137:

```
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-pre-
                       ; cision integer.
000000   567     D     ; Hex double-pre-
002547                 ; cision integer.
001067   567.          ; Decimal single-
                       ; precision integer.
000000   567.     D    ; Decimal double-
001067                 ; precision integer.
002547   567           ; Hex single-
                       ; precision integer.
005316   567     B14   ; Hex single-
                       ; precision int, bit
                       ; shifted one bit.
012634   567     B13   ; Hex single-pre-
                       ; cision 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 direct the assembly process, or they represent numeric values of internal assembler variables. A symbol can be written as a series of letters, numbers, or periods. Any other character in a symbol is interpreted as an error and is given a bad character (B) flag in the assembly listing. The various classes of symbols will be discussed in Chapter 4. Their source representation is given below:

a[b...b]break

where:

a can be one of the alphabet characters A through Z, or the period (.) or question mark (?).

b can be one of the characters A through Z, 0 through 9, period (.), question mark (?), or underscore (__).

break is any character not rated above, for example, space and comma.

By default, the assemblers recognize only the first five characters before the break character, although they print more than five on listings You can specify eight-character symbols with the MAC global /T switch (described in Chapter 8), which produces an extended (rather than 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. For more information on @ and #, see Chapter 4, or the appropriate reference manual for your computer.

Other special characters are a set of square brackets surrounding a symbol ([MYSYM]), a dollar sign in a macro definition (TR$ =), and a backslash followed by a symbol (\ ONES). See the .GOTO pseudo-op (Chapter 4) and Chapter 6 for these atoms.

@ Commercial AT Sign

Bits can be set when a commercial at sign (@) (or a series of such signs) is contained within a memory reference instruction (MRI), extended memory reference instruction, or before an expression, in the following ways:

1. When the rest of the MRI is evaluated, an at 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 is evaluated, an at sign sets bit zero of the word to 1.

```
000025     25
100025     @25
```

3. An at 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 is evaluated, the number sign 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 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 00000 to 00002, since all lines of source are assembled, but the second source line is not listed.

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

Chapter 4
Expressions

An expression is a symbol, single-precision integer, or a series of symbols and/or single-precision integers separated by operators. The format of an expression in source code is

[*operand*$_1$]*operator operand*$_2$

where *operator* is a Macroassembler operator.

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. If an expression contains an illegal operand, such as an external symbol, instruction mnemonic, double-precision number, or floating-point number, the source line is given an error flag on the assembly listing.

Operators

The assembler operators are shown in Table 4.1.

Type	**Operator**	**Meaning**
Arithmetic/ Logical	B	Bit alignment
	+	Addition, for example, (2+3), or unary plus, for example, (+3)
	−	Subtraction, for example, (5−2), or unary minus, for example, (−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.
Relational	==	Equal to
	<>	Not equal to
	<=	Less than or equal to
	<	Less than
	>=	Greater than or equal to
	>	Greater than

Table 4.1 Assembler operator

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 following 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

$0 \langle \textit{operand}_2 \langle 15_{10}$

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

for *operand*$_1$ B *operand*$_2$, the result value will be

$\textit{operand}_1 {*} 2(^{15} - \textit{operand}_2)$

where *operand*$_2$ is implicitly evaluated in decimal unless parentheses are used; for example,

```
.RDX8
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 posi-
                        ; tion 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 So far, all expressions have had absolute operands which produced absolute results.

An operand, however, can have one of five properties. These are:

- Absolute
- Page zero relocatable (ZREL)
- Page zero byte-relocatable (ZREL)
- Normal relocatable code (NREL)
- Normal byte-relocatable code (NREL)

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 NREL) 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 in Table 4.1. The expressions in the table are defined as:

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 Table 4.2, 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	Illgeal

Table 4.2 Expression definitions

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 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 following example 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-Abso-
                           ;lute 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    1/10   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
```

Page zero relocatable operands must reside in page zero Normal relocatable operands can be relocated anywhere in core except page zero These relocatable operands are converted to absolute during the loading process by the addition of a relocation constant. The relocatable loader maintains two relocation constants, a zero relocation (Cz) and a normal relocation (Cn) constant, to fix the addresses of relocatable values.

Byte relocatable operands are page zero or normal relocatable storage words that act as byte pointers: bits 0 through 14 of the pointer contain an address and bit 15 specifies the byte to be operated on. A byte pointer of this kind can be formed simply by doubling an address, and can be retrieved and regenerated by a shifting operation. The loader adds either 2Cz or 2Cn to each byte relocatable value to convert it to absolute.

During loading, the relocatable loader can add one (and only one) of five possible constants to a word: O, Cz, Cn, 2Cz, or 2Cn. This has two implications when relocatable operands are combined in expressions:

1. Although the assemblers permit the combining of page zero and normal relocatable operands in expressions, the operands must be such that either the page zero or the normal relocatable operands cancel out. For example, the following expression is valid

   Z1 + N1 − Z2 + N2 − N3

   (where Zn represents page zero relocatable operands; Nn represents normal relocatable operands).

   However, the expression

   Z1 + Z2 + N1

   is invalid.

2. Loader modification of an address by more than twice a relocation factor is illegal.

Symbols

The assemblers recognize 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: they direct the assembly process, and they represent numeric values of internal assembler variables.

The permanent symbol period (.), when used alone, is a special symbol with a value equal to the current contents of the location counter. Therefore, the instruction

```
LDA 3,.+6
```

and

```
LDA 3,6,1
```

have the same meaning.

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 5 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 position of the symbol 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 ⟨CR⟩
```

directs the assembler to pack bytes left/right.

Example 2

When enclosed in parentheses,

```
(.TXTM) ⟨CR⟩
```

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                .TXTM1

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 A lists all permanent symbols and Chapter 5 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 a given 16-bit instruction value. Instruction values are discussed later in this chapter

Semipermanent symbols can be saved and used, without redefinition, for all subsequent assemblies. The 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, more commonly, add to a given set. In addition to instruction mnemonics, several semipermanent symbols are provided by Data General for use as operands within expressions. These include the skip mnemonics (SKP, SZR, SNR, and SZC) used in arithmetic and logical instructions, and device codes (TTI, TTO, PTR, and PTP) used in I/O instructions.

Semipermanent symbols are not printed in the cross-reference listing unless enabled by the global /A switch (see Chapter 8).

User Symbols

You can define any symbol that does not conflict with permanent or semipermanent symbols. User symbols can:

- Name a location symbolically,
- Assign a numeric parameter to a symbol,
- Name external values,
- Define global values.

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 therefore 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 with the global /U switch (Chapter 8).

By default, only the first five characters in any user symbol are recognized, although longer symbol names may be printed 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 8, global /T switch, for more detail.

ASCII Character Conversion

a single ASCII character (except Null, Line Feed, and Rubout) can be converted to its 7-bit octal equivalent if it is preceded by a quotation mark ("); thus, an ASCII character can be represented as a single precision integer. For example.

```
000053        "+          ;ASCII "+"
000055        "-          ;ASCII "-"
```

The quotation mark (") can be used in expressions, for example:

```
000141        "A+40
000172        "Z+40
000112        "A+9.
```

Note that the quotation mark assembles as octal 15 and also terminates the line.

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 (for instance, [MYSYM], a dollar sign in a macro definition (as in TR$=), and a backslash followed by a symbol (for instance 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, or before an expression, a commercial at sign (@), or a series of at signs, will set bits in the following ways:

1. When the rest of the MRI has been evaluated, an at 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 at sign sets bit 0 of the word to 1.

```
000025     25
100025     @25
```

3. An at 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 supress 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
```

Indirect Addressing

Indirect addressing can be specified for a memory reference instruction or a data word if one or more at signs (@) are included anywhere in a source program line. If the assembler encounters an at sign, it evaluates the memory reference instruction or data word, then sets the indirect addressing bit (bit 5 for a memory reference instruction, bit 0 for a data word) to 1 For example:

```
000004     JMP    RLO
002004     JMP   @RLO
```

Setting No-Load Bit

You can set the no-load bit (bit 12) of an arithmetic or logical instruction by including one or more number signs (#) in the source program line. If the assembler encounters a #, it evaluates the arithmetic or logical instruction, then sets bit 12 to 1, thus preventing the loading of the shifter output.

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 5.

Table 4.3 shows the various instruction types.

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

Table 4.3 Instruction types
* Instructions marked with an asterisk (*) can be assembled on any machine, but executed on ECLIPSE computers only

Arithmetic and Logical (ALC) Instructions

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

dest-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, which prevents the *destination-ac* from being loaded and leaves the carry unchanged.

The following bit pattern shows each assembled ALC instruction and the effect of the instruction, shift, carry, and skip mnemonics.

1	Source-ac	Dest-ac	ALC mnemonic	Shift	Carry	NL	Skip
0	1 2	3 4	5 7	8 9	10 11	12	13 15

ALC Mne.	Bits 5 6 7	Effect
COM	0 0 0	Places logical complement of C (*source-ac*) in *destination-ac*
NFG	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	Subtracts 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 0
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 NSW 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

For more information on the effect of these instructions on the carry bit, see the appropriate programmer's reference for your computer.
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
```

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 that specifies a device.

The following bit pattern shows each assembled I/O instruction without accumulator, and the effect of the instruction and Busy/Done mnemonics.

0	1	1	0	0	I/O-mnemonic	B/D	Device-code
0	1	2	3	4	5 7	8 9	10 15

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)

Examples of I/O Instructions without an accumulator follow:

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

I/O Instructions With Accumulator

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

The following bit pattern shows each assembled I/O instruction with accumulator, and the effect of the instruction as well as busy/done mnemonics.

0	1	1	ac	IOA mnemonic	B/D	Device-code
0	1	2	3 4	5 7	8 9	10 15

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)

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

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

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

*The HALTA I/O instruction applies to ECLIPSE computers 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
READSaccumulator	DIAaccumulator,CPU
INTAaccumulator	DIBaccumulator,CPU
MSKOaccumulator	DOBaccumulator,CPU
HALTAaccumulator	DOCaccumulator,CPU

Remember that the HALTA I/O instruction applies only to ECLIPSE computers.

The following bit pattern shows each assembled I/O instruction without device code, and the effect of the instruction mnemonics.

0	1	1	ac	IAC mnemonic	0	0	1	1	1	1	1	1
0	1	2	3 4	5 7	8	9	10	11	12	13	14	15

IAC Mne.	Bits 5 6 7	Effect
READS	0 0 1	Reads contents of console data switches into AC
INTA	0 1 1	Places device code of first device on bus in bits 10 through 15 of AC, acknowledging interrupt
MSKO	1 0 0	Sets Interrupt Disable Flags in devices according to mask in AC

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 device 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 are shown in Table 4.4.

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

Table 4.4 Assembly I/O instructions without argument fields

Examples of these instructions follow

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

Memory Reference Instructions (MR)

Memory Reference Instructions Without Accumulator

A memory reference 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. The valid ranges for an address are 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 $E = C(LC) + \text{displacement}$ and, therefore $C(LC) - 200_8 \langle E \rangle C(LC) + 177_8$
2	Address is based on the contents of AC2: $E = C(AC2) + \text{displacement}$ and, therefore $C(AC2) - 200_8 \langle E \langle C(AC2) + 177_8$
3	Address is based on the contents of AC3: $E = C(AC3) + \text{displacement}$ and, therefore $C(AC3) - 200_8 \langle E \langle C(AC3) + 177_8$

An address can be 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 $E = C(LC) + \text{address}$.

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. Therefore, 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 37_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.

The following bit pattern shows each assembled MR instruction and the effect of the instruction mnemonics

0	0	0	MR mnem	I	Mode	Displacement
0	1	2	3 4	5	6 7	8 15

MR Mne.	Bits 3 4	Effect
JMP	0 0	Jumps to effective address (loads effective address into LC)
JSR	0 1	Jumps to subordinate 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 4.1 illustrates how effective addresses are formed.

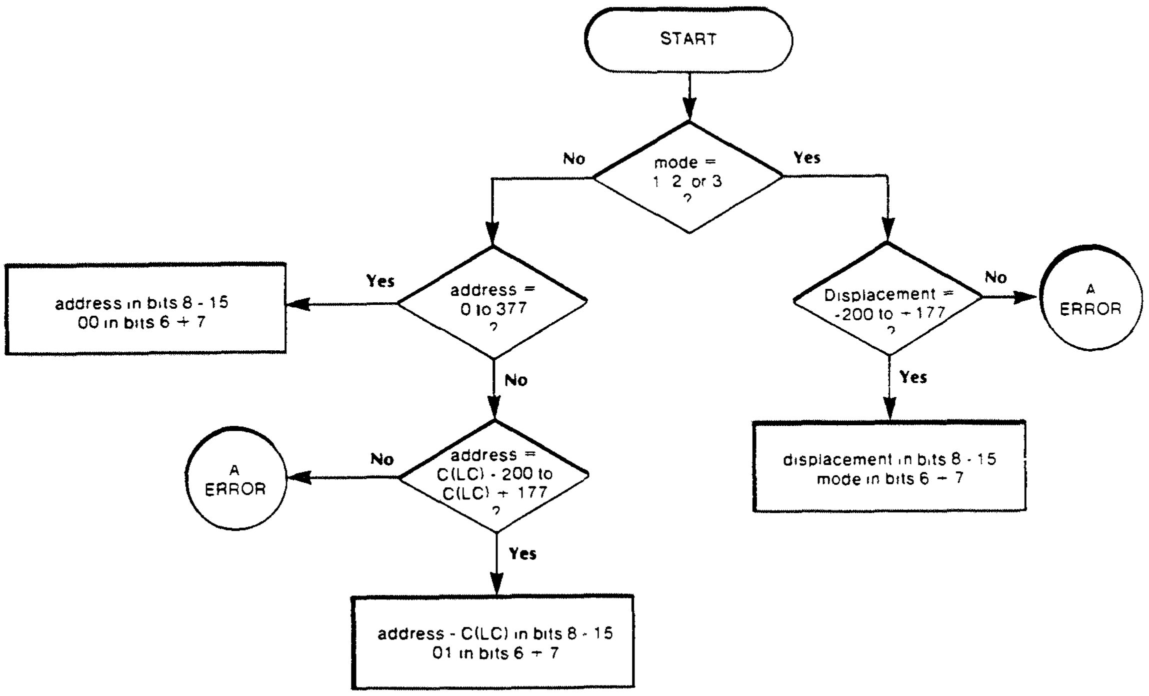

START
mode = 1 2 or 3 ?
No
Yes
address = 0 to 377 ?
Yes
address in bits 8 - 15 00 in bits 6 + 7
No
address = C(LC) - 200 to C(LC) + 177 ?
No
A ERROR
Yes
address - C(LC) in bits 8 - 15 01 in bits 6 + 7
Displacement = -200 to + 177 ?
No
A ERROR
Yes
displacement in bits 8 - 15 mode in bits 6 - 7

Figure 4.1 Formation of effective address for MR instruction. SD-00422

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:

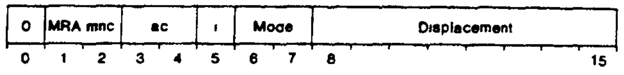
mra-mnemonic is a semipermanent symbol: LDA or STA.

ac 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

The following bit pattern shows each assembled MR instruction with accumulator and the effect of the instruction mnemonics.



MRA Mne.	Bits 1 2	Effect
LDA	0 1	Load contents of effective address in AC
STA	1 0	Stores contents of AC in effective address

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
```

ECLIPSE Instructions

MAC and ASM recognize 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 5. The ECLIPSE-only instructions are:

Extended Memory Instructions (defined by pseudo-ops .DEMR (MAC only) and .DMRA (ASM and MAC):

EDSZ
EISZ
EMJMP
EMSR
ELDA
ESTA
ELEF

Commercial Instructions (defined by pseudo-op .DCMR MAC only):

ELDB
ESTB

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

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.

ELSZ (MAC)
EISZ (MAC)
EJMP (MAC)
EJSR (MAC)

Those requiring an accumulator are:

ELDA (MAC)
ESTA (MAC)
ELEF (MAC)

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 a 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 are valid only when you are using the macroassembler. 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'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 instruction, accumulator field, and optional index; the second specifies the displacement.

Each of the instructions 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 the 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. Here is an example using a commercial extended MR instruction:

```
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

Floating-point instructions are valid only when you are using the macroassembler. There are two general types of floating-point instructions: those which use a displacement, and those which do not.

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 the following).
index	Is a value in bits 0 and 1, which the assembler used to calculate the effective address.

Index Value	Effective Address Determination
00	Displacement is treated as an 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
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.

Chapter 5

Pseudo-Ops and Value Symbols

This chapter lists and describes the pseudo-ops and value symbols, both by category and alphabetically.

Both the Extended Assembler (ASM) and the Macroassembler (MAC) assemble a source program, and the Extended Relocatable Loader (RLDR) processes it into an executable binary file. Pseudo-ops are permanent symbols that direct the assembly process. No pseudo-op or facility can change or delete them. Symbols defined by pseudo-ops are semipermanent; that is, they can be deleted (.XPNGed), but they usually exist 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 constitute 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 can be executed 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 instruction 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 an accumulator.
.DIAC	Define an instruction requiring an accumulator.
.DICD	Define an instruction requiring an accumulator and a count
.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 instruction 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. This instruction is valid only when you are using the macroassembler

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 Data General employ these pseudo-ops to define assembler instructions. These definitions reside in disk file .MAC.PS, which is usually required for assembly of source programs, and is further described in Chapter 8.

Symbol Table Pseudo-Op Format

Symbol table pseudo-ops have the form:

$$\textit{pseudo-op}\square\textit{user-symbol} = \begin{Bmatrix}\textit{instruction}\\ \textit{expression}\end{Bmatrix}$$

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 4.

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

Except for .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* = $\begin{Bmatrix} \textit{instruction} \\ \textit{expression} \end{Bmatrix}$

.

.

.

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

expr stands for expression.

$expr_1$, $expr_2$, and $expr_3$ 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 evaulate
        ;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

0       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) results.

```
        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:

1. 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) results.

2. If an expression does not meet the requirements of the field, for instance, if

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

   the field is unaltered, and an overflow error (O) results.

3. 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	Set the current of storage locations.
.NREL	Specify NREL code relocation.
.ZREL	Specify page zero relocation.

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

Intermodule Communication Pseudo-Ops

Intermodule (interprogram) communication pseudo-ops allow symbols in one module to be referenced by other 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 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 evaulation of an expression provided to the pseudo-op. The following pseudo-ops are provided:

Pseudo-Op	Instruction
.DO (MAC only)	Assembled the following source lines a specified number of times
.ENDC (MAC and ASM)	Define the end of a series of conditional-assembly source lines.
.GOTO (MAC only)	Supress assembly of source lines until the specified symbol is encountered.
IFE (MAC and ASM)	Assemble only if expression equals zero.
.IFG (MAC and ASM)	Assemble only if expression is greater than zero.
.IFL (MAC and ASM)	Assemble only if expression is less than zero.
.IFN (MAC and ASM)	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 6 and are valid only when you are using the macroassembler.

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 the 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 move 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. Stack pseudo-ops are valid only when you are using the macroassembler.

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 parity 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. Similarly, .TXTM can return the most recent .TXTM value.

Listing Pseudo-Ops and Values

Listing pseudo-ops are valid only when you are using the Macroassembler. This 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 listing 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.

The .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 the .END statement.

Pseudo-Op Dictionary

Current Location Counter (.)

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

Example

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

.ARGCT (MAC only)
Number of Most Recent Macro Arguments

.ARGCT

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

Example

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

.BLK (ASM and MAC)
Reserve a Block of Storage

.BLK *expression*

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

Example

```
                          .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 (ASM and MAC)
Reserve a Named Common Area

COMM *user-symbol expression*

This pseudo-op reserves a named common area for inter-program communication. The name of the area is *user-symbol*, and its size in words is *expression*. The first routine loaded that declares the named user-symbol reserves this area. 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.

The .COMM pseudo-op must come before other code, and immediately after the .TITL pseudo-op.

Example

```
        .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 appear as shown in Figure 5.1 when the program executes

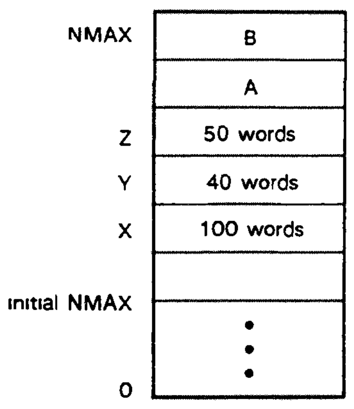


Figure 5.1 A and B after program execution ID-00122

.CSIZ (ASM and MAC)
Specify an Unlabeled Common Area

CSIZ *expression*

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

```
        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·

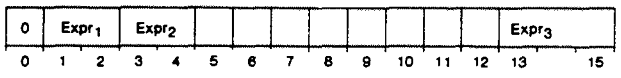
```
RLDR A X (CR)

     A
     X
     NMAX      001037
     ZMAX      000050
     CSZE      000050
```

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

.DALC (ASM and MAC)
Define ALC Instruction

.DALC *user-symbol* = $\begin{Bmatrix} \textit{instruction} \\ \textit{expression} \end{Bmatrix}$

The .DALC pseudo-op defines *user-symbol* as a semi-permanent 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 expression$_1$ *expr*$_2$ [*expr*$_3$]

Example

```
      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*.

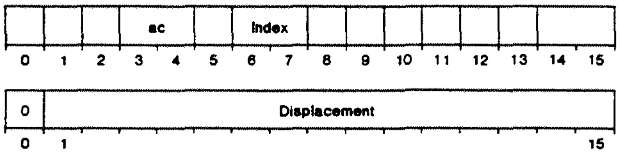
If you use this pseudo-op to define a three-character symbol which includes, or is followed immediately by, the letters Z, O, C, L, R, or S (or any combination of them), the Macroassembler will set bits 8 and 9 to 10 and 11 as follows:

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

.DCMR (MAC only)
Define Commercial Memory Reference Instruction

.DCMR *user-symbol* = {*instruction* | *expression*}

This pseudo-op defines *user-symbol* as any 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.



The format for using this semipermanent symbol is:

user-symbol displacement [*index*]

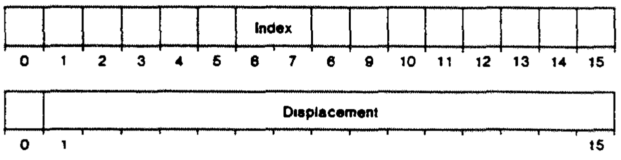
Example

```
       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 (MAC only)
Define Extended Memory Reference Instruction

.DEMR *user-symbol* = {*instruction* | *expression*}

This pseudo-op defines *user-symbol* as a semipermanent extended 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 4.

