Extended Relocatable Loaders

User's Manual

093-000080-05

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

Ordering No. 093-000080
©Data General Corporation, 1972, 1973, 1974, 1975, 1978
All Rights Reserved
Printed in the United States of America
Revision 05, March 1978
Licensed Material - Property of Data General Corporation

NOTICE

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

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

Extended Relocatable Loaders
User's Manual
093-000080

Revision History:

093-000080	
	Original Release - December 1972
	First Revision - April 1973
	Second Revision - November 1973
	Third Revision - February 1975
(RDOS 05)	Fourth Revision - September 1975
(RDOS 06/DOS 01)	Fifth Revision - March 1978
086-000026	
	Original Release - April 1975

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

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

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

PREFACE

The Extended Relocatable Loader (called RLDR) can process and link any number of files and libraries to produce an executable save file. RLDR can process binary files produced by the Data General assemblers MAC and ASM, and by compilers which use ASM. It can also process libraries produced by the Library File Editor utility (LFE).

Data General produces three versions of the Extended Relocatable Loader, and this manual describes them all. We have organized the manual as follows:

- Chapter 1 describes the Stand-alone Loader, which runs without an operating system.
- Chapter 2 explains the Stand-alone Operating System Loader, which runs under the SOS system.
- Chapter 3 describes the RDOS/DOS loader, which runs from disk.
- Chapter 4 explains the RDOS Overlay Loader, which can replace overlays in an existing overlay file.
- Appendix A details the types of binary blocks the Loader can process. Read this only if you plan to write your own language processor.
- Appendix B explains named common in overlays. Read this if you plan to have overlays declare new named common.
- Appendix C details the . LMIT feature, which instructs the Loader to partially load a binary.
- Appendix D explains the format of symbols in the Loader's symbol tables. Read this if you are interested in the RDOS/DOS Loader's internal structure.

(PREFACE continued)

If you have an RDOS system, read Chapters 3 and 4, and, optionally, the appendixes.

If you have a DOS system, read Chapter 3, and, optionally,the appendixes.

If yours is a SOS system, read Chapter 2, and, optionally, Appendix A.

To run the Loader without an operating system, read Chapter 1, and, optionally, Appendix A.

LOADER OVERVIEW

When an assembler or compiler processes a source file, it translates each symbol into a machine instruction or numeric value, and gives each instruction or value a "relative" location. The first instruction receives the relative location zero, the second receives the location of one, and so on. The Relocatable Loader assigns each relative location an "absolute" location for execution in memory.

The Loader maintains the value of the first location available for loading, based on the programs previously loaded. As each assembled program is loaded, the relocatable loader updates the value of the next location available for additional loading. In this way, any number of separately assembled modules can be loaded together without any conflict in absolute storage assignment.

Library files are simply collections of relocatable binary programs, one or more of which will be loaded to resolve external references appearing in previously loaded programs.

When the relocatable loader is invoked, relocatable binary programs, absolute binary programs, and libraries will be loaded in the order in which they are given to the loader. In the case of the stand-alone relocatable loader, this will be the order in which they appear on the input device. In the case of other versions of the loader, the order of loading is the order in which the names of the programs appear in an RLDR command line. Each library is searched once each time its name appears in the command line.

END OF PREFACE

TABLE OF CONTENTS

CHAPTER 1 - STAND-ALONE RELOCATABLE LOADER

CHAPTER 2 - SOS MAGNETIC TAPE/CASSETTE RELOCATABLE LOADER

CHAPTER 3 - THE RDOS/DOS RELOCATABLE LOADER (RLDR)

CHAPTER 1

STAND-ALONE RELOCATABLE LOADER

OPERATION

The relocatable loader tape is in binary format, and is loaded using the binary loader (091-000004). Once loaded, the relocatable loader will self-start and type:

SAFE=

on the Teletype. 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 5 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 will then prompt a user response by typing:

*

on the teletypewriter.

Relocatable loader action is initiated by single digit responses to the loader prompt, *. The possible loader mode responses are tabulated on the following page and are described throughout this manual. Each time the loader has completed its response to a user request, it will type

*

OPERATION (Continued)

and await a new request. If an illegal response is input, the loader prints:

?

and awaits a legal response. User-loader interaction is terminated 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, the user may issue the digit 7 in response to the * query.

TABLE OF RESPONSES TO LOADER PROMPT *

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.

RESTART PROCEDURES

To restart the loading process, the user may press RESET, enter 000377 in the data switches, 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 divided into a series of blocks and

INPUT TO THE STAND-ALONE LOADER (Continued)

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.

Title block
COMMON block(s)
Entry block(s)
.CSIZ block(s)
Displacement External block(s)
Relocatable data block(s) Global addition block(s) Global start and end block(s)
Normal external block(s)
Local symbol block(s)
Start block

To load a single relocatable binary tape, the user responds to the "*" 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 "*" 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 origined data is loaded at the locations specified to the assembler. For relocatable origined data, the loader is initialized to assume two relocation variables called NMAX and ZMAX.