Example

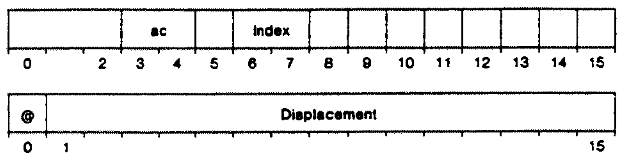
```
                .EXTN ADDR1, ADDR2, ADDR3
       102070   .DEMR EJMP = 102070
                .NREL

 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, ADDR2, 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 (MAC only)
Define an Extended Memory Reference Instruction with Accumulator

.DERA *user-symbol* = $\left\{\begin{matrix}\textit{instruction}\\ \textit{expression}\end{matrix}\right\}$

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.



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 are described in Chapter 4.

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

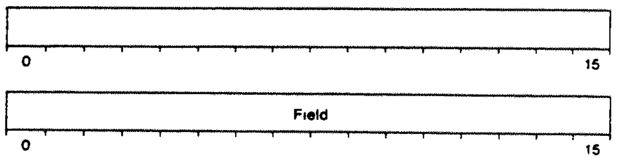
Example

```
                        .TITL TEST 9
         122070     .DERA ELDA = 122070
                    NREL
   00034'133470     ELDA 2, .+5, 3
         000041'
   00036'122470     ELDA 0, .+3
         000002
   00040'132470     ELDA 2, .+4
         0000003
FF0                 ELDA .+3      ;AC needed
```

.DEUR (MAC only)
Define Extended User Instruction

.DEUR *user-symbol* = $\left\{\begin{matrix}\textit{instruction}\\ \textit{expression}\end{matrix}\right\}$

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 an external displacement. This symbol implies the format of an 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

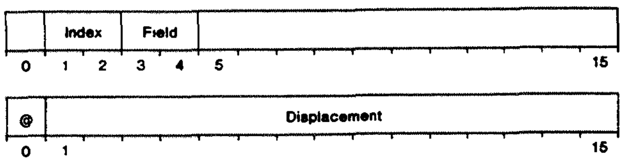
Example

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

.DFLM (MAC only)
Define Floating Load or Store Instruction

.DFLM *user-symbol* = $\left\{\begin{matrix}\textit{instruction}\\ \textit{expression}\end{matrix}\right\}$

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 this semipermanent symbol is:

user-symbol fpac displacement [*index*]

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

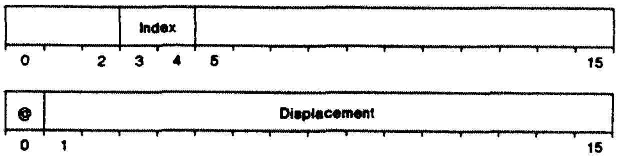
Example

```
        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 (MAC only)
Define Floating Load or Store Instruction

.DFLS *user-symbol* = $\left\{\begin{matrix}\textit{instruction}\\ \textit{expression}\end{matrix}\right\}$

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 instruction that does not require an accumulator. One field is required; an index is optional. These are assembled as shown.



The format for using this symbol is:

user-symbol displacement [*index*]

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

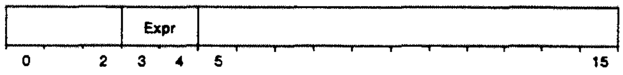
Example

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

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

.DIAC (ASM and MAC)
Define an Instruction Requiring an Accumulator

.DIAC *user-symbol* = $\left\{\begin{matrix} \textit{instruction} \\ \textit{expression} \end{matrix}\right\}$

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.



The format for using this symbol is:

user-symbol expression

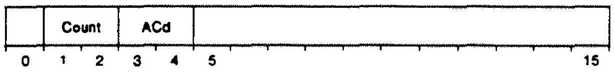
Example

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

.DICD (MAC only)
Define an Instruction Requiring an Accumulator and Count

.DICD *user-symbol* = $\left\{\begin{matrix} \textit{instruction} \\ \textit{expression} \end{matrix}\right\}$

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 an 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

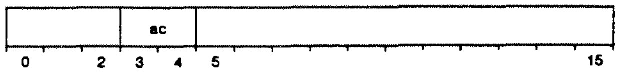
```
         100010     .DICD ADI = 100010
                    .NREL
     00055'104010   ADI 1, 1
     00056'110010   ADI 1, 2
     00057'160010   ADI 4, 0
    000060'104010   ADI 5, 1,     ,Overflow
          ;error- count field too large
FF                  ADI 1         ;Format
          ;error- 2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

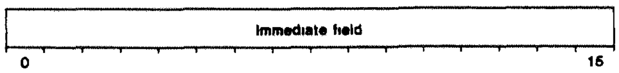
NOTE: Counts are entered in the instruction as (*specified value*-1). Thus, the range of permitted values is 1 through 4.

.DIMM (MAC only)
Define an Instruction Requiring an Accumulator and an Immediate Word

.DIMM *user-symbol* = $\begin{Bmatrix} \textit{instruction} \\ \textit{expression} \end{Bmatrix}$

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

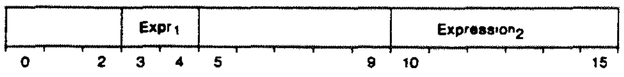
Example

```
         163770    .DIMM ADDI = 163770
                   .NREL
   00061 173370    ADDI 1002, 2
         001002
FF                 ADDI 0 ,Format error-
                         ;2 fields needed.
```

.DIOA (ASM and MAC)
Define an I/O Instruction With Two Required Fields

.DIOA *user-symbol* = $\begin{Bmatrix} \textit{instruction} \\ \textit{expression} \end{Bmatrix}$

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. One field is required, it is assembled as shown below.



The format of *user-symbol* is:

user-symbol expression$_1$ *expr*$_2$

Example

```
060400   .DIOA   DIA = 060400

         .NREL
070410    DIA    2, TTI
070610    DIAC   2, TTI
```

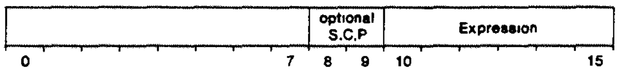
NOTE: If you use this pseudo-op to define a three-character symbol which is followed immediately by the letters S, C, or P, the Macroassembler will assume that letter to be an optional expression and will set bits 8 and 9 of the instruction word as folows:

Mnemonic	Bits 8 and 9
S	01
C	10
P	11

.DIO (ASM and MAC)
Define an I/O Instruction Without an Accumulator

.DIO *user-symbol* = {*instruction* / *expression*}

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.



The format of *user-symbol* is.

user-symbol expression

Example

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

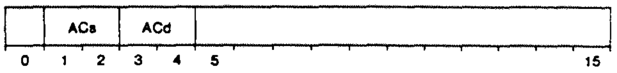
NOTE: If you use this pseudo-op to define a three-character symbol which is followed immediately by the letters S, C, or P, the Macroassembler will assume that letter to be an optional expression and will set bits 8 and 9 of the instruction word as follows:

Mnemonic	Bits 8 and 9
S	01
C	10
P	11

.DISD (MAC only)
Define an Instruction With Source and Destination Accumulators

.DISD *user-symbol* = {*instruction* / *expression*}

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 the semipermanent symbol is.

user-symbol source-ac destination-ac

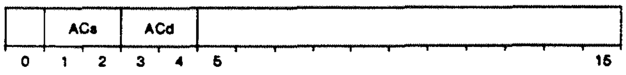
Example

```
000001    TXTM 1

102710    .DISD    LDB =    102710
          NREL
030402    LDA 2,   .PTR
146710    LDB 2,   1        ,AC1 loads
                            ,byte "A"


000162"
041101
042103    PTR. +1*2
000105    TXT "ABCDE"
```

.DISS (MAC only)
Define an Instruction With Source and Destination Accumulators Allowing Skip

.DISS *user-symbol* = $\begin{Bmatrix}\textit{instruction}\\ \textit{expression}\end{Bmatrix}$

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 destination-ac

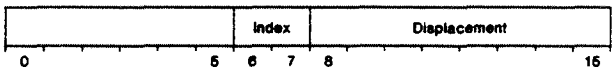
Example

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

.DMR (ASM and MAC)
Define a Memory Reference Instruction With Displacement and Index

.DMR *user-symbol* = $\begin{Bmatrix}\textit{instruction}\\ \textit{expression}\end{Bmatrix}$

This pseudo-op defines *user-symbol* as a semipermanent symbol having the value of *instruction* and *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.



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 4.

Example

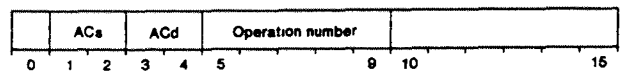
```
      000000  .DMR JMP = 000000
              .NREL
00000'000402  JMP .+2
00001'003001  JMP @1, 2
00002'001401  JMP 1, 3
```

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

.DMRA (ASM or MAC)
Define a Memory Reference Instruction With Two or Three Fields

.DMRA *user-symbol* = $\left\{\begin{matrix} \textit{instruction} \\ \textit{expression} \end{matrix}\right\}$

This pseudo-op defines *user-symbol* as a semipermanent symbol having the value of *instruction* and *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.



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 4

Example

```
      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 (MAC only)
Assemble Source Lines Repetitively

.DO *expression*

.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 magnetic tape, cannot execute a .DO loop more than once, because they cannot go back 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

```
                ;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.
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 6 describes the handling of .DOs and other conditionals within macros.

.DUSR (ASM and MAC)
Define User Symbol Without Formatting

.DUSR *user-symbol* = $\begin{Bmatrix} \textit{instruction} \\ \textit{expression} \end{Bmatrix}$

This pseudo-op defines *user-symbol* as a semipermanent symbol having the value of *instruction* or *expression* which follows the equal 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.

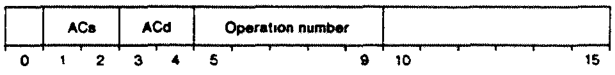
Example

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

.DXOP (MAC only)
Define an Instruction With Source, Destination, and Operation Fields

.DXOP *user-symbol* = $\begin{Bmatrix} \textit{instruction} \\ \textit{expression} \end{Bmatrix}$

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

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

.EJEC (MAC only)
Begin a New Listing Page

.EJEC

.EJEC begins a new page in the listing output.

Example

```
          ;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 (ASM and MAC)
End-of-Program Indicator

.END [*expression*]

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

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

        .END GTSET
```

.ENDC (ASM and MAC)
Specify the End of Conditional Assembly

.ENDC [*user-symbol*]

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 .IF*x* and suppresses the assembly of lines following .ENDC until the scan encounters another user-symbol enclosed in square brackets.

Example

```
                      .
                      .
 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 (ASM and MAC)
Define a Program Entry

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

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 any user symbol from separately-assembled programs by using one of the following pseudo-ops:

.EXTD
.EXTN
.GADD
.GLOC
.GREF

Example

```
                .TITL A
                .ENT B, .C
                .EXTN C
                .ZREL
00000-077777    .C: C

00000'006000-   B: JSR @ .C
                .END
```

.ENTO (ASM and MAC)
Define an Overlay Entry

.ENTO *overlay-name*

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: An overlay name cannot appear elsewhere in the file which delcares it as an overlay name, since its value is assigned at load time.

Example

```
                  .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 (MAC only)
Explicit End-of-File

.EOF
.EOT

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

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

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

.EXTD (ASM and MAC)
Define an External Displacement Reference

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

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:

$0 \langle$ *page-zero-address* $\langle 377$
$-200 \langle$ *displacement* $\langle 200$

NOTES: As a restriction, .EXTD can have only 256_{10} unique user symbols.

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

```
        .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 (ASM and MAC)
Define an External Normal Reference

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

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.

.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

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

.EXTU (ASM and MAC)
Treat Undefined Symbols as External Displacements

.EXTU

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

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

.GADD (ASM and MAC)
Add an Expression Value to an External Symbol

.GADD *user-symbol expression*

.GADD (global add) generates a storage word whose contents are 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

```
                .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 (ASM and MAC)
Reserve an Absolute Data Block

.GLOC *user-symbol*

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 have 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 binaries in the RLDR command line or a fatal RLDR will occur.

Example

```
             .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 (MAC only)
Suppress Assembly Temporarily

.GOTO *user-symbol*

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

Example

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

.GREF (MAC only)
Add an Expression Value to an External Symbol (0B0)

.GREF *user-symbol expression*

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

Example

See .GADD.

.IFE, .IFG, .IFL, .IFN (ASM and MAC)
Perform Conditional Assembly

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

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 listing is 1 if the condition is true and 0 if the condition is false.

Example

```
      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 6 describes handling of .IFs and DOs within macros.

.LMIT (MAC only)
Load Part of a Binary Module

.LMIT *user-symbol*

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. The *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*; see Figure 5.2.

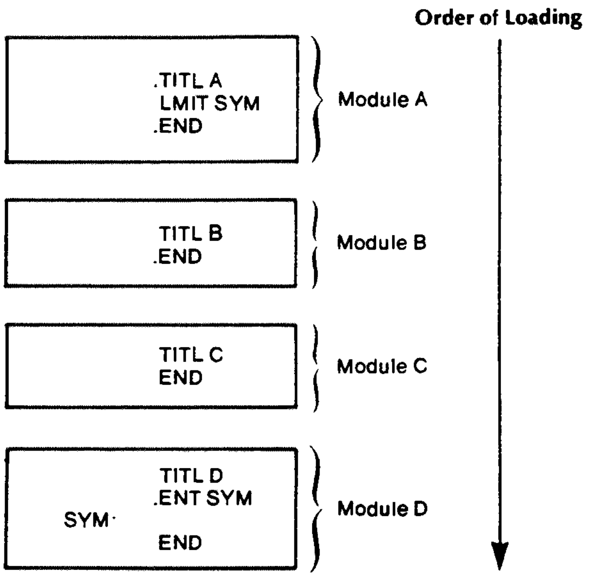


Figure 5.2 Partial loading of a binary module SD-00645

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 (that is 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 if 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 the RLDR 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 does not flag it as a multiply-defined symbol

.LOC (ASM and MAC)
Set the Current Location Counter

.LOC *expression*

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

Exceptions

If .LOC is pushed (.PUSH) to the assembler variable stack (see Stack Pseudo-op and Values in this chapter) and is subsequently used to restore the location counter, for example,

```
.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

```
 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 (MAC only)
Name a Macro Definition

MACRO *macro-name*

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

```
                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 (MAC only)
Indicate Macro Usage

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

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

Example

```
.MACRO TEST2
.DO     .MCALL)0
JSR @   .X              ;JSR if not 1st call.
.ENDC
.DO     .MCALL= =0      ;If 1st call, gen-
                        ;erate subroutine.
                        ;Save location counter
.PUSH   .LOC
.ZREL
.Z: 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 (MAC only)
Inhibit or Re-enable Listing Condition Lines

.NOCON *expression*

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.

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

Example

```
              ;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
              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
              3                 ;3
              .ENDC             ;.ENDC
```

.NOLOC (MAC only)
Inhibit or Re-enable Listing Source Lines Without Location Fields

.NOLOC *expression*

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.

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

Example

```
;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 (MAC only)
Inhibit or Re-enable Listing Macro Expansions

.NOMAC *expression*

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.

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

Example

```
                   .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 TEST 4
                                     5
                                     6
                                     .NOMAC 1  :You can inhibit
                                     ;or re-enable expansions
                                     ;at any time in a macro.
                                     7
                                     10
                                     %
                                     ;Now, call TEST 4:
                                     TEST 4
00006 000005                         5
00007 000006                         6
```

.NREL (ASM and MAC)

Specify NREL Code Relocation

.NREL

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

Example

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

.PASS (MAC only)

Number of Assembly Pass

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

.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
  1
%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 (MAC only)
Pop and Value and Relocation of Last Item Pushed Onto Stack

[*expression*] POP

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

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

 00001 000015   A

       000025   A=.POP
 00002 000025   A
O00003 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 (MAC only)
Push a Value and its Relocation Property Onto a Stack

.PUSH *expression*

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

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

```
000010   RDX8
000010  .PUSH .RDX
000012  .RDX 10
000010  .RDX POP
```

.RB (MAC only)
Name a Relocatable Binary File

.RB *filename*

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 as follows:

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

```
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 (ASM and MAC)
Radix for Numeric Input Conversion

.RDX *expression*

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

2⟨*expression*⟨20

The default radix is 8.

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.

(.RDX)

Example

(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 (MAC only)
Radix for Numeric Output Conversion

.RDXO *expression*

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.

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

Example

```
        00012  .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 (MAC only)
Set the Revision Level

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

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 through 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 contain 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

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

.TITL (ASM and MAC)
Entitle an RB file

.TITL *user-symbol*

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

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

.TOP (MAC only)
Value and Relocation of Last Stack Expression

.TOP

.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

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

.TXT, .TXTE, .TXTF, .TXTO (ASM and MAC)

Specify a Text String

.TXT*astrınga*
TXTE*astringa*
.TXTF*astringa*
.TXTO*astringa*

These pseudo-ops cause the assembler to scan the input followıng the character *a* up to the next occurrence of the character *a* ın strıng mode. The character *a* may be any character not used withın the strıng 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 lıne 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 containıng ASCII codes for the bytes Storage of a character of a string requıres seven bıts of an eight-bıt byte. You can set the leftmost (parıty) bıt to 0. 1. even parıty or odd parity as follows:

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

By default. bytes are packed left/rıght, 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 wıth the .TXTN pseudo-op.

Wıthin the string. you can use angle brackets (⟨⟩) to delimit an arithmetıc expression. The expression will be evaluated, masked to seven bits. and the eighth bit set as specified by the pseudo-op. Thıs 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 wıthın the expressıon.

```
.TXT "LINE 1 ⟨15⟩ ⟨12⟩" ⟨CR⟩
```

Example

```
      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 strıng
      120303
      142000
00011 140702  .TXTO /AB CD       .  AB CD
      020103
      142000
```

.TXTM (ASM and MAC)
Change Text Byte Packing

.TXTM *expression*

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.

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

Example

```
        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 (ASM and MAC)
Determine Text String Termination

.TXTN *expression*

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.

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

Example

```
          000000    .TXTN 0
00000     030462    .TXT "1234"

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

.XPNG (ASM and MAC)
Remove All Nonpermanent Macro and Symbol Definitions

.XPNG

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 8 further describes this mechanism.

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

Example

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

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

MAC/S XP ⟨CR⟩

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

.ZREL (ASM and MAC)
Specify Page Zero Relocation

.ZREL

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

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

Chapter 6

Advanced Features of the Macroassembler

The Macro Facility

The macro facility allows a string of source characters, sometimes consisting of many lines, to be named and subsequently referenced by name. This string of source characters if 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

Although you may write a macro definition only once, you can use the macro as often as needed. Each macro definition has the form:

.MACRO □ *user-symbol* ⟨CR⟩
macro-definition-string%

where:

user-symbol is the name that 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 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* which actually contains a backarrow, 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 through 9. To reference arguments 10 through 63, you must define a symbol in the form of a or ?a having the desired argument number in the range 10 through 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 %.

Example 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*$_1$ ↑ ⟨CR⟩ [*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, $string_1$ replaces every occurrence of ↑1 (or replaces ↑a where a evaluates to 1), $string_2$ replaces 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 single break characters.

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

Figure 6.1 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=B  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 6.1 Macro calls and expansions DG-25171

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 ⟨CR⟩
3,4]
```

calls macro ABC with 4 arguments. The call

```
ABC [1,2, ⟨CR⟩
3,4]
```

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

Figure 6.2 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 6.2 Forms 2 and 3 macro calls DG-25172

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 which 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 is 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).

Figure 6.3 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 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    .
                          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
```

Figure 6.3 Incorrect .DO example DG-25173

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, because the .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.

Macro Examples

A number of macro examples follow in Figures 6.3 through 6.6. Note the use of the recursive property in the macro, FACT, and the use of macro continuation and the special character within the macro VFD.

The first example is a macro to compute the logical OR of two accumulators. Its call takes a form similar to

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 6.4 Logical OR macro DG-25174

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

Figure 6.5 Factorial macro DG-25175

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 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 is as part of the macro expansion
:to avoid repetition

        .MACRO  PACK
**      .PUSH   .NOMAC
**      .NOMAC  1
        .PUSH   .RDX
        .PUSH   .ROXO
        .RDX 10
        .RDXO 16
I=      .ARGCT
J=      I-1
B=      11
W=      3+(`^J-`+/2)
J=      J-1
        .LOC    .+^I-1
        .DO     ^1
        .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 6.6 Packed decimal macro

DG-25176

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 5.)
3. Many statements are assembled with each macro call, but listing is inhibited except for the storage words assembled.

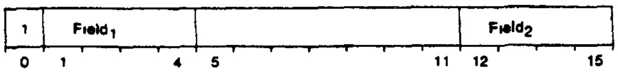
A powerful macro, used to associate a specified field layout with a given name, is shown below. The macro, VFD, defines a new macro named as the first argument in the call to VFD. Subsequent use of the name given in the VFD call generates a 16-bit storage word with a primary value to which fields are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

VFD□*type-name*□*primary-value*□*field*$_1$*-right-bit*□ ↑
⟨CR⟩
field$_1$*-mask*□...*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:

type-name□*field*$_1$□*field*$_2$...

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



```
We define three macros in this figure  VFD  ERROR
and SPECL  VFD actually uses ERRDR 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
                        ERRDR [FIELD NONZERO]
                         ENDC

                         IFE VALUE 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 6.7 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
            %
      D00003       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 > =^7
                    ERROR [FIELD OVERFLOW]
                     ENDC
            %
      000006        I=I+2
                         MACRD SPECL

                    J=J+1
            %
                    .ENDC
                         MACRO SPECL
```

DG-25177

```
                              IFN 17 > =^J
                             MASK=17
                             DATA=^J
               %
     000005            I=I-1
                                MACRO SPECL
                                       .DO 1S -1S

                                       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 D
                              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 6.7 VFD, ERROR and SPECL macros (continued)

DG-25177

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 through 9, A through 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 6.8 shows the generation of label entries in the macro BKT.

```
;The macro definition is:

          .MACRO BXT
          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          .DO5
                        BKT
  00000'014000          DSZ      COUNT
        000001'  TR$001=  .      ;Unique label.
                        .ENDC
                        BKT
  00000'014000          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 6.8 Generating labels

DG-25178

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 may 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:

$$\textit{memory-reference } [ac,] = \begin{Bmatrix}\textit{expression}\\ \textit{instruction}\end{Bmatrix}$$

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 (CR)
        .
        .
        .
TX:     .TXT "TEXT STRING" (CR)
```

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 AI \ I000, etc., is not included in the actual symbol. It appears in the symbol table as A000:.

Chapter 7
Extended Assembler Operating Procedures

One or more source files can be assembled by the RDOS Extended Assembler using the CLI command:

ASM *filename*$_1$ . . . *filename*$_n$

Input to the assembler must be ASCII source files. Output can be a relocatable binary file, a listing file, or both.

Switches

The following global switches can be appended to the ASM command name:

Switch	Result
/L	Produces a listing file.
/N	Does not produce a relocatable binary file.
/U	Includes local (user) symbols in the relocatable binary output.
/E	Suppresses error messages.
/S	Skips pass two. A BREAK is signaled after pass assembler containing user semipermanent symbols.
/T	Does not produce a symbol table list as part of the listing. (If a listing is requested, a symbol table is produced by default; this switch must be used to suppress the symbol table.)
/X	Produces a cross reference list of the symbol table.

The following local switches can be appended to a filename.

Switch	Result
/B	Outputs the relocatable binary to the specified file name.
/E	Outputs error messages to the specified file name.
/L	Outputs the listing to the specified file name, overriding the global /L.
/S	Skips this file on pass two. (This switch should be used only if the files do not assemble any storage words.)
/N	Does not list the specified file. (This switch is used when a listing is requested and only a selected number of files are to be assembled.)

File Name Searches

For input files, a search is performed first for *filename*.SR If it is not found and *filename* has no extension, a search is then made for *filename*. On output, a relocatable binary file. *filename*.RB and a listing file (if the global /L switch is specified), *filename*.LS, are produced; *filename* is the first source file specified without a /S, /L, or /B local switch.

Warning: The following command would cause the loss of the first relocatable binary disk image:

```
ASM ($PTR,$PTR)$LPT/L
```

Although two distinct source files are read by the high-speed reader, each relocatable binary produced is labeled $PTR.RB. Thus, the first relocatable binary is overwritten by the second.

Error Messages

The following error messages may be produced during assembly.

Message	Meaning
NO SOURCE FILE SPECIFIED.	No input source file was specified in the command line.
ILLEGAL FILE NAME.	A file name is illegal.
FILE DOES NOT EXIST.	An input source file does not exist.
FILE ALREADY EXISTS.	An output file already exists.
FILE WRITE PROTECTED.	An attempt has been made to write to a write-protected output file.
FILE READ PROTECTED.	An attempt has been made to read from a read-protected input file.
SWITCH ERRORS	The same file has been specified for both the listing and binary files.

Examples

The following command line assembles source file Z, producing the relocatable binary file Z.RB.

```
ASM Z ⟨CR⟩
```

This command line assembles file A, producing a listing file A.LS.

```
ASM/N/L A ⟨CR⟩
```

The following RDOS command assembles six files. It reads files A, B, C, and D from your current directory, file E from the fixed-head disk unit 0, and the sixth file (which has no name) from the paper tape reader. (The source program mounted on the reader must be reloaded, since the assembler requires two passes.) Binary relocatable files for each source file are output on the high-speed punch. Separate assembly listings are produced on the line printer.

```
ASM(A.B,C.D.DK0:E,$PTR) $PTP/B $LPT/L
```

RTOS Operation

To assemble one or more relocatable binaries, use the RDOS Macroassembler only.

To load and execute relocatable binaries under RTOS, follow the procedures given in Appendix B of the RTOS Manual, 093-000056. As described there, the relocatable binaries may be loaded using the RDOS relocatable loader.

Chapter 8

Macroassembler Operating Procedures

Assembler Files

Data General has supplied the macroassembler along with your operating system in the following files:

File	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 Data General 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] ⟨CR⟩
```

For microNOVA systems:

```
MAC/S MBID, OSID, NSID [,PARU,] ⟨CR⟩
```

For other NOVA systems:

```
MAC/S NBID, OSID systype, [,PARU] ⟨CR⟩
```

For ECLIPSE systems:

```
MAC/S NBID, OSID, NEID, [NCID]systype ↑ ⟨CR⟩
[,NFPID] [,PARU], ⟨CR⟩
```

To find systype, type LIST -DOS.SR ⟨CR⟩ from the master directory, and insert the name displayed. PARU is the system parameter file, which you will 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 is PARU.SR and OSID is OSID.SR.

The proper MAC command 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 would type the following commands from the nonmaster directory:

```
LINK MAC.SV       Dxx:MAC.SV ⟨CR⟩
LINK MACXR.SV     Dxx:MACXR.SV ⟨CR⟩
LINK MAC.PS       Dxx:MAC/PS ⟨CR⟩
```

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 nondisk devices, such as magnetic tapes or card readers, 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[*global switches...*]*filename*[*local switch...*] ⟨CR⟩

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 in order to execute it.

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.

Global Switches

You can include the following switches in the assembler command line:

Switch	Meaning
/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 B.
/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 which are 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.
/M	Flag multiple-definition errors on pass one. Normally MAC flags these errors only if they remain at the end of pass two.
/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).
/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 later in this chapter. under Macroassembler Symbol Table Files.
/T	Invoke the eight-character symbol feature. This instructs MAC to recognize symbol names of as many as 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; for example, .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 (for example, 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 in Chapter 11 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 will 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.

Switch	Meaning
/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

Switch	Meaning
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 ⟨CR⟩, and MAC A.SR B.SR C ⟨CR⟩

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 ⟨CR⟩

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 cannot 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 ⟨CR⟩

assembles PROG1, PROG2, PROG3, and PROG4 (see the CLI Reference Manual for other uses of ⟨⟩). TFILE will contain listing of the four files in the order they were assembled. The Macroassembler will insert either one or two form feeds between listing 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 listing 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 5).

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*$_n$ ⟨CR⟩

Then, rename the MAC.PS file to a useful name:

RENAME MAC.PS SYMBOLS3.PS ⟨CR⟩

Then, use the /T switch to specify your special file:

MAC FILEA SYMBOLS3.PS/T ⟨CR⟩

After each rename step, you 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 Appendix C.

Part II

Library File Editor

Chapter 9

Introduction to the Library File Editor

The Library File Editor (LFE) provides a means of updating and interpreting library files. A library file (also called a library) is composed of a set of relocatable binary files (produced by the Extended Assembler or by the Macroassembler) and is marked by special beginning and ending blocks. For example,

LIBRARY START BLOCK

prog$_1$.RB

.

.

.

prog$_n$.RB

LIBRARY END BLOCK

where each *prog*.RB represents one of a set of relocatable binary programs.

Library START and END blocks are described in Part III of this manual. Library tapes are supplied with the DGC operating systems and with subsystems such as ALGOL and FORTRAN.

With LFE, you can analyze the contents of a library file, list titles in a library file, merge or update libraries, extract logical records from a library file, and create new library files from existing system files or new material. The LFE is of special importance in ordering relocatable programs in a library file, since the relocatable loader uses this order to determine which programs will be loaded.

To be loaded, a program must have a global entry which resolves to an external declaration in a previously loaded program, or you must have included the LFE /F switch, which force-loads this library. This means that if program A on library file 1 has been loaded and contains a call to program B on library file 1, then B must follow A in the RLDR command line. If there are no unresolved external symbols defined as entries in the relocatable binary program and thus no calls to the program, the program is not loaded (unless you included the LFE /F switch).

In some cases it may be necessary to provide two or more copies of a given program on a library file to insure proper referencing. For example:

Program A calls   B calls   C calls   A

(Assume that C follows A in the library file.) If a previously loaded program has called A, then A, B and C are loaded by the standard mechanism. However, if a previously loaded program has called B, then only programs B and C would be loaded. In this case, a second copy of program A should be placed after program C. One of the LFE commands, Analyze (A), allows you to determine whether the programs on the file are the proper selection and in the correct order for your purposes, since the command creates a listing of global declarations of the library file.

All forms of the LFE operate in nearly identical fashion, their principal differences being the ways in which they communicate with the operator (error messages, operator cues, command format, and so on).

The terminology used to describe the relocatable binaries that make up a library is as follows:

Term	**Meaning**
logical record	A relocatable binary record contained within a library.
binary	Outside the library file, the relocatable binary may be a separate file called a binary. It may be produced by extracting one or more relocatable binary records from a library, or it can be the output of an assembly.
update	A binary that is to be inserted into a library, either to replace a current logical record or to create a new logical record.

Chapter 10

RDOS Library File Editor

The RDOS Library File Editor (LFE) is supplied with RDOS and DOS systems, or as a DUMP tape, and has the name LFE.SV.

The LFE works with conventional RB files and with extended RB files (produced by certain compilers and by the Macroassembler if global /T was specified to MAC).

Operation

You communicate with the LFE through the command line interpreter (CLI). When the CLI prints its ready prompt (R) on your console, you can enter an LFE command string. Error and caution messages are issued by the program on the console output.

Commands

Each command string starts with the CLI command LFE, followed by a single LFE command key letter. This key letter indicates what operation is to be performed on the arguments that follow the key letter. Key letters and the commands they represent are:

Key Letter	**Command**
A	Analyze a set of library files and/or binaries, or analyze selected records in a library.
D	Delete logical records from a library.
I	Insert binaries into either a new or an existing library.
M	Merge libraries to form a new library.
N	Create a new library from one or more binaries.
R	Replace logical records in a library with new binaries.
T	List titles in a set of libraries or binaries.
X	Extract specific logical records from a library.

Command String Structure

An LFE command string consists of LFE, followed by a command key letter, followed by arguments. Arguments can be filenames or logical record names. Arguments must be separated by at least one space; additional spaces are ignored.

An argument can have one or two switch options. A switch is indicated by a slash (/) following the argument; the slash is immediately followed by a letter or a number.

Each command string is terminated by a carriage return The command string structure is:

LFE *keyletter filename/switch recordname*

The following five rules apply to LFE command strings and to the commands involved.

1. Only one command key may be given for each command string.
2. An input library file and an update file cannot reside on the same device, for example, the $PTR. Both can, of course, be on disk.
3. An input file is searched for in the specified RDOS directory as *inputname*.LB; if not found, a search is made for *inputname*.
4. An update file is searched for in the specified RDOS directory as *filename*.RB (or .LB when using A, M or T command keys); if not found, a search is made for *filename*.
5. All references to logical records are satisfied by the first matching five-character record title in the library file. Therefore, each logical record on a file should have a unique title.

Command strings can be extended beyond one line by typing SHIFT-N (or SHIFT-6 on a DASHER) immediately before the carriage return (⟨CR⟩).

Switches

Arguments can be modified by switches. A switch is indicated by a right slash (/) followed by either a letter or a decimal digit. A blank space between the switch indicator (/) and the argument it modifies is optional. However, no space is permitted betweeen the slash and the letter or number following.

Letter Switches

Letter switches have distinct meanings that depend upon the arguments they modify and the command string in which they are found. All allowable letter switches will be explained in the descriptions of specific command functions. Arguments having switches /I (input library), /L (listing switch), /O (output switch), or /F (force-load switch) can be situated anywhere in the command string following the key letter.

Numeric Switches

Numeric switches specify the number of times that the previous argument is to be repeated. For example, $PTR/2 is equivalent to $PTR $PTR. A numeric argument of 1 (/1) has no effect. If two numeric switches are used, only the last one listed will be used. For example:

$PTR/3/2

will result in

$PTR $PTR

Command Descriptions

The following pages describe LFE commands. Optional switches and arguments are enclosed in square brackets in the command string format.

Analyze (A)

The A command itemizes the global declarations of a library file, of specific logical records within a library file, or of single relocatable binary records. Records are analyzed in the order of their appearance during the serial scanning of the input.

The A command produces the following output:

1. The name of the library, printed at the start of the analysis.
2. A listing of all global declarations (symbol, symbol type, and flags).
3. A cross-reference of all external records in the file called by each analyzed record.
4. The title of the module containing each external record referenced by the analyzed record.
5. A count of ZREL and NREL locations required by each analyzed record.

At the end of a library analysis a total count of all needed ZREL and NREL locations is given. (The total count given after a single binary analysis is the same as the count named in 5 above.)

Symbol types are:

CM	Named common symbol
ED	Entry displacement (a page zero entry)
EN	Entry normal (an entry outside page zero)
EO	Entry overlay
GD	GADD reference
GL	GLOC reference
GR	GREF reference
T	Title of record
XD	External displacement
XN	External normal

Each entry containing either a definition error or phase error is also flagged. Symbol flags are:

M	Multiply-defined entry. Symbol definitions must be unique in their first five characters. References to multiply-defined entry names are preceded by an asterisk (*).
U	Undefined entry which is referenced by an external normal or external displacement.
P	Phase error is an external normal or external displacement whose entry was defined before the external reference.

Format

LFE A [*listingname*/L] *inputname* [*recordname*]...

LFE A/M [*listingname*/L *inputname inputname*]...

Where:

listingname is the name of the file into which the analysis is to be written

inputname is the name of the file from which the input is read

recordname specifies a particular logical record in the input library to be analyzed.

Switches

/M	Multiple input library files. The switch modifies the command key A and causes all file names following, with the exception of a listing file, to be analyzed as one library.
/B	inputname is a binary. Default extension is .RB.
/F	Form feed. Each logical record analysis is on a separate page.
/L	Listing file. By default the analysis is listed on $LPT.

The switch assigns the preceding file of listing.

Examples

LFE A/M MATH1.LB MATH2.LB $LPT/L ⟨CR⟩

The library files MATH1.LB and MATH2.LB are analyzed as one library and the results are printed on the line printer. (Note that *recordname* may not be used with the M switch.)

LFE A DP1:M.LB ⟨CR⟩

The input file is M.LB in directory DP1.
All the logical records in this library file are analyzed and the results are printed on $LPT (default listing file).

LFE A MATH.LB SIN COS TAN $LPT/L ⟨CR⟩

The input file is MATH.LB.
The logical records SIN, COS, and TAN are analyzed and the results are printed on the line printer.

Output

The following is a sample of output generated by the A command.

```
AEL.LB

  T  RESID
  ED K10       OVL2
  ED K30       OVL2
  ED LDBT      OVL1     OVL2
  ED STBT      OVL1     OVL2
  ED SAVE      OVL1     OVL2
  ED RTRN      OVL1     OVL2
  EN WRLI
  EN WRIB
  EN RLIN
  EN RBIN
U GL LOCAT
  XN OVLA1     OVL1
  XN OVLA2     OVL2
U XN OVLA3
U XN RLOC
U XN BTAB
U XN BEND
U XN LOCAT

PAGE ZERO RELOCATABLE DATA = 000166
NORMAL RELOCATABLE DATA    = 001700
UNLABELED COMMON SIZE      = 000011

  T    OVL1
  EO   OVLA1   RESID
P XD   LDBT    RESID
P XD   STBT    RESID
U GD   LOCA2
P XN   SAVE    RESID
P XN   RTRN    RESID
U XN   LOCA2

PAGE ZERO RELOCATABLE DATA = 000000
NORMAL RELOCATABLE DATA    = 000336

  T    OVL2
  CM   CMAR
  ED   OVLA2   RESID
P XD   K10     RESID
P XD   K30     RESID
P XD   STBT    RESID
P XD   LDBT    RESID
P XN   SAVE    RESID
P XN   RTRN    RESID

PAGE ZERO RELOCATABLE DATA = 000000
NORMAL RELOCATABLE DATA    = 000322

TOTAL ZREL COUNT = 000166
TOTAL NREL COUNT = 002560
```

Delete (D)

The D command deletes one or more logical records from a library.

Format

LFE D *inputname outputname*/O *recordname* [*recordname*...]

where:

inputname is the name of the file containing the library whose selected logical recordname is to be deleted; *outputname* is the name of the device that produces the new library.

Switches

/O	Output master library file. This switch must always modify the name of the output library file, and can appear anywhere within the command line.

Example

```
LFE D $TTR ULTIL.LB/O MOVE LDBYT STBYT DIV
↑ ⟨CR⟩
        MULT COMP⟨CR⟩
```

The input file is $TTR.

The output file is UTIL.LB.

The logical records deleted from the input file are:

MOVE
LDBYT
STBYT
DIVI
MULT
COMP

Insert (I)

The I command merges update files and logical records on an input library file to produce an output library file.

By default, update files in the order listed in the command will be inserted before the first logical record in the input file. To insert an update file or files before or after a given logical record, use the /A or /B switches as described below. A given logical record may appear only once in a command.

No local symbols present in the update files are transferred to the output file.

Format

LFE I [*inputname*] *outputname*/O [*insertname*[/F]...]...
    [*recordname*/A *insertname*[/F]...]...
    [*recordname*/B *insertname*[/F]...

where:

inputname is the name of the file from which the existing file is taken.

outputname is the name of the file to contain the new library;

insertname is the name of a file from which binaries are taken for insertion;

recordname is the name of a logical record in the existing library before or after which insertions are to be made.

Switches

/O	Output library file. The switch modifies the name of the output library file.
/F	Force-load this binary. This sets the force-load flag in the binary; RLDR will load this binary whether or not it satisfies an external reference.
/A	Inserts after. The switch is used to insert arguments after the logical recordname which it modifies.
/B	Insert before. The switch appears after a logical recordname, and is used to insert arguments before this recordname.

Example

```
LFE I $PTR MATH.LB/O A.RB R.RB SINE/A C.RB D.RB
        COS/A X.RB Y.RB Z.RB ⟨CR⟩
```

The inputname is $PTR.

The outputname is MATH.LB.

Files A.RB and B.RB are inserted at the beginning of the output file.

Files C.RB and D.RB are inserted after the program SINE in the output file.

Files X.RB, Y.RB and Z.RB are inserted after the program COS in the output file (Note that SINE need not precede COS on the input file.)

Merge (M)

The M command combines existing libraries into one library

Format

LFE M *outputname*/O *inputname* [*inputname*...]

where:

outputname is the file to receive the new library

inputname is the name of a file from which a library is to be taken.

Switches

/O	Output library file switch, naming the file that is to receive the new library.

Example

```
LFE M FORT.LB/O FORT1.LB FORT2.LB FORT3.LB
↑⟨CR⟩
         FORT4.LB⟨CR⟩
```

The four FORTRAN library files are merged into a single FORTRAN library file called FORT.LB.

New (N)

The N command creates a new library file named *outputname* from one or more relocatable binary files.

Format

LFE N *outputname*/O *inputname* [*inputname*[/F]...]

where:

outputname is the file to receive the new library;

inputname is the name of the file containing a relocatable binary to be *included* in the new library.

Switches

/O	Output library file. This switch modifies the outputname file.
/F	Force-load this binary; set the force-load flag. RLDR will load this binary whether or not it satisfies an external reference.

Example

```
LFE N LIB3.LB/O A.RB B.RB/F C.RB ⟨CR⟩
```

The output is file LIB3.LB; it consists of binaries A.RB, B.RB, and C.RB.

B.RB has the force-load flag set; RLDR will always load it when LIB3.LB appears in the RLDR command line.

Replace (R)

The R command produces an output file, replacing logical records in the input file with relocatable binary update files.

No local symbols present in the update files are transferred to the output master.

Format

LFE R *inputname outputname*/O *recordname updatename* [/F]
[*recordname updatename*[/F] ...]

where:

inputname is the name of the file containing the existing library;

outputname is the name of the file that is to receive the new library;

recordname is the name of a logical record in the existing library that is to be replaced;

updatename is the name of the file from which a binary is to be taken.

Switches

/O	Output library file. The switch always modifies the outputname filename.
/F	Force-load this binary; set the force-load flag. RLDR will load this binary whether or not it satisfies an external reference.

Format

```
LFE R MATH.LB NEWMATH.LB/0 ATAN NEWATAN
TAN TAN.RB ACOS X.RB ⟨CR⟩
```

The input file is MATH.LB.

The output file is NEWMATH.LB.

Logical record ATAN is replaced by file NEWATAN.RB.

Logical record ACOS is replaced by file X.RB.

Logical record TAN is replaced by file TAN.RB.

Note that all these replacements will be made regardless of the order of the specified logical records on the input file.

Titles (T)

The T command produces a title (.TITL) listing of logical records of one or more input libraries.

Format

LFE T *inputname* [*outputname*/L] [*inputname*]...

where:

inputname is the name of the file containing an input library;

outputname is the name of the device that is to produce the listing.

Switches

/L	indicates the listing device. The listing device argument may appear anywhere in the command line after the function key T.

Example

```
LFE T $LPT/L $PTR F1.LB $TTR ⟨CR⟩
```

The library file is $PTR.

Additional library files are F1.LB and $TTR.

Titles are listed on the line printer.

Extract (X)

The X command extracts one or more logical records on a library file as separate relocatable binary files. The relocatable binary files will retain the names that they had as logical records.

This command cannot extract a record whose title (MAC's .TITL Pseudo-op) contains a period (.) character, unless it is the first character; then it is changed to a dollar sign ($). For example, .MAIN becomes $MAIN.RB.

Format

LFE X *inputname recordname* [*recordname*. .]

where:

inputname is the name of the file containing the library from which records are to be extracted;

recordname is the name of a record to be extracted from the input library file.

Switches

None.

Example

```
LFE X MATH.LB SINE COSIN TAN ⟨CR⟩
```

Library file MATH.LB is searched and the logical records SINE, COSIN and TAN are extracted, creating relocatable binary files SINE.RB, COSIN.RB, and TAN.RB.

Part III

Extended Relocatable Loaders

Chapter 11

Introduction to RDOS/DOS Relocatable Loader

RLDR Overview

The Relocatable Loader processes binary files (RBs), RB libraries, extended RBs, or extended RB libraries, and builds them into an executable save file on disk. The following Data General utility programs create RBs: The Extended Assembler (ASM), any compiler, and the Macroassembler (MAC). Certain compilers and MACs can generate extended RBs. You can create libraries from any RBs with the Library File Editor (LFE) utility.

You invoke the Loader with the CLI command RLDR. The Loader then scans your command line and builds the save file upward, including the RB and library modules in the command line. Each library is searched whenever its name appears in the command line. The system library (SYS.LB) is searched at the end of the command line and whenever you place its name in the command line. You can direct RLDR not to search SYS.LB at the end of the command line with the global /N switch.

NOTE: Because the system library (SYS.LB) differs for each type of system (unmapped or mapped), a program loaded under one system will probably not execute under another system. To load under one system for a different system, you must obtain the proper system library for the target system, and make sure that RLDR searches it, not the current system library, during the load. You can do this by performing the load process from a subdirectory which contains both the target system library and links to RLDR.SV and RLDR.OL (or the RLDR files themselves).

RLDR makes only one pass over the command line as it places the files specified one-by-one into the save file. It does not back up to re-scan files, or adjust locations already assigned.

By default, RLDR includes an entire module when you include the module name. You can instruct RLDR to include part of a module using the macroassembler .LMIT pseudo-op, described later in this chapter.

RLDR always builds certain system tables (User Status Table, Task Control Blocks, and so on) into the save file, starting at location 400_8 unless you are building a save file for execution in an unmapped foreground (local /F switch). See the local /F switch for details.

RLDR Files

RLDR uses two files to operate: RLDR.SV and RLDR.OL. It also scans SYS.LB during each load and copies task-processing modules into the save file. Normally, files RLDR.SV, and RLDR.OL, and SYS.LB are in the master directory (which holds the operating system); thus, you can always issue RLDR commands while operating in the master directory.

To use RLDR from another directory, get into the other directory and enter the command:

```
LINK RLDR.SV %MDIR%:RLDR.SV ⟨CR⟩

LINK RLDR.OL %MDIR%:RLDR.OL ⟨CR⟩

LINK SYS.LB %MDIR%:SYS.LB ⟨CR⟩
```

These commands create links to the original files, which enable you to operate RLDR from this directory. %MDIR% is a CLI variable which contains the master directory name; if any of the files RLDR uses are not in the master directory, type the full directory specifier name instead of %MDIR% for this file; for example,

```
LINK RLDR.SV DP1:RLDR.SV ⟨CR⟩
```

Command Line

The general formats of the RLDR command are:

RLDR binary$_1$...*binary*...*library*...*binary*

RLDR binary$_1$...*library*...*binary* [*ovname*, *ovname*...] *binary*...

where:

binary is any relocatable binary

... (ellipsis) means that you can repeat the preceding argument

library is any RB library built with the LFE utility

[indicates the start of an overlay node in memory

ovname is the name of an RB which you want to load as an overlay

, (comma) separates one ovname from another within square brackets

] indicates the end of an overlay node in memory.

Unless you specify another name with the local /S switch, RLDR names the save file $binary_1$, with the .SV extension; it names the overlay file (if any) $binary_1$ with the .OL extension.

Global Switches

Each global switch modifies RLDR's *operation* globally. You append each global switch to the command word RLDR. Local switches, described later, modify *arguments* in the RLDR command line. Following is a list of RLDR global switches and what they do.

/A Produces a second, alphabetical, symbol listing. RLDR always produces a list of global symbols as part of its load map; this switch tells it to produce a second, alphabetical list. You must also specify a device or disk file ($LPT/L, MAPFILE/L) with the local /L switch to receive the listing. For example:

```
RLDR/A MYPROG MYPROG.AL/L ⟨CR⟩
```

/B Uses short Task Control Blocks (words in unmapped NOVA multitask programs only). In unmapped NOVA multitask programs, the last four words of each Task Control Block (TCB) are unused. You can specify short (13 word) TCBs with this switch. For single-task programs, RLDR ignores global /B.

/C Creates this file for an RTOS environment. A save file created with global /C cannot execute under RDOS or DOS.

When you use this switch, RLDR starts NREL code at 440_8 (plus the length of the RTOS overlay directory if any), inserts the program's starting address in USTSA, starts the file at location zero, and does not search the system library, SYS.LB. For example:

```
RLDR/C/Z RTOSPROG PROG2 PROG3 ⟨CR⟩
```

/D Includes a debugger and symbol table in the save file This switch places an external DEBUG in RLDR's symbol table; by default, the symbolic debugger is loaded to the save file, it also builds a symbol table into the save file.

The symbol table is needed to help you debug or edit the save file. Unless you include global /S, RLDR places the symbol table immediately above your program. To place the symbol table in high memory, see global /S.

NOTE: You can include a symbol table without the debugger by including an .EXTN .SYM. statement in one of the modules to be loaded.

To copy the interrupt-disable debugger instead of the standard debugger, use a global /D and insert the name xIDEB.RB somewhere in the command line before SYS.LB is searched. (To find xIDEB, get into the master directory and type LIST - IDEB.RB. The name returned is the version of IDEB for your operating system.)

NOTE: Unlike the rest of the save file, RLDR builds the program's symbol table downward (from high addresses to low) in the save file. Thus, when you are examining a symbol table, the normal order of symbols is reversed. The symbols are three words long; the first two words contain the symbol name (in radix 50) and the third word is the value of the symbol.

/E Displays error messages when a listing file has been specified (local /L). Normally, when you specify a listing file with local /L, RLDR does not send error messages to the console; instead, it sends these messages to the listing file. Use this switch when you want error messages sent to both the listing file and the console.

/G Prints labeled common warning messages whenever new labeled common appears in overlays. Normally, RLDR prints each warning message only once for a load. Labeled common in overlays is described later in this chapter. Also, see global /R.

/H Prints all numeric output in hexadecimal. Normally, RLDR prints this output in octal.

/I Creates absolute data files or absolute data programs. Does not create a UST, TCB, or other system tables. Starts NREL code at 445_8 and ZREL code at 50_8. To start ZREL at an address other than 50_8, you must use the local /Z. Use the global /Z to start the save file at 0. Programs created with /I cannot execute under any operating system.

/K	Keeps the special RLDR symbol table file in the disk file named binary$_1$.ST. Normally, RLDR deletes this file after the load. This file is described later in this chapter.
/M	Suppresses all RLDR output to the console. This speeds loading on slow systems, but you should use it carefully since it also suppresses error messages
/N	Does not search the system library, SYS.LB, unless its name appears in the command line. Normally, RLDR searches the system library at the end of the command line, to try to resolve undefined symbols.
/O	Omits the program symbol table even though global /D is used.
/P	Prints the starting NREL address of each RB or library module along with its title as it is loaded. When you include this switch, and the first module loaded specifies a number of tasks in a .COMM TASK statement, and you omit local /K, the starting address of the .COMM .TASK module is reported incorrectly (without adjustment for multiple tasks).
/R	Places new overlay common in the root program. If an overlay in this command line declares a new named common symbol (.COMM pseudo-op), places this common in the root portion (which is always memory-resident during execution). Normally, RLDR places new overlay common in the overlay node, where it will be overwritten by the next overlay loaded into this node. NOTE: RLDR cannot load virtual overlays (local /V) if you use global /R.
/S	Places the program's symbol table in high memory (used with global /D). Normally, RLDR places the symbol table directly above the program. The /S switch instructs RLDR to place the table as high as possible in memory (beneath the operating system).
/U	Does not resolve undefined symbols to −1. Normally, the memory locations which reference an undefined symbol are chained to one another. If the symbol is never defined, the address chain remains intact when you include the global /U switch.
/X	Allows up to 128 system overlays. The SYSGEN program uses this switch when it instructs RLDR to generate a new operating system.
/Y	Allows up to 256 system overlays. See /X above.
/Z	Starts the save file at location zero. Such a file cannot execute under RDOS or DOS, but can execute under RTOS. After loading, you must process the save file with the MKABS/Z command; you can then execute the file with the binary loader or BOOT (see the BOOT command in the RDOS/DOS Command Line Interpreter [DGC No. 069-400015]).

Local Switches

A local switch modifies an argument in the RLDR command line. The local switches are.

n/C	Allots *n* (octal) I/O channels to the program Each file or device the program uses must be opened on a channel. You can also specify channel (and task) information in a .COMM TASK statement: if there is a .COMM TASK, the /C specification overrides it. If you omit both /C and .COMM TASK, RLDR allots eight channels to the program.
name/E	Sends error messages to file *name* when a listing file has been specified (local /L). You must also include global /E. For example: RLDR/E MYPROGRAM $LPT/L $TTO1/E ⟨CR⟩ Error messages go to the second console (not the default console, $TTO): the load map goes to the line printer.
n/F	For execution in an unmapped foreground only. Starts this save file's system tables at octal address *n*. You must use this switch, and local /Z, to allow this program to execute in an unmapped foreground. If *n* is not an integer multiple of $(400_8) + 16$, RLDR will round it to the next multiple. For example: RLDR MYPROG 20000/F 200/Z MYPROGFG/S ⟨CR⟩ This command builds a save file name MYPROGFG.SV from the binary MYPROG. MYPROGFG's system table begins at 20016_8.

n/K

Allots *n* (octal) TCBs (tasks) to this program during execution. You can also specify tasks in a COMM TASK statement; if so, the /K specification overrides the .COMM TASK. If you omit both .COMM TASK and /K, RLDR assigns one TCB (task) and loads the single-task scheduler with the program.

NOTE: You must specify multiple tasks for a multitask program (one which uses the .TASK or .QTSK task calls).

name/L

Sends the load map and all errors to file *name*. If *name* is a disk file name, RLDR creates a file and writes the load map and any errors to this file. If *name* exists, the map and errors are appended to the file. Unless you also specify global /E, errors are not displayed on the console (instead, they will go to the file or device).

The load map for any program can help you patch it with the patch utilities. For example:

```
RLDR MYPROG PROGA PROGB
MYPROG.LM/L (CR)
```

n/N

Starts loading NREL code from the next module (in the command line) at octal address *n*. Normally, RLDR builds the save file upward from lower memory, placing each module directly above the last module loaded. It maintains an NREL pointer to keep track of the next available location in NREL space. Local /N moves the pointer upward. Address *n* must be greater than the current NREL pointer value (higher than the last location loaded). If *n* is not higher than the last address loaded, RLDR ignores the switch and continues loading normally. For example:

```
RLDR MYPROG 4000/N PROGA
6000/N PROGB (CR)
```

MYPROG's NREL code starts normally, above the system tables; PROGA's NREL starts at 4000_8, and PROGB's NREL code starts at 6000_8.

name/S

Gives the save *filename*. Normally, the CLI gives the save file the name of the first binary in the command line, with the .SV extension, deleting any existing save file with the same filename. The /S switch names the save file *name*.SV, and the overlay file (if any) *name*.OL. An existing save (and overlay) file with the default (first binary) *name* is left intact. For example:

```
RLDR MYPROG [OVLY0, OVLY1]
MYPROGXXX/S (CR)
```

This produces the save file MYPROGXXX.SV and the overlay file MYPROGXXX.OL.

NOTE: Local /S does not instruct RLDR to load anything: it simply gives the save file a name other than that which it would have received normally.

/U

Include local (user) symbols in the save file. Normally, symbols which are not specified by a .ENT pseudo-op are not retained. If you specify global /U to the assembler, and local /U to the RLDR, these symbols are placed in the program's symbol table. (You must also specify RLDR global /D to include the symbol table.) Local symbols can help you debug or edit the program. For example:

```
MAC/U (FILEA, OVLY0, OVLY1)
.
R
RLDR/D FILEA/U [OVL60/U, OVLY1/U]
```

This assembles three files separately, including user symbols; then it includes user symbols in the program symbol table via RLDR. Local overlay symbols are also included.

[overlays]/V	Loads overlays as virtual overlays (mapped RDOS only). Virtual overlays occupy mapped extended address space during program execution: they are further described in the RDOS System Reference (DGC No. 093-400027). Virtual overlays have meaning only in a mapped RDOS system, although RLDR will build them on any system. RLDR allots node space for virtual overlays in multiples of 1.024 (2000_8) words and aligns the beginning of each virtual node with a 1K boundary from location zero. All virtual overlays must precede conventional overlays in the RLDR command line. For example: RLDR MYPROG [OV0, OV1, OV2]/V [OVA, OVB, OVC]
n/Z	Start loading ZREL code from the next module (in the command line) at octal address *n*.

Library Files

You can enter the name of a library (built with the LFE utility) anywhere in the RLDR command line. A library module can be part of the save file, or it can be in an overlay, within [,,]. RLDR will scan the libraries you include, but will not load any modules from the library unless one of two conditions exists: either the module satisfies an unresolved symbol from another module, or you specified a force load when you build the library. When a module is loaded because of the first condition, any symbols satisfied by that module receive defined status. After a symbol is defined, it can no longer force a module to be loaded.

Loading Overlays

You can specify an overlay node anywhere within the save file. The node receives the overlays specified within the square brackets one-by-one when the program executes. The overlays (*ovnames*) that you specify are placed within the overlay file. The node in the save file is vacant during program execution, until the save file loads (with the .OVLOD or .TOVLD system calls) an overlay into the node. When the program is finished with this overlay, it can then load another from the overlay file into this node.

The square brackets established an overlay node and associate a group of overlays with that node. Within the square brackets, you must use a comma to separate one overlay from another. Any overlay can consist of one or more RB files. For example:

```
RLDR MYPROG [OV1 OV2, OV3, OV4] <CR>
```

This command line creates save file MYPROG SV and overlay file MYPROG.OL. MYPROG.SV has one overlay node to receive the overlays in MYPROG.OL one at a time The three overlays in MYPROG.OL are OV1 and OV2. OV3, and OV4 because the first overlay consists of binaries OV1 and OV2.

When RLDR encounters square brackets, it scans each binary within the set to determine the size of the largest overlay It then rounds the size of the largest overlay to the next multiple of 256 (400_8) words, and allots this size to this node. (It rounds virtual overlays–described under the local /V switch–to the next multiple of 1.024 words) By this procedure, RLDR ensures that the node will be large enough for the largest overlay Then it pads each overlay with zeroes to the node size and copies the overlays to the overlay file

In the preceding command line, if RBs OV1, OV2, OV3, and OV4 are 10, 15, 20, and 40 words respectively, RLDR uses the length of OV4 (40), and rounds it up to the next multiple of 256 words: thus, 256 words is the size of the node. If the sizes were 200, 60, 290 and 90 words instead. RLDR would use the length of the first overlay (OV2 with 200 words, plus OV2 with 60 words) and would round 260 up to the next multiple of 256. This is 512: thus the node would be 512 words long.

The save file can reference overlays by name if you named each overlay with the .ENTO pseudo-op; if you omitted .ENTO, the save file must access each overlay by node number and overlay number Consult your system reference manual for more on overlays.

Overlays and Named Common

Named common is code that you can specify with the .COMM pseudo-op or a higher-level language equivalent. When a module which is not an overlay declares named common for the first time, RLDR places the common in the program save file. During program execution, this common remains in memory and any module can use it.

Overlays That Declare Named Common

If you omit the global /R switch and an overlay declares new common, RLDR places this common in the overlay itself. This means that the program can use this common only while the overlay is resident

If you include the global /R switch, RLDR tries to place, within the root program, all named common. Only the first overlay in each pair of square brackets can declare new common when you use global /R. RLDR takes the named common from the overlay and adds it to the root node. This moves the starting address of the overlay node upward. RLDR cannot update the load map after moving the node upward, so, if overlays declare new common and you include global /R, the load map's node start figure will be wrong. To indicate the incorrect node start address, the map contains the warning message

LABELED COMMON IS ROOTBOUND

Summary

When RLDR encounters a named common symbol in an overlay, it takes the following steps To illustrate, we will call the named common symbol .COMM. and its size SIZE. The command line might be·

RLDR PROGA [A.B.C] ⟨CR⟩

where module A declares COMM.

If you omit global /R, RLDR checks its symbol table for COMM If COMM is new, RLDR allots space for it within the overlay node If COMM was declared earlier (possibly by module PROGA), and has received a memory location, RLDR checks that location and other parameters as follows:

- If the assigned memory location of COMM is less than the node start location, RLDR displays the warning message

LABELED COMMON IN NODE IS DEFINED OUTSIDE NODE

and continues. RLDR assumes that it has already allotted space, it displays the message to warn you that the labeled common may be defined in another overlay–outside this node–and thus may be unavailable to this node's overlays.

- If COMM's assigned memory location is the next available location, RLDR increments the next available location by SIZE, and continues.
- If the assigned memory location of COMM is greater than the node start, and this location plus SIZE is less than the next available location, RLDR continues.
- If the assigned memory location of COMM is greater than node start, and this location plus SIZE is greater than the next available location, RLDR displays the message

ALIGNMENT ERROR

and aborts the command.

If you include global /R, when RLDR encounters any new named common in the first module of a node, it places COMM in the root program (PROGA.SV) and writes the LABELED COMMON warning message to the load map Then it moves the node start upward by the SIZE of COMM. The module which declares COMM must be the first within this set of square brackets (in the example, module A), because RLDR cannot adjust a node starting address after it has loaded data.

The Load Map

The load map of RLDR has two parts: a memory map and a list of symbols When you specify global /A, it has a third part· an alphabetical list of symbols. For example, the command line

RLDR ROOT0 [OVL00, OVL01] ROOT1 [OVL10, OVL11]

produces a load map that looks like the following (RLDR does not print the line numbers; we have included them for clarity):

```
0     ROOT0
1.              001470
2               000,000    OVL00      000004
3               000,001    OVL01      000004
4               0002070
5     ROOT1
6               002276
7               001,000    OVL10      001006
8               002,002    OVL11      000775
9               003676
10    TMIN
11.   NSAC3
12    NO STARTING ADDRESS FOR LOAD MODULE
13    NMAX      003752
14    ZMAX      000050
15    CSZE      000000
16.   EST       000000
17.   SST       000000
18.   USTAD     000400
19    START     000452
20.   LOV10     000453
```

In this load map, lines 0 and 5 contain the program RB titles. If the command line contains a user library, the titles of modules loaded from the library are also printed. RLDR prints library module names when it encounters the module's TITLE block. On an error, it prints an error message after the module title.

Lines 1 and 4 in the map show starting and ending addresses for the first overlay node (node 0); lines 6 and 9 show these addresses for the second node (node 1). Remember that RLDR allots node space for conventional overlays in multiples of 400_8 words, thus giving addresses 1470 through 2070 to the first node, and 2276 through 3676 to the second node. Each node is large enough to accept the largest overlay associated with it.

Lines 2, 3, 7 and 8 describe the overlays for node 0 and 1, respectively. You can interpret line 2 as follows:

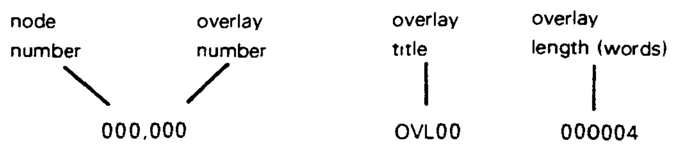


Figure 11.1 Interpretation of line 2 DG-25179

Lines 10 and 11 contain the names of the single-task scheduler (TMIN) and module NSAC3, taken from SYS.LB. These names are explained in the Example Program section later in this chapter

At this point in the map, RLDR gives error flags and prints other error messages If there is an undefined or multiply-defined symbol, or named common, in the load map, one of the following codes, followed by the symbol name, are displayed·

Code	Meaning
XD	External displacement (EXTD) undefined.
XN	External normal (.EXTN) undefined.
C	Named common symbol
M	Multiply-defined symbol

There are no RLDR flags in this load map, but line 12 contains an error message. This message means that neither ROOT0 nor ROOT1 specified a starting address after a .END pseudo-op You will have to correct this in the source, then reassemble and reload the program, before the program runs correctly

NMAX, in line 13, is the lowest available address during execution; it is one location higher than the highest address used by the program ZMAX-1, line 14, is the highest page zero (ZREL) address used by the program; by default RLDR starts ZREL at location 50_8. In this program there is no ZREL code, thus ZMAX is 50_8 CZSE, line 15, is the size of the named common area; in this program, there is no such area. EST and SST are the end and start of the program symbol table (global /D switch); again, in this program, there is no symbol table.

Line 18 describes the starting address of the program User Status Table; this begins the system tables described later. Lines 19, 20, and so on identify symbols used in the modules.

System Tables

Normally, RLDR builds a user status table (UST) and one or more task control blocks (TCBs) into each save file. It also builds an overlay directory into the save file if the command line specified overlays. The UST usually extends from location 400_8 to 423_8. Generally, each TCB occupies 21_8 words. The size of the overlay directory varies with the number of overlay nodes. Figure 11.2 shows the arrangement of these tables in the save file:

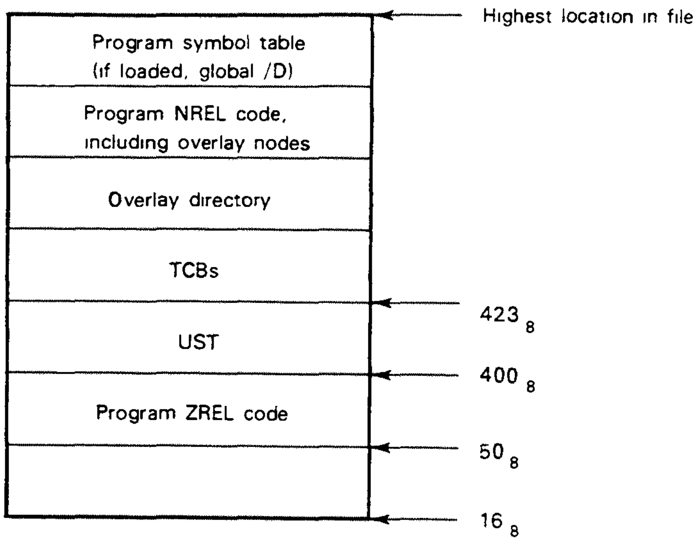


Figure 11.2 System tables in a save file ID-00123

The system uses locations 0 through 15_8

User Status Table (UST)

The user status table records execution and runtime information on the program. RLDR builds it upward from location 400_8 to 423_8. Locations 400 through 437 contain the UST, as shown here The RDOS System Reference (DGC No. 093-400026) describes the UST further

```
USTPC  = 0
USTZM  = 1         ,ZMAX
USTSS  = 2         ;START OF SYMBOL TABLE
USTES  = 3         ;END OF SYMBOL TABLE
USTNM  = 4         ;NMAX
USTSA  = 5         ;STARTING ADDRESS
USTDA  = 6         ;DEBUGGER ADDRESS
USTHU  = 7         ;HIGHEST ADDRESS USED
USTCS  = 10        ;FORTRAN COMMON AREA
                   ;SIZE
USTIT  = 11        ,INTERRUPT ADDRESS
USTBR  = 12        ;BREAK ADDRESS
USTCH  = 13        ;NUMBER OF CHANNELS/
                   ,TASKS
USTCT  = 14        ;CURRENTLY ACTIVE TCB
USTAC  = 15        ,START OF ACTIVE TCB CHAIN
USTFC  = 16        ,START OF FREE TCB CHAIN
USTIN  = 17        ;INITIAL START OF NREL CODE
USTOD  = 20        ,OVERLAY DIRECTORY
                   ,ADDRESS
USTSV  = 21        ;FORTRAN STATE VARIABLE
                   ;SAVE ROUTINE
USTEN  = USTSV     ,LAST ENTRY
```

Task Control Blocks

Each task control block (TCB) records runtime information on a program task. RLDR builds one TCB into the program for each task that you specify You can compute the number of words for the TCB area with the formula:

(*number-of-tasks*) * TLN

Where. entry TLN is the number of words for each task control block; generally, this is 21_8. The system parameter file, PARU.SR, describes TLN; see PARU.SR cross-reference in the RDOS System Reference. That manual contains additional information task control blocks.

Overlay Directory

The overlay directory maintains information on each overlay node RLDR creates the overlay directory only if your command line specifies overlays. The directory length is:

(4*number-of-nodes*) + 1

For more information on the overlay directory, consult the RDOS System Reference

Memory Map Illustration

The following illustration shows how file ROOT.SV looks in main memory during execution. We showed the command line and load map for this file in the Load Map section earlier Assume that the program has loaded (.OVLOD) an overlay into each node.

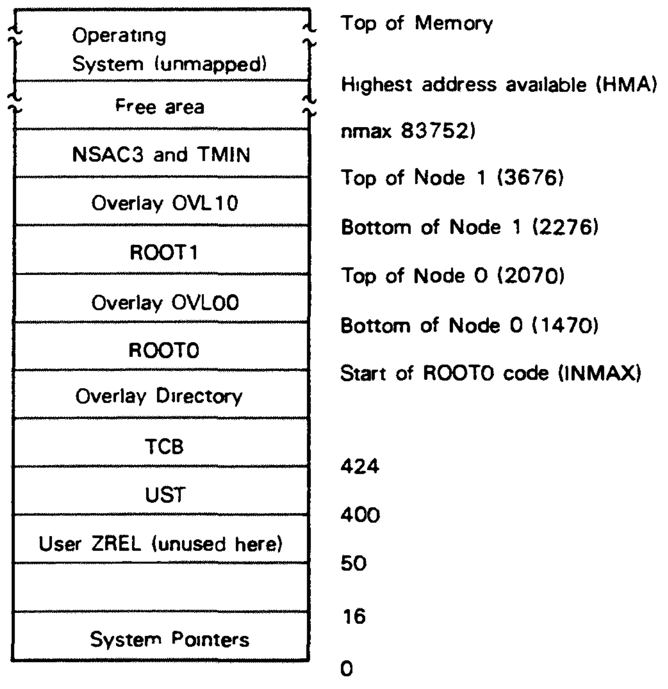


Figure 11.3 Memory Map ID-00124

A mapped RDOS system is not at the top of memory. However, you can ignore the position of a mapped system because it is logically isolated from user space, and invisible to user programs. If you have a mapped system, ignore the position shown for the system in the rest of this chapter

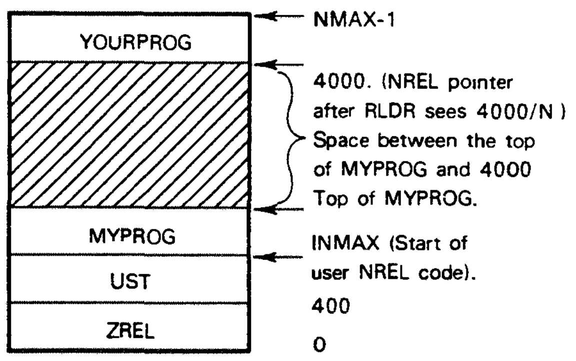
The highest memory address available to user programs is called high memory address (HMA) INMAX is the start of normal relocatable code (NREL). (Figure 11.3 is not to scale.)

Specifying NREL Address

As described under the local /N switch, you can increment the RLDR's NREL pointer to start loading a file at a given address. Naturally, the value you specify with local /N must exceed the last address loaded. For example, the command line.

```
RLDR MYPROG 4000/N YOURPROG
```

instructs RLDR to build MYPROG this way in memory before writing it to disk:



YOURPROG
MYPROG
UST
ZREL
NMAX-1
4000. (NREL pointer after RLDR sees 4000/N)
Space between the top of MYPROG and 4000
Top of MYPROG.
INMAX (Start of user NREL code).
400
0

Figure 11.4 How RLDR builds MYPROG in memory ID-00125

Any program can change its NMAX during execution with the .MEMI system call; this is very important for a swap or a chain. In a swap, one program is written to disk, and another is put in its place. the first can be called back. In a chain, program one calls program two, and program one is lost.

The .LMIT Feature

You can instruct RLDR to load a module partially with the Macroassembler's .LMIT pseudo-op. To use this pseudo-op, declare a .LMIT symbol in a module, then use the symbol in the module that you want partially loaded. This module must enter the symbol. Only the first module which enters and uses the symbol is partially loaded. If you place the .LMIT symbol in ZREL, all of the module's NREL code and its ZREL code up to the symbol is loaded If the symbol is in NREL, RLDR loads all of the ZREL code, as well as the NREL code up to the symbol. In the RLDR command line, you must enter the module that establishes the .LMIT symbol before entering the module that you want partially loaded. You cannot limit-load more than one module with a single .LMIT symbol. If the same .LMIT symbol occurs as an entry in a module which follows the partially-bound module, RLDR returns a multiple-definition error.

If a module to be partially loaded defines entries after the .LMIT symbol, RLDR sets the value of these symbols to −1. For example:

Module A	**Module B**	**Module C**
.TITL A	.TITL B	.TITL C
.	.	.
.LMIT LIM1	.ENT ALPHA, ROUT1	.
.EXTN ROUT1	.ENT LIM1	.
.	.	.
.	LIM1:LDA....	
.	ROUT1:....	.
	ALPHA:....	.

After assembling each module, type

```
RLDR A B C (CR)
```

The limit symbol, LIM1, instructs RLDR to load module B partially. RLDR finds that ROUT1, which A declares external, is defined in B after the limit symbol; therefore, RLDR stores the value −1 for ROUT2 in its symbol table. The utility then proceeds to load all of A and C, and B– up to symbol LIM1. If module C mentions ROUT1, RLDR inserts −1 whenever each ROUT1 symbol occurs; if C mentions ALPHA, RLDR returns the error

```
UNDEFINED SYMBOL
```

For the .LMIT pseudo-op to work properly, you must insert entries in the proper order. Normally, you define entries in logical order, but since MAC reverses their order during assembly, you must declare them in reverse order. In the example above, .ENT ALPHA is not entered in the symbol table at all. If the entries in module B were reversed (LIM1, ROUT1, ALPHA), .ENT ROUT1 and ALPHA would receive NREL values in the symbol table before RLDR realized that module B was limited.

Symbol Tables

RLDR creates two symbol tables for its own use whenever it builds a .SV file. The first table resides only in memory and contains undefined symbols. The second table, a disk file named *savefilename*.ST, contains both undefined and defined symbols. As RLDR resolves symbols, it removes them from the memory-resident table and marks them in *savefilename*.ST as defined. As it processes the command line, it continues resolving symbols in the memory-resident table and marks them in *savefilename*.ST. The memory-resident table grows as RLDR encounters external symbols, then shrinks as it resolves them. At the end of an error-free load, all symbols are resolved in *savefilename*.ST and RLDR then deletes *savefilename*.ST (unless you used the global /K switch in the command line).

If the memory-resident table grows too large for available memory, RLDR displays an overflow error message and aborts the command. You can often solve the overflow problem by reducing the number of symbols that are undefined at one time. Do this by rearranging the order of files in the command line.

There is a third symbol table which differs from the tables RLDR creates for its own use. This is the table you can have inserted in the save file by appending the global /D switch, which loads a debugger. For clarity, we will call this the program symbol table. The program symbol table is most useful for debugging and editing the save file, especially if you included local symbols when you assembled and loaded the file. You can include the program symbol table *without* including a debugger by inserting a .EXTN .SYM. statement in one of the program modules.

At present, RLDR truncates any symbol of more than five characters to five characters before placing it in the program symbol table. If the program contains extended RBs (as generated by the MAC global /T switch and certain compilers), duplicate symbols may exist in the program symbol table. This can present debugging problems for users of extended RBs.

The position of the program symbol table in the file varies according to the switches you specify in the RLDR line. The following illustrations show how different configurations of a sample program (MYPROG.SV) would look in memory during execution.

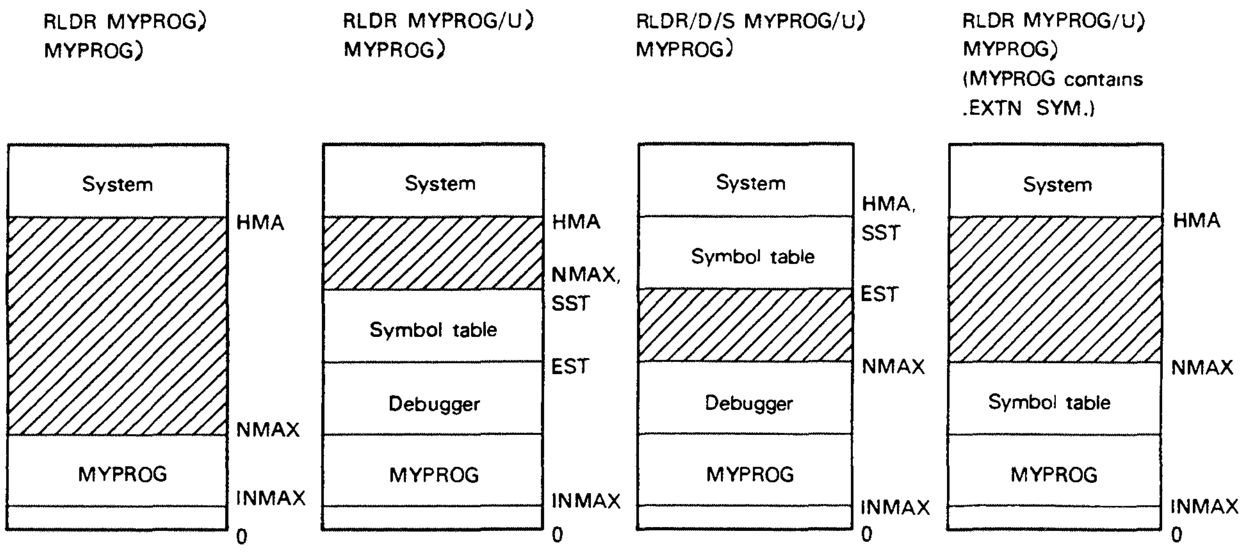


Figure 11.5 Program configuration in memory

ID-00126

Aside from the position of the system, these diagrams apply to both mapped systems and background programs in unmapped systems.

Symbol Table Formats

There are two symbol tables which RLDR builds for its own use. The first table is the .ST file on disk, which contains an entry for both defined and undefined symbols. The second table remains in memory and contains only undefined symbols. Formats for these RLDR tables follow.

The third symbol table, which you can have written to the save file by including the global /D switch (or .EXTN .SYM. in a program module), is described in Chapter 1, under The Symbol Table (Loader Map).

Disk Table Format For Symbols

Word 0	Symbol type in left byte, length of name in words in right.
Word 1	Symbol equivalence.
Word 2	For ENTOs: overlay node/number word.
	For ENTs: node/overlay word of overlay where symbol is defined.
	For COMMs: named common size.
Word 3 to Word n	symbol name in ASCII, packed in ASCII, 2 character per word. 1st in left, 2nd in right, etc.

Word 1 of an ENTO is not a memory address; it is the overlay node/number word (same as word 2 for .ENTs).

Memory Table Format for Symbols (Undefined Only)

Undefined symbols are represented in memory as well as on disk.

Word 0	Block number and channel bits in left byte.
Word 1	Offset in left byte, type in right byte. Offset is offset of symbol in disk symbol block. Type designations are as shown in Appendix D. Bit 13 is flag for extended format.
Word 2-n	Chain pointers

Checking Save File Size

Because RLDR loads the save file directly onto the disk, you can load a file that is too large to execute in user space. If you try to execute such a file, you will receive the message

INSUFFICIENT MEMORY TO EXECUTE PROGRAM

You can check for overflow by loading a symbol table into high memory with the RLDR global /D and /S switches. RLDR builds the symbol table down from the highest address available. If the symbol table is going to overwrite the program code during execution, RLDR displays the message:

SYMBOL TABLE TOO LARGE FOR CORE STORAGE

This does not necessarily mean that the program will not fit into memory, because the debugger and symbol table were also loaded, and they require significant space If you do not receive the error message, you can be sure that the program fits into memory, and you can then issue the RLDR command needed for this program

Command Line Examples

This section shows two RLDR command lines, and memory maps of the save files that they produce The first example loads a multitasking program, with program symbol table and debugger, the second loads a program for execution in an unmapped foreground.

```
RLDR/D PROGA DATA0 [COMPA, COMPB, COMPC,
COMPD] ↑ (CR)
    DATA1 [MULT1, MULT2, MULT4] 14/C 6/K (CR)
```

This command line builds a save file named PROGA SV and a corresponding overlay file named PROGA.OL. When PROGA executes (after it .TOVLDs COMPA into node 0 and MULT11 into node 1), the memory appears as in Figure 11 6.

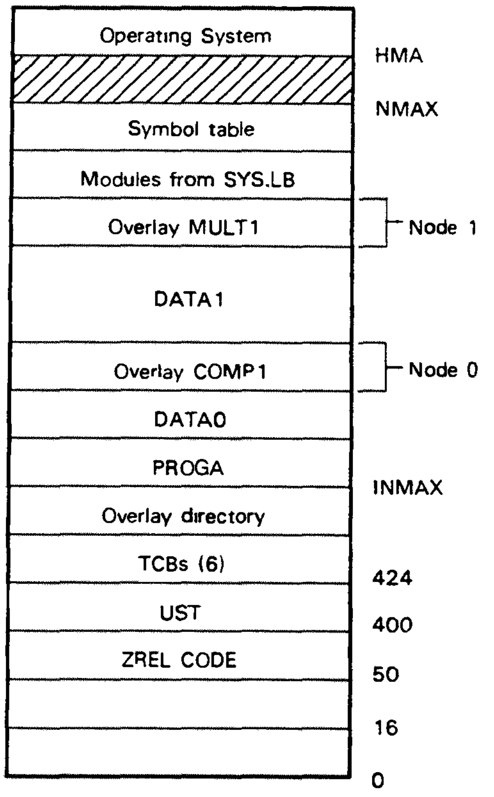


Figure 11.6 Memory map (multitasking) ID-00127

In a mapped RDOS system, the operating system would not be at the top of memory, but it would be invisible to your programs, so the example is useful, nonetheless. PROGA would have the same structure if it ran in the foreground, but addresses 0 through HMA would be foreground locations This does not affect execution

```
RLDR MYPROG 300/Z 24000/F MYPROGFG/S (CR)
```

This command line builds a save file named MYPROGFG.SV for execution in an unmapped RDOS foreground The original background version (MYPROG SV) remains intact and can be executed in the background at any time

This example assumes that the CLI continues running in the background after MYPROG.SV is executed with the command

```
EXFG MYPROGFG (CR)
```

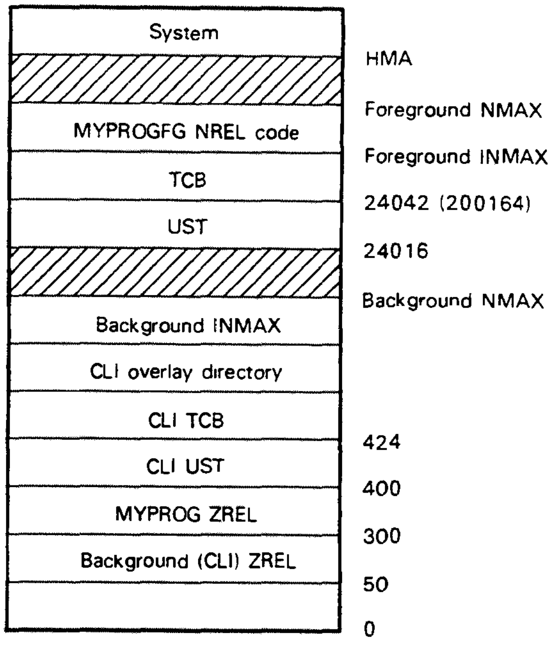


Figure 11.7 Memory map (unmapped foreground) ID-00128

Sample Program (ASM and MAC)

The following listings show a simple program named ROOT, which uses two overlays, OVLY0, and OVLY1. An RLDR load map and explanation follow the assembly listings. The command line that assembled the source file was:

```
MAC/L (ROOT, OVLY0, OVLY1)
```

or

```
ASM/L (ROOT, OVLY0, OVLY1)
```

depending upon whether you use the Assembler or the Macroassembler. Each produces the same result. Either command line assembled the three source files separately, producing ROOT.RB, OVLY0.RB, and OVLY1.RB, and listing files ROOT.LS, OVLY0.LS, and OVLY1.LS.

Although ROOT, OVLY0, and OVLY1 are written in assembly language, compiler users may note some similarities between their compiler commands and operating system calls. For example, FORTRAN commands CALL OVOPN and CALL OVLOD correspond roughly to system calls .OVOPN and .OVLOD. Generally, if you are using a compiler, you will want to skip to the RLDR load map. This will be more familiar and instructive than the program listings.

In the listings for OVLY0 and OVLY1, note the .ENTO pseudo-op. ROOT declares each overlay external by name, and .ENTO allows ROOT to access each overlay by name.

```
 0001    OVLYO MACRO REV XX XX            09:01:31 07/25/xx
                              .TITL OVLYO
02                            .ENTO OVLYO
03                            .ENT PRNTB
04                            .EXTN ER. LOV1
05          000001             TXTM 1
06                            .NREL    ;Use NREL for each overlay.
07
08 00000 020407 PRNTB:        LDA 0.B  ;Get the "B".
09 00001'006017               .SYSSTM  ;Print
10 00D02 020000               .PCHAR   ;it.
11 00003'002403               JMP @.+3 ;Jump to ER.
12 00004'002401               JMP @.+1 ;Normal return--
13 00005'077777               LOV1     ;to ROOT.
14 00006'077777               ER       ;Back to "ER" in ROOT.
15 00007'D00102 B:            "B"
16                            .END

**00000 TOTAL ERRORS.   00000 PASS 1 ERRORS


 0001 OVLY1   MACRO REV 06.20             09:02:02  07/25/77
                              .TITL OVLY1
02                            .ENTO OVLY1
03                            .ENT PRNTC
04                            .EXTN ER. RTURN
05                            .NREL    ;NREL for each overlay.
06
07 00000'020407     PRNTC:  LDA 0.0 ;Get the "C".
08 00001'006017               .SYSTM   :Print
09 00002'010000               .PCHAR   ;it.
10 00003'002403               JMP @.+3 ;Error goes to "ER".
11 00004'002301               JMP @.+1 ;GO TO
12 00005'077777               RTURN    ;"RTURN" in ROOT.
13 00006'077777               ER
14 00006'000103 C:            "C
15                            .END

**00000 TOTAL ERRORS.   00000 PASS 1 ERRORS


 0001    ROOT  MACRO REV 6.20             09:02:02  07/25/77
                            .TITL ROOT
02                          .ENT START. LOVO. LOV1. RTURN. ER
03                          .EXTN OVLYO. PRNTB. OVLY1. PRNTC
04        000001            .TXTM 1
05                          .NREL         ;NREL code.
06
07
08 00000'020440 START:    LDA 0. OFILE :Get overlay filename
09 00001'126400           SUB 1.1      ;Default mask.
10 00002'006017           .SYSTM       :Open overlay file
11 00003'012000           .OVOPN 0     ;on I/O channel 0.
12 00004'000424           JMP ER    :Error return
13 00005'020426           LDA 0.A   ;GET the "A".
14
15 00006'006017           .SYSTM    :Print
16 00007'010000           .PCHAR    ;it.
17 00010'000420           JMP ER    :Error return.
18 00011'020423 LOVO:     LDA 0.OVO :Get OVLYO addr.
19 00012'126400           SUB 1.1   :Conditional load.
20 00013'006017           .SYSTM    :Load OVLYO
21 00014'020000           .OVLOD 0  :on channel 0.
22 00015'000413           JMP ER    ;Required.
23 00016'002420           JMP @.PRB :To PRNTB code in OVLYO.
24
25 00017'020416 LOV1:     LDA 0.    ;Get OVLY1 addr.
26 0020'126400            SUB 1.1   ;Conditional load.
27 00021'006017           .SYSTM    ;Load
28 00022'020000           .OVLOD 0  :OVLY1 on channel 0.
29 00023'000405           JMP ER    ;Required.
30 00024'002413           JMP @.PRC ;To PRNTC code in OVLY1.
31
32 00025'006017 RTURN:     .SYSTM
33 00026'004400            .RTN
34 00027'000401            JMP .+1   ;Reserved. never taken.
35
36 00030'006017 ER:        .SYSTM    ;Get CLI to
37 00031'006400            .ERTN     ;report problem.
38 00032'000401            JMP .     ;Impossible.
39 00033'000101 A:         "A
40 00034'077777 OVO:       OVLYO
41 00035'077777 OV1:       OVLY1
42 00036'077777 .PRE:      PRNTB
43 00037'077777 .PRC:      PRNTC
44 00040'000102"OFILE:     .+1*2
45 00041'051117            .TXT "ROOT.OL"
46       047524
47       027117
48       046000
49                          .END START
```

DG-25180

The RLDR command line for the program is:

RLDR ROOT [OVLY0, OVLY1] ROOT.LM/L ⟨CR⟩

This created a save file named ROOT.SV and an overlay file named ROOT.PL. It also created a disk file named ROOT.LM and sent the load map to this file. File ROOT.LM contains the following information after the load:

```
ROOT.SV LOADED BY RLDR REV XX.XX AT 09:05:43 07/25/XX
 ROOT
        000517
        000,000  OVLY0    000010
        000,001  OVLY1    000010
        001117
TMIN
NSAC3

  NMAX        001213
  ZMAX        000050
  CSZE        000000
   EST        000000
   SST        000000

  USTAD       000400
  START       000452
  LOV0        000463
  LOV1        000471
  RTURN       000477
  ER          000502
  PRNTB       000517
  PRNTC       000517
  TMIN        001120
  .SAC0       020016
  .SAC1       024016
  .SAC2       030016
  .SAC3       034016
  OVLY0       000,000
  OVLY1       000,001
```

Figure 11.9 File ROOT.LM after load DG-25183

Lines in the listing have the same meanings described under Load Map, earlier in the chapter. Below the program name ROOT is the starting address of the overlay node for OVLY0 and OVLY1. This node begins at 517_8 and ends at 1117_8 (Remember that RLDR rounds node size up to an even multiple of 400_8.)

The number 000,000 describes the node number (000 for the first node), and the overlay number (000 for the first overlay). We can access this overlay by using its name (OVLY0) instead of its number, since OVLY0 used .ENTO The node/overlay number (000,001) describes the same node (000) and the second overlay (001). The length of each overlay follows its name; in this case, each overlay is eight (000010_8) words long.

The next module, TMIN, is the single-task scheduler, which RLDR copied from the system library. If this were a multitask program (specified by a .COMM TASK statement or local /K switch), RLDR would have copied the multitask scheduler TCBMON instead.

Chapter 12
The RDOS Overlay Loader

Any existing overlay can be replaced by one or more different overlays. This replacement is a two-step process. First, the loader, named OVLDR.SV, creates the overlay replacement file; then the CLI command REPLACE uses the overlay replacement file to replace individual overlays in the existing overlay file. OVLDR can create a replacement file for up to 127 overlays. OVLDR handles standard RBs only, not extended RBs.

The format of the OVLDR command is:

OVLDR *filename overlay-descriptor*$_1$ /N *overlay-list* ↑ ⟨CR⟩
[*overlay-descriptor*$_2$ /N ↑ ⟨CR⟩
overlay-list ...] [*filename*/L] [*filename*/E] ⟨CR⟩

where:

filename is the name of the save file associated with the overlay file in which overlays are to be replaced. The replacement overlay file is named *filename*.OR.

overlay-descriptor is either a 1 to 6 digit octal number giving the node number/overlay number that identifies the overlay or is the symbolic name of the overlay. The overlay descriptor must be followed by the local /N switch. If you use a symbolic name, it must have been declared in a .ENTO name in the overlay file, and a .EXTN name in the save file.

overlay-list is a list of one or more relocatable binaries that are to replace the preceding overlay.

filenames followed by /E and /L are optional error and listing files.

The global switches are:

/A	Creates an alphabetical/numeric memory map of new symbols. You must also designate a listing file with local /L.
/E	Sends error messages to the console. (Used only when there is a listing file (/L) that suppresses console error output.
/H	Prints all numeric output in hexadecimal. By default, output is printed in octal.
/R	You must use this switch if global /R was specified to RLDR when the original overlay file was created. This switch instructs OVLDR to place any new labeled common in the root program as specified in the RLDR command. OVLDR cannot change the node size in the root program (.SV file); therefore, any new named common triggers an error message and aborts the OVLDR command.

If you omit global /R, OVLDR tries to place any new named common in the overlay node.

The local switches are:

name/E	Sends error message and other messages to file *name*. The file preceding the /E is designated to receive error and information messages.
name/L	Sends memory map to file name. This includes new symbols (those declared in replacement overlay nodes). The map is numeric unless you include the global /A switch.
old-overlay/N	Must follow an overlay-descriptor. Creates replacement for *old-overlay*. The *old-overlay* is an overlay-descriptor.

OVLDR works only when the following conditions exist:

1. An overlay file, *filename*.OL, is created by the RLDR command.
2. The save file, *filename*.SV, contains a symbol table (RLDR/D switch or .EXTN.SYM. in a module).

For example, assume that the following RLDR command was executed:

```
RLDR/D PROGB [OV0, OV1 OV2, OV3]↑
        FILEA [OV10, OV11] (CR)
```

The save and overlay files in Figure 12 1 are created.

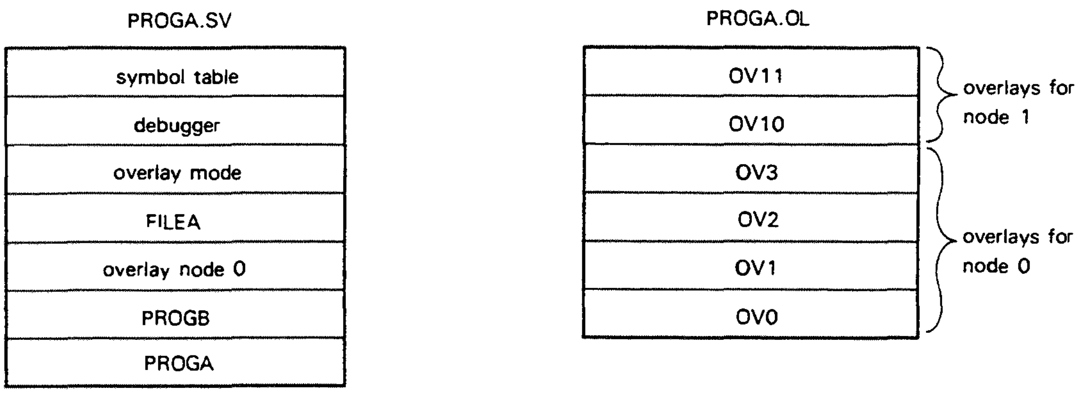


Figure 12.1 Save and Overlay files ID-00129

The /D global switch loaded the symbol table, but it also loaded the debugger. To conserve memory space, you can load only the symbol table by including it as an external normal in one of the source modules (for example, in PROGB or FILEA)·

```
.EXTN .SYSM.
```

Next, create an overlay replacement file via OVLDR. Suppose that you want to replace OV3 (node 0, number 2), and OV10 (node 1, number 0). Substitute the binaries NEW3 and NEW4 for OV3, and binary NEW10 for OV10. If we identified OV3 and OV10 with .ENTOL, and declared OV3 and OV10 in a .EXTN in the save file, you can simply type:

```
OVLDR/A/E PROGA OV3/N NEW3 NEW4 OV10/N
NEW10 PROGA.RM/L (CR)
```

If you omitted .ENTO, you must specify an octal number as an overlay descriptor. The number must be a 16-bit word, with the node number in the left byte and overlay number in the right byte. The command is:

```
OVLDR/A/E PROGA 000002/N NEW3 NEW4 00400/N
NEW10 PROGA.RM/L (CR)
```

(You can omit the leading zeroes in the overlay descriptor.)

Either command creates the overlay replacement file PROGA.OR, and the listing file PROGA.RM (the .RM stands for Replacement Map). The listing file contains a normal and alphabetical map of new symbols. Error messages go to both the console and the listing file. The replacement file PROGA.OR looks like this:

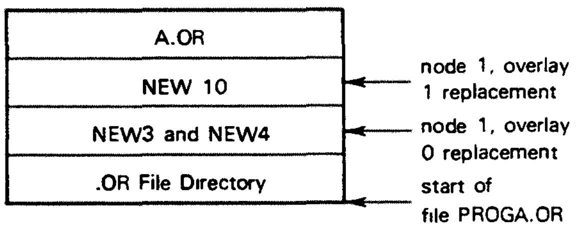


Figure 12.2 Replacement file PROGA.OR ID-00130

Now, assuming we received no errors, we execute the REPLACE step.

REPLACE PROGA ⟨CR⟩

After this command executes, PROGA.OL appears as follows:

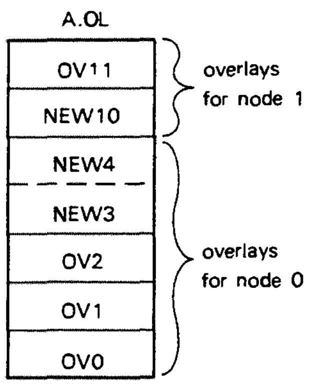


Figure 12.3 PROGA.OL ID-00131

Appendix A

Extended Assembler and Macroassemble Pseudo-op Summary

.ARGCT	MAC only	Number of most recent macro arguments
.BLK	ASM.MAC	Reserve a block of storage
COMM	ASM.MAC	Reserve a Named Common Area
.CSIZ	ASM.MAC	Specify an Unlabeled Common Area
.DALC	ASM,MAC	Define ALC Instruction
.DCMR	MAC only	Define Commercial Memory Reference Instruction
.DEMR	MAC only	Define Extended Memory Reference Instruction
.DERA	MAC only	Define Extended Memory Reference Instruction with Accumulator
.DEUR	MAC only	Define Extended User Instruction
.DFLM	MAC only	Define Floating Load or Store Instruction
.DFLS	MAC only	Define Floating Load or Store Instruction
.DIAC	ASM,MAC	Define an Instruction Requiring an Accumulator
.DICD	MAC only	Define an Instruction Requiring an Accumulator and Count
.DIMM	MAC only	Define an Instruction Requiring an Accumulator and an Immediate Word
.DIO	ASM.MAC	Define an I/O Instruction Without an Accumulator
.DIOA	ASM.MAC	Define an I/O Instruction With Two Required Fields
.DISD	MAC only	Define an Instruction With Source and Destination Accumulators
.DISS	MAC only	Define an Instruction with Source and Destination Accumulators Allowing Skip
.DMR	ASM.MAC	Define a Memory Reference Instruction with Displacement and Index
.DMRA	ASM,MAC	Define a Memory Reference Instruction with Two or Three Fields
.DO	MAC only	Assemble Source Lines Repetitively
.DUSR	ASM,MAC	Define User Symbol Without Formatting
.DXOP	MAC only	Define an Instruction with Source, Destination, and Operation Fields
.EJEC	MAC only	Begin a New Listing Page
.END	ASM,MAC	End-of-Program Indicator
.ENDC	ASM,MAC	Specify the End of Conditional Assembly
.ENT	ASM,MAC	Define a Program Entry
.ENTO	ASM,MAC	Define an Overlay Entry

.EOF.EOT	MAC only	Explicit End-of-File
.EXTD	ASM.MAC	Define an External Displacement Reference
.EXTN	ASM.MAC	Define an External Normal Reference
EXTU	ASM.MAC	Treat Undefined Symbols as External Displacements
.GADD	ASM.MAC	Add an Expression Value to an External Symbol
.GLOC	ASM,MAC	Reserve an Absolute Data Block
.GOTO	MAC only	Suppress Assembly Temporarily
.GREF	MAC only	Add an Expression Value to an External Symbol (OBO)
.IFE,.IFG. .IFL,.IFN	ASM.MAC	Perform Conditional Assembly
.LMIT	MAC only	Load Part of a Binary Module
.LOC	ASM.MAC	Set the Current Location Counter
.MACRO	MAC only	Name a Macro Definition
.MCALL	MAC only	Indicate Macro Usage
NOCON	MAC only	Inhibit or Re-enable Listing Condition Lines
.NOLOC	MAC only	Inhibit or Re-enable Listing Source Lines Without Location Fields

.NOMAC	MAC only	Inhibit or Re-enable Listing Macro Expansions
.NREL	ASM,MAC	Specify NREL Code Relocation
.PASS	MAC only	Number of Assembly Pass
.POP	MAC only	Pop the Value and Relocation of Last Item Pushed onto Stack
.PUSH	MAC only	Push a Value and its Relocation Property onto a Stack
.RB	MAC only	Name a Relocatable Binary File
.RDX	ASM,MAC	Radix for Numeric Input Conversion
.RDXO	MAC only	Radix for Numeric Output Conversion
.REV	MAC only	Seat the Revision Level
.TITL	ASM,MAC	Entitle an RB File
.TOP	MAC only	Value and Relocation of Last Stack Expression
.TXT, .TXTE, .TXTF, .TXTO	ASM,MAC	Specify a Test String
.TXTM	ASM,MAC	Change Text Byte Packing
.TXTN	ASM,MAC	Determine Text String Termination
.XPNG	ASM,MAC	Remove all Nonpermanent Macro and Symbol Definitions
.ZREL	ASM,MAC	Specify Page Zero Relocation

Appendix B

Error Messages

Extended Assembler and Macro Assembler

The assemblers output two types of error messages. The first type consists of one or more letter codes which are placed beside a line of source code on the listing. These letter flags indicate assembly errors that do not abort the assembler command. The second type of error occurs when the utility cannot continue with the command (possibly because it lacks an essential file like MACXR.SV); this is a fatal error. The alphabetical error codes are described first, followed by the fatal error messages.

Assembly Error Codes

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 as well as to the console. If the listing is suppressed, the error listing is still output to the console.

The list of possible assembler error codes is as follows:

A	Address error (ASM, MAC)
B	Bad character (ASM, MAC)
C	Macro error (MAC)
C	Colon error (ASM)
D	Radix error (ASM, MAC)
E	Equivalence error (ASM, MAC)
F	Format error (ASM, MAC)
G	Global symbol error (ASM, MAC)
I	Parity error on input (ASM, MAC)
K	Conditional or repetitive assembly error (ASM, MAC)
L	Location counter error (ASM, MAC)
M	Multiply-defined symbol error (ASM, MAC)
N	Number error (ASM, MAC)
O	Overflow error (ASM, MAC)
P	Phase error (ASM, MAC)
Q	Questionable line (ASM, MAC)
R	Relocation error (ASM, MAC)
S	Symbol table overflow error (ASM)
T	Symbol table pseudo-op error (ASM)
U	Undefined symbol error (ASM, MAC)
V	Variable label error (MAC)
X	Text error (ASM, MAC)
Z	Expression contains illegal operand (ASM)

Address Error (A)

Flags an error that appears in a memory reference instruction (MR), and indicates an illegal address.

Bad Character (B)

Indicates an illegal character in a 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.

Macro Error (C)

Occurs under the following circumstances:

1. You attempted to continue the definition of a macro that is not the last macro defined.

2. A macro has exhausted assembler working space. However, this should occur only if the macro definition causes endless recursion.

3. You specified more than 63_{10} arguments.

Radix Error (D)

Occurs on a .RDX or .RDXO psuedo-op when .RDX contains an expression that is not in the range of 2 through 20, when .RDXO contains an expression that is not in the range of 8 through 20, or when you use a digit that is not within the current input radix.

Equivalence Error (E)

Occurs when an equivalence line contains an undefined symbol on the right-hand side of the equal sign. This may occur on pass one before the symbol on the right-hand side has been defined, or on pass two if the symbol is not defined

Format Error (F)

Happens as a result of any attempt to use a format that is not legal for the type of line. 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 which have been assembled before the error is detected.

Global Symbol Error (G)

Occurs when there is an error in the declaration of an external or entry symbol.

Parity Error on Input (I)

Occurs when an input character does not have even parity. The assembler substitutes a backslash ( ) for any incorrect character and flags the line containing the error with an I.

Conditional or Repetitive Assembly Error (K)

Occurs when an .ENDC pseudo-op does not have a preceding .DO or .IF*n* pseudo-op.

Location Counter Error (L)

Occurs when an error is detected in a line that affects the location counter.

Multiply-Defined Symbol Error (M)

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 is flagged with the letter M at each appearance of the symbol.

Number Error (N)

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}.
2. A double-precision integer is greater than or equal to 2^{32}. The number is evaluated modulo 2^{32}.
3. A floating-point number is larger than $7.2*10^{75}$.

Overflow Error (O)

A field overflow error occurs when any of these conditions exist:

- Variable stack space is exceeded.
- A .TOP or .POP is given with no previous .PUSH.
- An instruction operand is not within the required limits, for example, 0 through 3 for an accumulator, or 0 through 7 for a skip field.

When overflow occurs in an instruction field, such as an accumulator field, the field will remain unchanged.

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 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.

Questionable Line (Q)

Occurs when you have used a # or @ atom improperly, a ZREL value where an absolute value is expected, or an instruction that may cause a skip immediately before a two-word instruction.

Relocation Error (R)

Indicates that an expression cannot be evaluated as a legal relocation type (absolute, relocatable, or byte relocatable) or that the expression mixes ZREL and NREL symbols.

Symbol Table Overflow Error (S)

Indicates that the symbol table capacity has been reached.

Symbol Table Pseudo-op Error (T)

Indicates that a symbol table pseudo-op has been specified incorrectly.

Undefined Symbol Error (U)

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.

Variable Label Error (V)

Occurs if anything other than a symbol follows the pseudo-op .GOTO.

Text Error (X)

Occurs when an error 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.)

Expression Contains Illegal Operand (Z)

Indicates that an expression contains an illegal operand such as an external symbol, instruction mnemonic, double-precision number, or floating-point number.

Fatal Errors

The following error messages abort the assembler commands and return the control to the CLI.

ATTEMPT TO POP LINKED ELEMENT WHEN NONE EXISTS

Internal assembler error. An internal stack containing linkage information has become too small. When it tries to pop a link from the stack, MAC does not find any link. This error produces a BREAK.SV file (FBREAK.SV if MAC is 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 the current directory or LINK to it from the current directory.

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

Library File Editor Error Messages

The Library File Editor (LFE) generates two types of fatal error messages, one indicating errors in the command string, and the other warning of errors occurring during the processing of files.

Command String Errors

The following messages indicate fatal errors in the LFE command string. The system will return to the CLI without processing any files

ILLEGAL HEADER IN INPUT LIBRARY

Header omitted or incorrect header block entered in the library file.

INVALID SWITCH FOR. string

You have submitted an invalid switch.

NOT A LFE COMMAND: key

Command key is unrecognized by the LFE: any letter key other than A, D, I, M, N, R, T, or X.

NOT ENOUGH ARGUMENTS

Insufficient number of arguments, such as unpaired arguments to the Replace (R) command.

TOO MANY ARGUMENTS

The argument string is too long for the allocated storage (currently, 500 characters).

UNEXPECTED ARGUMENT AT OR FOLLOWING. string

You have supplied an argument to a string that does not require an argument.

When there is no string following the colon in the error message, the error occurred at the end of the command line.

Processing Errors

The following messages indicate fatal errors while processing files. When these errors occur, the output files are terminated with a binary end block before the system returns to the CLI.

CHECKSUM ERROR IN LOGICAL RECORD: recordname

Bad record. If the checksum occurs within a title block itself, *recordname* is the name of the previous logical record. If no previous record exists, *recordname* is the name of the library itself.

CHECKSUM ERROR IN UPDATE FILE: filename

Bad record within filename.

ILLEGAL BLOCK IN LOGICAL RECORD: recordname

A logical record contains a bad block. If the expected title is missing, the record name will be the name of the previous logical record within the library.

ILLEGAL BLOCK UPDATE FILE: filename

For example, if a source file is specified as input instead of a binary file, illegal blocks will be encountered.

System Errors

The following message indicates a fatal error detected by the system rather than the LFE.

FILE DOES NOT EXIST, FILE. filename

Occurs when no input library file is found for the command.

Other fatal errors from the system refer to the LFE.SV file.

Caution Messages

The following messages result from nonfatal errors. Processing continues as indicated for each error.

DEFAULT OUTPUT IN FILE - filename

An output file specification is expected and is not found. *Filename* is used instead as the output file.

FILE DOES NOT EXIST, FILE: filename

An update file cannot be found. A search is made for filename and filename.RB. When they cannot be found, the file is omitted in processing.

FILE ALREADY EXISTS - filename

On an Extract (X) command there is already a file on the output device with the same name as the logical record to be extracted. The logical record is omitted in processing.

LOGICAL RECORD NOT FOUND - recordname

The input master does not contain *recordname*. The record (and any corresponding argument) are passed in processing.

UPDATE FILE MATCHES INPUT MASTER: filename

The result is nonfatal as long as there exists at least one valid update file argument. In this case, the matching update file is ignored.

Extended Relocatable Loader Error Messages

The Extended Relocatable Loader (RLDR) outputs error messages for both nonfatal and fatal errors. By default, messages go to the console; you can specify another file with local /L, or global and local /E.

Nonfatal Errors

Nonfatal errors do not abort the RLDR command, but they may produce an erroneous or useless save file. The nonfatal error messages are:

BINARY WITHOUT END BLOCK

The binary file has no end block. The file is loaded up to the point where the error is discovered.

DISPLACEMENT OVERFLOW nnnnnn [000000]

The displacement is too large when the loader attempts to resolve an external displacement. The following conditions cause the displacement to be too large:

1. The index = 00 and the unsigned displacement is $>$ 377.
2. The index $\neq$ 00 and the displacement is outside the range of $-200 <$ displacement $< +200$.
3. *nnnnnn* is the absolute address where overflow occurred. The displacement is left unresolved with a value of 000.
4. 000000 is the node number/overlay number if an overflow occurs in an overlay.

EXTERNAL UNDEFINED IN EXTERNAL EXPRESSION sssss

A .GADD block is encountered that references an undefined symbol, *sssss*. Zero is stored in the memory cell.

ILLEGAL BLOCK TYPE nnnnnn

The input is not a relocatable binary or library file. The file in error is not loaded. Octal number *nnnnnn* is the illegal block.

ILLEGAL NMAX VALUE nnnnnn

You have attempted to force the value of NMAX to a value lower than the current value of NMAX, for example, if the octal value following a /N local switch is lower than the current value of NMAX. *nnnnnn* is the illegal value. NMAX is unchanged.

LABELED COMMON IN NODE DEFINED OUTSIDE NODE

When an overlay declares labeled common, and you omit the global /R switch, RLDR gives this message if the common's address is outside the node. See global /G for message control

LABELED COMMON IS ROOT BOUND

RLDR inserts this message in the load map for each load which includes the global /R switch. The load map's starting node address and the starting address of the first overlay may be wrong by the amount of common allocated from that node's first overlay. See global /G for message control.

MULTIPLY DEFINED ENTRY sssss nnnnnn

You have entered an entry symbol or named common (.COMM) symbol, *sssss*, having the same name as one already defined. *nnnnnn* is the absolute address at which the symbol was originally defined. Of course, two or more named commons with the same name can occur, but an attempt to redefine an ENT as a COMM or a COMM as a .ENT results in an error.

NO SCHEDULER STARTING ADDRESS

Occurs in a standalone load (global /C) if the start block does not contain a starting address. The starting address can be patched into the starting address of the task scheduler (USTSA).

NO STARTING ADDRESS FOR LOAD MODULE

At assembly time, you have failed to terminate at least one of the programs to be loaded with a .END pseudo-op, followed by a starting address for the save file.

The starting address can be patched into word 0 of the task control block (TCB) pointed to by the current TCB pointer (USTCT). It must be stored as the starting address multiplied by 2.

SYSTEM LIBRARY NOT FOUND

The system library (SYS.LB) cannot be found on the current directory.

TASKS OR CHANNELS SPECIFIED = 0

There is a .COMM task block with the left or right byte of its equivalence word = 0, or when 0/K or 0/C appears in the COM.CM file.

*WARNING *** ZERO LENGTH OVERLAY*

You have attempted to load an overlay that contains nothing.

Fatal Errors

If an error is fatal, the error message and the location at which it was discovered are followed on the next line by the following message:

*** FATAL LOAD ERROR ***

For example:

LOAD OVERWRITE 001700
*** FATAL LOAD ERROR ***

The message is output to the error file, and control is returned to the CLI, which prints the message.

FATAL SYSTEM UTILITY ERROR

The fatal errors are:

ALIGNMENT ERROR IN NODE ASSIGNMENT OF LABELED COMMON

This error can occur when you omit global /R and overlays declare named common. When RLDR sees common declared within an overlay module, it checks whether this symbol has been previously defined as labeled common. If so, RLDR checks to see that the symbol was defined within this node. Then RLDR checks to make sure that it fits correctly. If the common has been defined within this node, and does not fit correctly, RLDR displays this error message.

CHECKSUM ERROR nnnnnn

A checksum that is computed on some block differs from zero. *nnnnnn* is the incorrect checksum.

EXTERNAL LOCATION UNDEFINED sssss

A .GLOC block is encountered with data to be loaded at the address of a symbol, *sssss*, that is still undefined.

ILLEGAL LOAD ADDRESS

You have attempted to load into locations 0 through 15.

LOAD OVERWRITE nnnnnn

The loader does not permit save or overlay file locations to be overwritten by subsequent data once they are loaded. This error occurs if you attempt to overwrite these file locations. The absolute address where the overwrite was attempted is given by *nnnnnn*.

NAMED COMMON ERROR sssss nnnnnn

Two programs have different sizes for a given area of labeled COMMON (defined by .COMM statements) and the second is larger. *sssss* gives the symbol name of the labeled COMMON and *nnnnnn* indicates the size of labeled COMMON requested by the present .COMM.

NEGATIVE ADDRESS nnnnnn

Bit 0 of an address word is set to 1. The assembler restricts addresses to the range: 0 ⟨ address ⟨ 2^{15}; however, the error can be caused by a reader error. *nnnnnn* represents the negative address.

NODE .COMM DEFINITION OUTSIDE FIRST MODULE

You have tried to declare new common with an incorrect overlay. If an overlay declares new common, and you use the global /R switch, only the first overlay within the square brackets can declare new common.

OVERLAY DIRECTORY OVERFLOW

The number of nodes exceeds 128, or the number of overlays at a given node exceeds 256.

PAGE ZERO OVERFLOW nnnnnn

Occurs in loading page zero relocatable data if the data overflows the page zero boundary (377_8). The absolute address of the first word of the data block that caused the overflow is given by *nnnnnn*.

RDOS ERROR

The loader issued a system call that could not be completed and resulted in an exceptional return. See the RDOS System Reference for system calls and possible error returns.

SYMBOL TABLE OVERFLOW

Occurs during loading when the size of the symbol table becomes so large that it would overwrite the loader in core.

SYMBOL TABLE TOO LARGE FOR CORE IMAGE

A global switch /S is given in the RLDR command line and the symbol table is going to overwrite loaded programs in the save file built by the loader.

TASK MONITOR ERROR (USTCH)

A .COMM block with symbol TASK has been encountered at a point when NMAX differs from the initial value of NMAX. This happens if .COMM TASK occurs in some module after the first module is loaded.

TCB OVERLAY TABLE OUTSIDE FIRST MAP PAGE BOUNDARY

The system tables (TCBs, overlay directory, etc.) extend above 2000_8, causing the program to run incorrectly in a mapped system.

VIRTUAL OVERLAYS AND GLOBAL /R DO NOT MIX

You cannot load virtual overlays and include the global /R switch. The remedy is to omit the global /R switch from the command line.

OVLDR Error Messages

These error messages may occur in executing an OVLDR command. They are all fatal errors.

COMMON SIZE ERROR

An overlay defines blank COMMON area to be larger than that in the save file.

EXTERNAL LOCATION UNDEFINED OR NOT WITHIN OVERLAY

Either a symbol is not defined within the save file or the symbol value is not legal for the overlay area.

ILLEGAL LOAD ADDRESS

You have attempted to load into an address outside the overlay node.

INSUFFICIENT MEMORY

OVLDR cannot execute in the available memory.

NEW .COMM CANNOT BE DECLARED IN NEW OVERLAY

You have used global /R, and a replacement overlay to declare new labeled common OVLDR cannot change the node size in the root program, and it cannot assign new labeled common outside the node.

NO OVERLAY DIRECTORY

The save file does not contain the overlay directory, OLDIR.

NO SYMBOL TABLE

The symbol table was not created when the save and overlay files were loaded.

.SV FILE SYMBOL IS EXTD/N FOR LABELED COMMON

A .SV file common symbol is .EXTN or .EXTD. RLDR normally resolves all references to undefined .EXTNs or .EXTDs to − 1. Therefore, no new overlay can define such undefined common symbols.

.SV FILE SYMBOL IS OTHER THAN COMM, EXTN, EXTD

When it analyzes labeled common, OVLDR displays this message if a .SV file symbol is not a COMM, .EXTN, or .EXTD. This means that the symbol has already been defined, and cannot be redefined.

Appendix C

ASCII Character Set

DECIMAL	OCTAL	HEX	KEY SYMBOL	MNEMONIC
0	**000**	00	↑@	NUL
1	**001**	01	↑A	SOH
2	**002**	02	↑B	STX
3	**003**	03	↑C	ETX
4	**004**	04	↑D	EOT
5	**005**	05	↑E	ENQ
6	**006**	06	↑F	ACK
7	**007**	07	↑G	BEL
8	**010**	08	↑H	BS (BACKSPACE)
9	**011**	09	↑I	TAB
10	**012**	0A	↑J	NEW LINE
11	**013**	0B	↑K	VT (VERT TAB)
12	**014**	0C	↑L	FORM FEED
13	**015**	0D	↑M	CARRIAGE RETURN
14	**016**	0E	↑N	SO
15	**017**	0F	↑O	SI
16	**020**	10	↑P	DLE
17	**021**	11	↑Q	DC1
18	**022**	12	↑R	DC2
19	**023**	13	↑S	DC3
20	**024**	14	↑T	DC4
21	**025**	15	↑U	NAK
22	**026**	16	↑V	SYN
23	**027**	17	↑W	ETB
24	**030**	18	↑X	CAN
25	**031**	19	↑Y	EM
26	**032**	1A	↑Z	SUB
27	**033**	1B	ESC	ESCAPE
28	**034**	1C	↑\	FS
29	**035**	1D	↑]	GS
30	**036**	1E	↑↑	RS
31	**037**	1F	↑—	US

DECIMAL	OCTAL	HEX	KEY SYMBOL
32	**040**	20	SPACE
33	**041**	21	!
34	**042**	22	" (QUOTE)
35	**043**	23	#
36	**044**	24	$
37	**045**	25	%
38	**046**	26	&
39	**047**	27	' (APOS)
40	**050**	28	(
41	**051**	29	)
42	**052**	2A	*
43	**053**	2B	+
44	**054**	2C	, (COMMA)
45	**055**	2D	-
46	**056**	2E	. (PERIOD)
47	**057**	2F	/
48	**060**	30	0
49	**061**	31	1
50	**062**	32	2
51	**063**	33	3
52	**064**	34	4
53	**065**	35	5
54	**066**	36	6
55	**067**	37	7
56	**070**	38	8
57	**071**	39	9
58	**072**	3A	:
59	**073**	3B	;
60	**074**	3C	<
61	**075**	3D	=
62	**076**	3E	>
63	**077**	3F	?
64	**100**	40	@

DECIMAL	OCTAL	HEX	KEY SYMBOL
65	**101**	41	A
66	**102**	42	B
67	**103**	43	C
68	**104**	44	D
69	**105**	45	E
70	**106**	46	F
71	**107**	47	G
72	**110**	48	H
73	**111**	49	I
74	**112**	4A	J
75	**113**	4B	K
76	**114**	4C	L
77	**115**	4D	M
78	**116**	4E	N
79	**117**	4F	O
80	**120**	50	P
81	**121**	51	Q
82	**122**	52	R
83	**123**	53	S
84	**124**	54	T
85	**125**	55	U
86	**126**	56	V
87	**127**	57	W
88	**130**	58	X
89	**131**	59	Y
90	**132**	5A	Z
91	**133**	5B	[
92	**134**	5C	\
93	**135**	5D	]
94	**136**	5E	↑ OR ^
95	**137**	5F	← OR _
96	**140**	60	` (GRAVE)

DECIMAL	OCTAL	HEX	KEY SYMBOL
97	**141**	61	a
98	**142**	62	b
99	**143**	63	c
100	**144**	64	d
101	**145**	65	e
102	**146**	66	f
103	**147**	67	g
104	**150**	68	h
105	**151**	69	i
106	**152**	6A	j
107	**153**	6B	k
108	**154**	6C	l
109	**155**	6D	m
110	**156**	6E	n
111	**157**	6F	o
112	**160**	70	p
113	**161**	71	q
114	**162**	72	r
115	**163**	73	s
116	**164**	74	t
117	**165**	75	u
11B	**166**	76	v
119	**167**	77	w
120	**170**	78	x
121	**171**	79	y
122	**172**	7A	z
123	**173**	7B	{
124	**174**	7C	|
125	**175**	7D	}
126	**176**	7E	~ (TILDE)
127	**177**	7F	DEL (RUBOUT)

Figure C.1 ASCII character set

DG-05495

Appendix D
Miscellaneous Relocatable Binary Information

This appendix begins by describing radix 50 representation, which is used by MAC, ASM, and compilers that use ASM, to condense five-character symbols. It then describes the block types that MAC and ASM create and use for conventional relocatable binaries. The next section illustrates block types in a real RB file; the final section briefly describes extended RBs.

Radix 50 Representation

MAC and ASM use radix 50 representation to condense symbols of five characters into two words of storage using only 27 bits. Each symbol consists of from 1 to 5 characters; a symbol having five characters may be represented as:

$a_4\ a_3\ a_2\ a_1\ a_0$

where each a may be one of the following characters:

A through Z (one of 26 characters)

0 through 9 (one of 10 characters)

. or ? (one of 2 characters)

All symbols are padded, if necessary, with nulls. Each character is translated into octal representation as follows:

Character a	**Translation b**
null	0
0 to 9	1 to 12_8
A to Z	13_8 to 44_8
.	45_8
?	46_8

If any a is translated to b, the bits required to represent the original a can be computed as follows:

$N_1 = (b_4 *50 + b_3) *50) + b_2$

$N_{1\ maximum} = (50_3 - 1 = 174777$, which can be represented in 16 bits (one word)

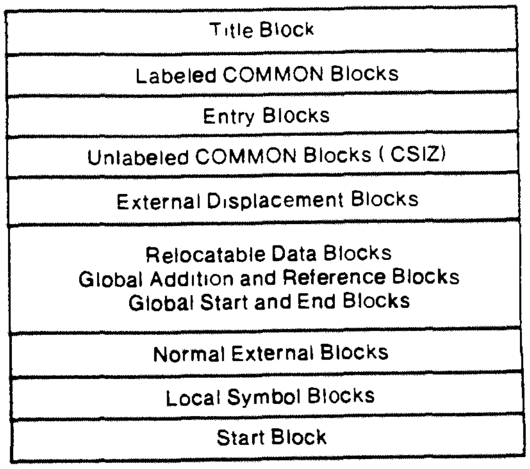
$N_2 = (b_1 *50) + b_0$

$N_{2\ maximum} = (50)^2 - 1 = 3077$, which can be represented in 11 bits.

Thus, any symbol a can be represented in 27 bits of storage. as shown below in the binary output block formats

Relocatable Binary Block Types

MAC, ASM, and any other language code generator divides binary output into a series of blocks The order in which blocks appear, if each type of block is present, is as follows:



Title Block
Labeled COMMON Blocks
Entry Blocks
Unlabeled COMMON Blocks (CSIZ)
External Displacement Blocks
Relocatable Data Blocks
Global Addition and Reference Blocks
Global Start and End Blocks
Normal External Blocks
Local Symbol Blocks
Start Block

Figure D.1 Binary block order SD-00646

The relocatable binary output must contain at least a title block and a start block. The presence of one or more of the other types of blocks will depend upon source input The pages following describe each block, in the order shown in Figure D.1.

Bytes are always swapped in the word; thus, 003400 means 00 000 111/00 000 000; and, after swapping, 7 The first word of each block contains a number indicating the type of block. The number is in the range of 2 through 20_8

The second word of each block is the word count. It is always in two's complement format. Where the word is a constant for every block of the particular type, the word count constant is shown in parentheses in the format

Words 3 through 5 are reserved for relocation flags. Some block types contain these flags, others do not. The relocation property of each address, datum, or symbol is defined in three bits. For example, for a relocatable data block, bits 0 through 2 of word 3 apply to the address, bits 3 through 5 apply to the first data word, bits 6 through 8 apply to the second data word, and so on. The meaning of the bit settings is given in Table D 1.

Bits	Meaning
000	Illegal
001	Absolute
010	Normal Relocatable
011	Normal Byte Relocatable
100	Page Zero Relocatable
101	Page Zero Byte Relocatable
111	Illegal

Table D.1 Bit settings

All other blocks use bits of word 3 only and set words 4 and 5 to zero

Word 6 contains a checksum, so the sum of all words in the block is 0.

For those blocks containing user symbols, each symbol entry is three words long.

The first 27 bits of the three-word entry contain the user symbol name in radix 50 form. The last five bits of the second word are used as a symbol type flag, where the currently defined types are:

Bit	Meaning
00000	Entry Symbol
00001	Normal External Symbol
00010	Labeled Common
00011	External Displacement Symbol
10100	Title Symbol
00100	Overlay Symbol
01000	Local Symbol

Table D.2 Symbol types

Block Formats

The setting of the third word allocated for each user symbol entry varies with the type of block and is described in the format writeups of each block.

Title Block (.TITL)

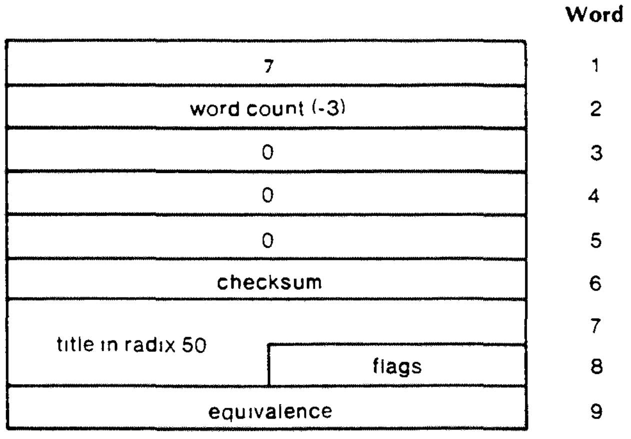


Figure D.2 .TITL SD-00647

The third word of the user symbol entry for a title is set to 0.

Labeled Common Block (.COMM)

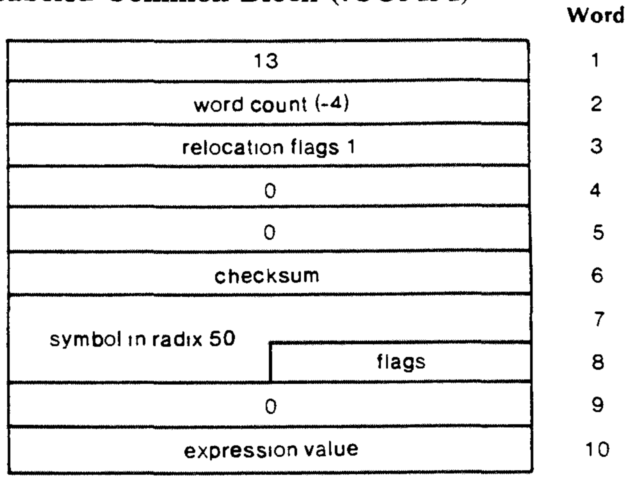


Figure D.3 .COMM SD-00648

Bits 0 through 2 of the relocation flags (word 3) apply to the expression (expr following .COMM). All other bits of the word are set to zero.

Entry Block (.ENT)

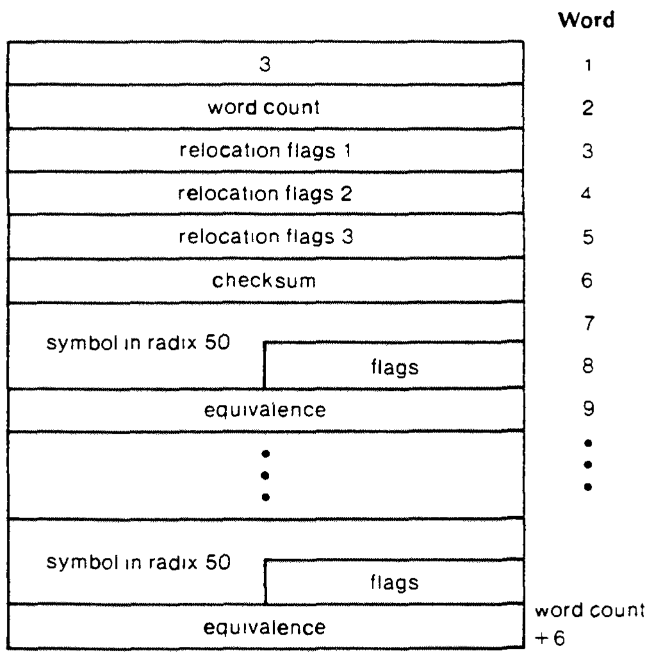


Figure D.4 .ENT SD-00649

Note that the relocation flags for the entry block are as previously described. except that they apply to the third word of every user symbol entry. (For entry block user symbols, the third word of the user symbol is used to equivalence entry symbols.) Because each equivalence requires relocation flags, and there are only three words for flags, there is a limit of 15_{10} symbols for each block.

Overlay entry .ENTO is the same as .ENT, except for the flags in the last five bits of the flag word(s)

Unlabeled Common Size Block (.CSIZ)

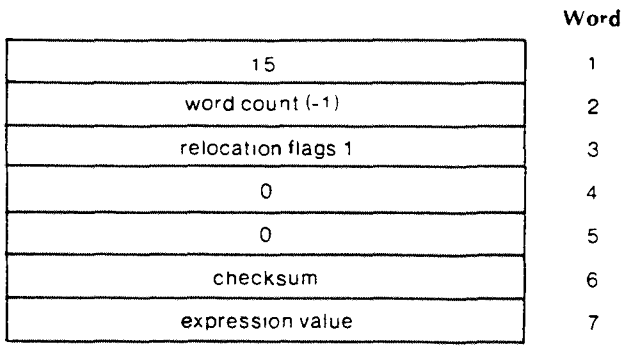


Figure D.5 Unlabeled commmon size block (.CSIZ) SD-00650

Bits 0 through 2 of the relocation flags (word 3) apply to expression (*expr* following .CSIZ). All other bits of word 3 are zeroed.

External Displacement Block (.EXTD)

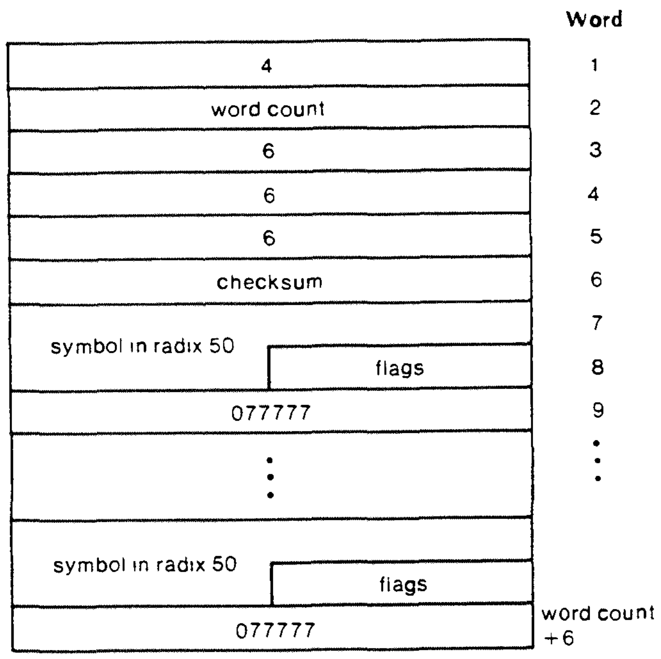


Figure D.6 External displacement block (.EXTD) SD-00651

The third word of each user symbol entry in the external displacement block is set to 077777.

Relocatable Data Block

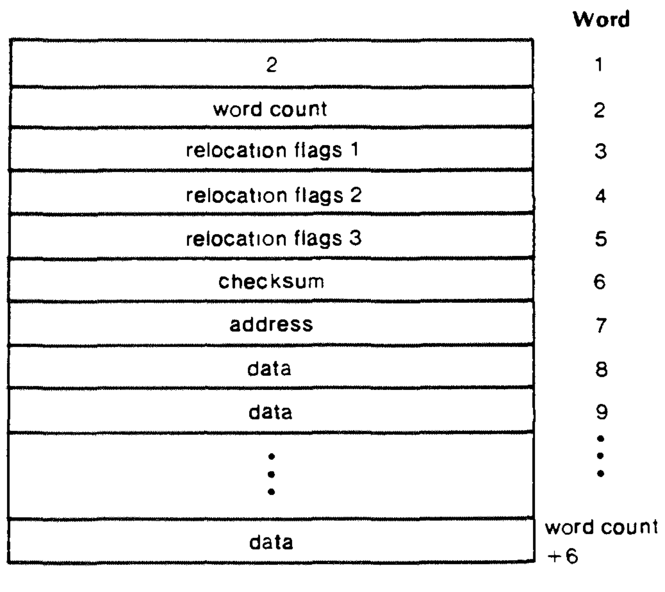


Figure D.7 Relocatable data block SD-00652

Contents of the relocation flag words (words 3 through 5) are as described previously. Because of relocation flag requirements, there is a limit of 14_{10} data words per block.

Global Addition Block (.GADD) And Global Reference Block (.GREF)

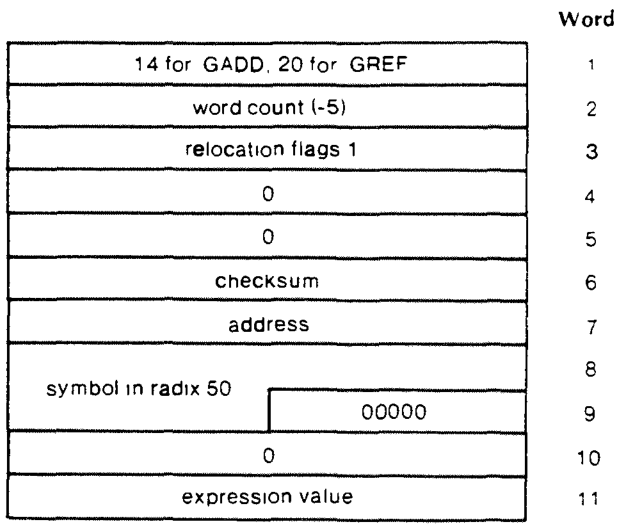


Figure D.8 Global addition and global reference blocks (.GADD and .GREF) SO-00653

Bits 0 through 2 of the relocation flags (word 3) apply to the address, and bits 3 through 5 apply to the expression All other bits of word 3 are zeroed.

Global Location Start And End Blocks (.GLOC)

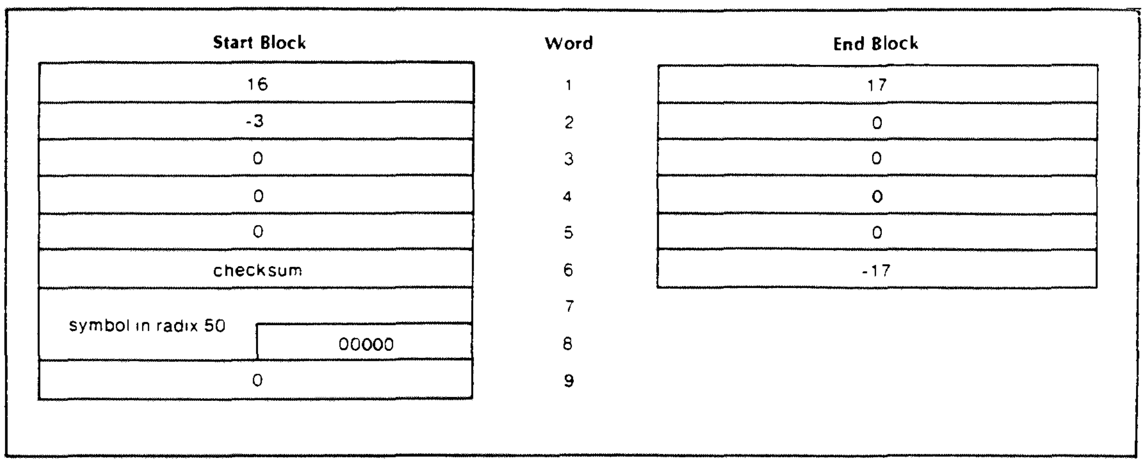


Figure D.9 Global location start and end blocks (.GLOC) SD-00654

Normal External Block (.EXTN)

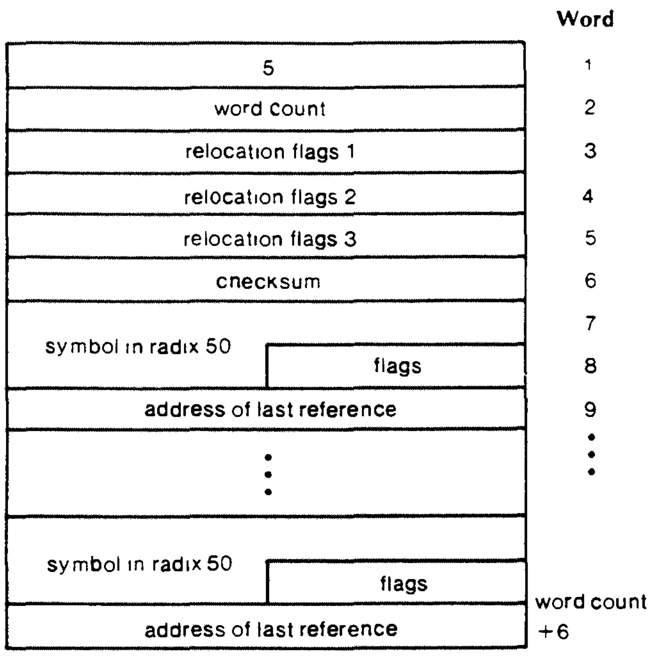


Figure D.10 Normal external block (.EXTN) SD-00655

The third word of each user symbol entry in the normal external block contains the address of the last reference. Relocation flags are used as in .ENT blocks. There is a limit of 15_{10} symbols per block because of space required for relocation flags.

Local Symbol Block

	Word
10	1
word count	2
relocation flags 1	3
relocation flags 2	4
relocation flags 3	5
checksum	6
symbol in radix 50	7
flags	8
equivalence	9
• • •	• • •
symbol in radix 50	
flags	word count
equivalence	+6

Figure D.11 Local symbol block SD-00656

The third word of every symbol entry is used for the equivalence of local symbols. Relocation flags are used as in .ENT blocks. There can be only 15_{10} local symbols per block because of relocation flag space requirements.

Library Start And End Blocks

Library Start Block	Word	Library End Block
11	1	12
0	2	0
0	3	0
0	4	0
0	5	0
-11	6	-12

Figure D.13 Library start and end blocks SD-00658

These blocks mark the beginning and end of a series of binary blocks which make up a library file. They are created by the Library File Editor utility, not MAC or ASM; hence, their format differs from the format of other relocatable binary blocks.

Start Block

	Word
6	1
word count (-1)	2
relocation flags 1	3
0	4
0	5
checksum	6
address	7

Figure D.12 Start block SD-00657

Bits 0 through 2 of the first relocation flag word are used for address relocatability; other bits of word 3 are 0.

RB File Illustration

Having described each RB block structure, we can now examine a sample RB file, and see the structures of some actual blocks. We will use the source program ROOT, shown in Chapter 3. For clarity, we have repeated the listing, without comments, below.

```
                          .TITL ROOT
                          .ENT START. LOVO. LOV1. RTURN  ER
                          .EXTN OVLYO. PRNTB. OVLY1. PRNTC
          000001          .TXTM 1
                           NREL


00000'020440 START        LDA 0. OFILE
00001'126400              SUB 1.1
00002'006017              .SYSTM
00003'012000              .OVOPN 0
00004'000424              JMP ER
00005'020426              LDA 0.A

00006'006017              .SYSTM
00007'010000              .PCHAR
00010'000420              JMP ER
00011'020423 LOVO:        LDA 0.OVO
00012'126400              SUB 1.1
00013 006017               SYSTM
00014'020000              .OVLOD 0
00015'000413              JMP ER
00016'002420              JMP @.PRB

00017'020416 LOV1         LDA 0.
00020'126400              SUB 1.1
00021'006017              .SYSTM
00022'020000              .OVLOD 0
00023'000405              JMP ER
00024'002413              JMP @.PRC

00025'006017 RTURN.        .SYSTM
00026'004400              .RTN
00027'000401              JMP .+1

00030'006017 ER:           .SYSTM
00031'006400              .ERTN
00032'000401              JMP.
00033'000101 A:           "A
00034'077777 OVO·         OVLYO
00035'077777 OV1:         OVLY1
00036'077777 .PRE:        PRNTB
00037'077777 .PRC:        PRNTC
00040'000102"OFILE:       .+1*2
00041'051117              .TXT "ROOT.OL"
          047524
          027117
          046000
                          .END START
```

Figure D.14 Sample RB File DG-25181

The RB file built by MAC (not ASM) from source file ROOT appears below. We used the command FPRINT/Z /L to get it; the addresses of the words are shown in the left column, and the ASCII values in each 20-word series (if any) are shown in the right column (For the rest of this appendix, all numbers will be octal, unless we specify otherwise.) ROOT.RB contains a title block, an entry block, a data block, an external normal (.EXTN) block, and a start block. We have drawn square brackets to delimit each block in the RB.

```
  0 [003400  176777  000000 000000 000000 163666  000663 012226
                                                  ...... ... 6.3..
 10  000000][001400  170777 022111 000000 000000  164635 020142
                                                  ......$I.... .
 20  000000  014000  147663 000217 012400 104215  000012 007400
                                                  ....03...... ..
 30  104215  000005  004400 175671 140217 000000][001000 170777
                                                  .  ....9@.... .
 40  111104  111044  111044 131111 000000 020041  000255 007414
                                                  .D.$.$2I .'.- .
 50  000024  012001  013041 007414 000020 010001  011441 000255
                                                  .....'..  . .'.-
 60  007414  000040  005401 001000 170777 111104  111044 111044
                                                   .... .   .D.$.$
 70  125474  007000  010005 007041 000255 007414  000040 002401
                                                  + <...  ' -.....
100  005405  007414  000011 000401 997414 000015  000401 040400
                                                   ........ .. .A.
110  001000  173377  111104 111144 000000 114072  016000 177577
                                                  .... D.......
120  177577  177577  177577 041000 047522 052117  047456 000114]
                                                  ......B.ORTOO..L
130 [002400  172377  020111 000000 000000 050630  174246 120627
                                                  ....I....G..&'.
140  017400  053241  040657 016400 174246 100627  017000 053241
                                                  ..V'A/...&... V'
150  020657  016000][003000 177777 000100 000000  000000 175677
                                                  !/.. ....@.....?
160  000000] ----    ----   ----   ----   ----    ----   ----
                                                  ..... .... .....
```

Figure D.15 RB file built by MAC DG-25182

Words 0 through 10 comprise the *title* block (words 1 through 8 in .TITL figure). Word 0 is 7 when we swap bytes, and this corresponds to block type 7, as described under title block. The second word, 176777, is a −3 after we swap, as it should be for the second word in the title block. The next three words are 0, as they should be. Word 5, 163666, is the checksum. The title, ROOT, is placed in radix 50 in word 6 and in bits 0 through 11 of word 7. The last 5 bits in word 7 evaluate to 00100, meaning *title symbol.*

Word 11 starts the *entry* block; swapped, 001400 is 3, which starts an entry block. Word 12 specifies the word count in two's complement; words 13, 14 and 15 contain relocation flags, and word 16 is the checksum. Then five three-word groups follow, one for each .ENT symbol in ROOT. The entry block ends at word 35.

A *data* block begins at word 36, with 001000 (or 2 after swapping); number 2 identifies the block. The next word, at 37, is the word count in two's complement; words 40, 41, and 42 are relocation flags; word 43 (131111) is the checksum; and word 44 (000000) is the address of the first datum. The three data blocks extend from 45 (020041) to 127 (000114).

Word 130 starts an *external normal* (.EXTN) block. It contains 5 (swapped) as a numeric identifier, is followed by the word count in two's complement, then by three words containing relocation flags. Following the flags, in word 135, is the checksum; then four three-word descriptors, one descriptor for each symbol declared in .EXTN, in ROOT.

Finally, words 152 through 160 comprise the *start* block. The block type, 6 (swapped) is in the first word, −1 is in the second word, relocation flags are in word 154, 0 is in the next two words, the checksum is in 157, and the start address (000000) is in word 160.

Extended RBs

Certain Data General compilers produce Extended RBs, which allow for the recognition of eight-character symbols. The Macroassembler, MAC, also produces an Extended RB if you include the global /T switch in the MAC command.

In terms of program development, there are no functional differences between Extended and conventional RBs: but there are differences in block words, as follows:

1. Within extended RBs, all bytes are in their proper order (not swapped as in standard RBs), except for the first word of the TITLE block. The TITLE block's first word is 003600_8 (as opposed to 003400_8 for a standard block), and the extra bit tells RLDR to expect extended format blocks. Library START/END blocks are in standard RB format.

2. Each three-word symbol in standard RB format (radix 50 words and equivalence word) has the following format in an RB block:

Word 0	Right byte contains the number of characters in the symbol name. Left byte contains the symbol type. This is specified only in a .ENT or .ENTO block, where RLDR uses it to distinguish between the two. For all other block types, RLDR needs only the first (header) word in the block to derive the block type.
Word 1 to *n*	Contains the symbol in ASCII representation, two characters per word, with any odd character zeroed. The number of characters cannot exceed 32_{10}.
Word $n+1$	Contains the symbol equivalence, as follows:

TITLE and GLOC Start	= 0.
EXTD	= 077777_8.
EXTN	= address of last reference.
COMM	= common size.
GADD and GRED	= expression.
ENT and ENTO	= equivalence.

The relocation flags have the same meaning as in standard RBs, and each block has the same size restrictions (15_{10} for ENT, 14_{10} for a data block, etc.).

Index

Data General

TP________

TIPS ORDER FORM
Technical Information & Publications Service

BILL TO	SHIP TO. (if different)
COMPANY NAME________	COMPANY NAME________
ADDRESS ________	ADDRESS ________
CITY________	CITY________
STATE________ ZIP ________	STATE________ ZIP ________
ATTN. ________	ATTN. ________

QTY	MODEL #	DESCRIPTION	UNIT PRICE	LINE DISC	TOTAL PRICE

(Additional items can be included on second order form)

[Minimum order is $50.00]	TOTAL	
Tax Exempt #________ or Sales Tax (if applicable)	Sales Tax	
	Shipping	
	TOTAL	

METHOD OF PAYMENT

☐ Check or money order enclosed
For orders less than $100.00

☐ Charge my ☐ Visa ☐ MasterCard
Acc't No.________ Expiration Date________

☐ Purchase Order Number:________

SHIP VIA

☐ DGC will select best way (U.P.S or Postal)

☐ Other:
☐ U.P.S. Blue Label
☐ Air Freight
☐ Other ________

NOTE: ORDERS LESS THAN $100, INCLUDE $5.00 FOR SHIPPING AND HANDLING.

Person to contact about this order ________ Phone ________ Extension ________

Mail Orders to:

Data General Corporation
Attn: Educational Services/TIPS F019
4400 Computer Drive
Westboro, MA 01580
Tel. (617) 366-8911 ext 4032

____
Buyer's Authorized Signature Date
(agrees to terms & conditions on reverse side)

____
Title

____
DGC Sales Representative (If Known) Badge #

DISCOUNTS APPLY TO MAIL ORDERS ONLY

012-1780

educational services

CUT ALONG DOTTED LINE

DATA GENERAL CORPORATION
TECHNICAL INFORMATION AND PUBLICATIONS SERVICE
TERMS AND CONDITIONS

Data General Corporation ("DGC") provides its Technical Information and Publications Service (TIPS) solely in accordance with the following terms and conditions and more specifically to the Customer signing the Educational Services TIPS Order Form shown on the reverse hereof which is accepted by DGC.

1. **PRICES**
   Prices for DGC publications will be as stated in the Educational Services Literature Catalog in effect at the time DGC accepts Buyer's order or as specified on an authorized DGC quotation in force at the time of receipt by DGC of the Order Form shown on the reverse hereof. Prices are exclusive of all excise, sales, use or similar taxes and, therefore are subject to an increase equal in amount to any tax DGC may be required to collect or pay on the sale, license or delivery of the materials provided hereunder.

2. **PAYMENT**
   Terms are net cash on or prior to delivery except where satisfactory open account credit is established, in which case terms are net thirty (30) days from date of invoice

3. **SHIPMENT**
   Shipment will be made F.O.B Point of Origin. DGC normally ships either by UPS or U.S. Mail or other appropriate method depending upon weight, unless Customer designates a specific method and/or carrier on the Order Form. In any case, DGC assumes no liability with regard to loss, damage or delay during shipment

4. **TERM**
   Upon execution by Buyer and acceptance by DGC, this agreement shall continue to remain in effect until terminated by either party upon thirty (30) days prior written notice. It is the intent of the parties to leave this Agreement in effect so that all subsequent orders for DGC publications will be governed by the terms and conditions of this Agreement.

5. **CUSTOMER CERTIFICATION**
   Customer hereby certifies that it is the owner or lessee of the DGC equipment and/or licensee/sub-licensee of the software which is the subject matter of the publication(s) ordered hereunder.

6. **DATA AND PROPRIETARY RIGHTS**
   Portions of the publications and materials supplied under this Agreement are proprietary and will be so marked. Customer shall abide by such markings. DGC retains for itself exclusively all proprietary rights (including manufacturing rights) in and to all designs, engineering details and other data pertaining to the products described in such publication. Licensed software materials are provided pursuant to the terms and conditions of the Program License Agreement (PLA) between the Customer and DGC and such PLA is made a part of and incorporated into this Agreement by reference. A copyright notice on any data by itself does not constitute or evidence a publication or public disclosure

7. **DISCLAIMER OF WARRANTY**
   DGC MAKES NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY AND FITNESS FOR PARTICULAR PURPOSE ON ANY OF THE PUBLICATIONS SUPPLIED HEREUNDER.

8. **LIMITATIONS OF LIABILITY**
   IN NO EVENT SHALL DGC BE LIABLE FOR (I) ANY COSTS, DAMAGES OR EXPENSES ARISING OUT OF OR IN CONNECTION WITH ANY CLAIM BY ANY PERSON THAT USE OF THE PUBLICATION OF INFORMATION CONTAINED THEREIN INFRINGES ANY COPYRIGHT OR TRADE SECRET RIGHT OR (II) ANY INCIDENTIAL, SPECIAL, DIRECT OR CONSEQUENTIAL DAMAGES WHATSOEVER, INCLUDING BUT NOT LIMITED TO LOSS OF DATA, PROGRAMS OR LOST PROFITS.

9. **GENERAL**
   A valid contract binding upon DGC will come into being only at the time of DGC's acceptance of the referenced Educational Services Order Form. Such contract is governed by the laws of the Commonwealth of Massachusetts. Such contract is not assignable. These terms and conditions constitute the entire agreement between the parties with respect to the subject matter hereof and supersedes all prior oral or written communications, agreements and understandings. These terms and conditions shall prevail notwithstanding any different, conflicting or additional terms and conditions which may appear on any order submitted by Customer.

DISCOUNT SCHEDULES

DISCOUNTS APPLY TO MAIL ORDERS ONLY.

LINE ITEM DISCOUNT

5-14 manuals of the same part number - 20%
15 or more manuals of the same part number - 30%

DISCOUNTS APPLY TO PRICES SHOWN IN THE CURRENT TIPS CATALOG ONLY.

Data General

TIPS ORDERING PROCEDURE:

Technical literature may be ordered through the Customer Education Service's Technical Information and Publications Service (TIPS).

1. Turn to the TIPS Order Form.
2. Fill in the requested information. If you need more space to list the items you are ordering, use an additional form. Transfer the subtotal from any additional sheet to the space marked "subtotal" on the form.
3. Do not forget to include your MAIL ORDER ONLY discount. (See discount schedules on the back of the TIPS Order Form.)
4. Total your order. (MINIMUM ORDER/CHARGE after discounts of $50.00.)

   If your order totals less than 100.00, enclose a certified check or money order for the total (include sales tax, or your tax exempt number, if applicable) plus $5.00 for shipping and handling.
5. Please indicate on the Order Form if you have any special shipping requirements. Unless specified, orders are normally shipped U.P.S.
6. Read carefully the terms and conditions of the TIPS program on the reverse side of the Order Form.
7. Sign on the line provided on the form and enclose with payment. Mail to:

   TIPS
   Educational Services - M.S. F019
   Data General Corporation
   4400 Computer Drive
   Westboro, MA 01580
8. We'll take care of the rest!

educational services

Data General Corporation, Westboro, MA 01580

069-400019-01