RELOCATION VARIABLES ZMAX AND NMAX (Continued)

ZMAX has an initial value of octal 50, where 50 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 octal 440, 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 ZMAX and NMAX values, plus current values of the start and end of the symbol table (SST and EST) and CSZE (unlabeled COMMON size) by responding to the "*" prompt by

```
5
```

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

```
NMAX nnnnnn
XMAX 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

THE SYMBOL TABLE (LOADER MAP) (Continued)

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 "*" prompt).

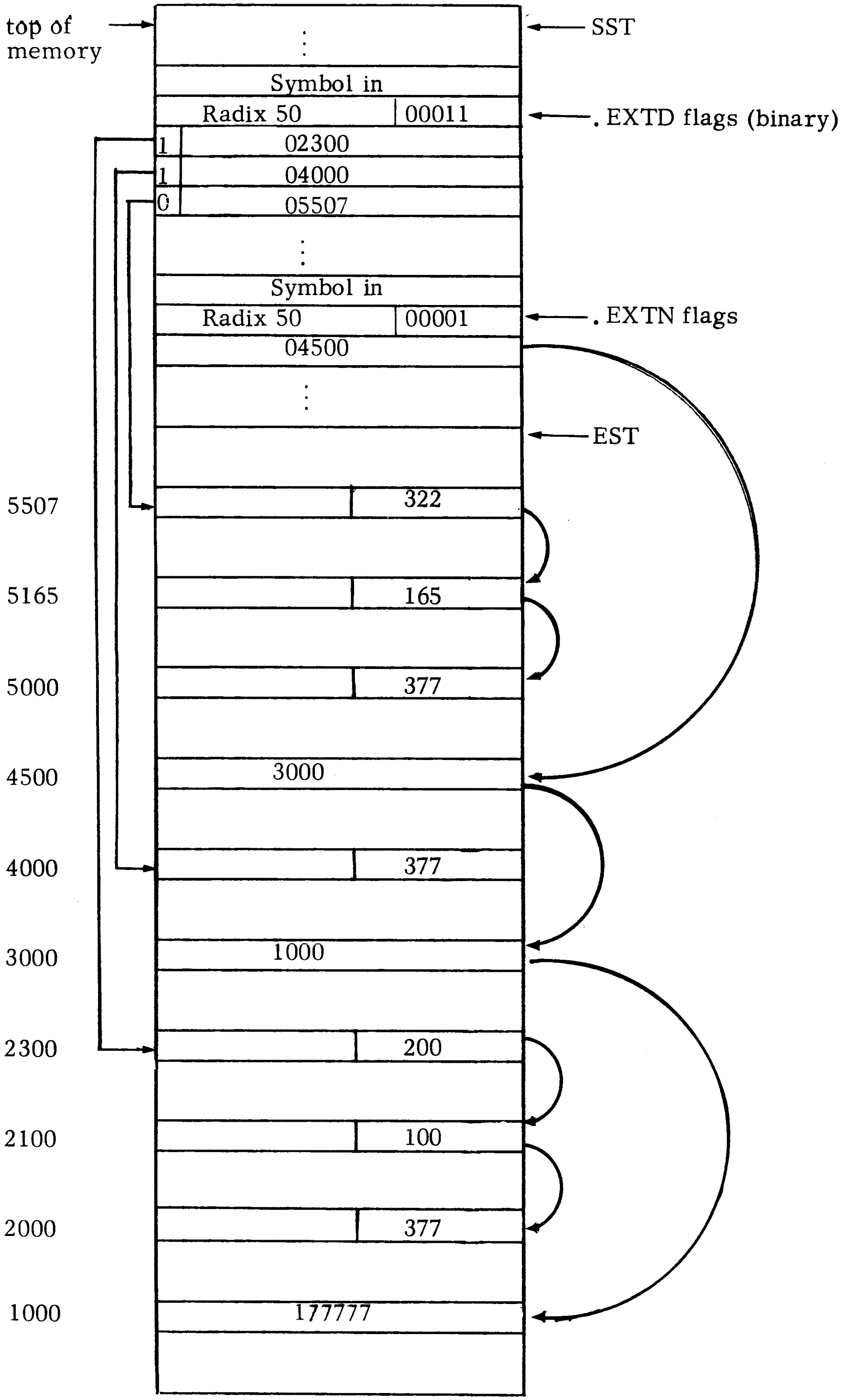
A defined symbol is represented by three words. The first two words contain the symbol name (in radix 50, using 27 bits). There is a six-digit octal value in the third word.

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 via 8-bit relative displacements, contained in the rightmost byte (right half) of each storage word. Each chain is terminated by a word having 377_8 in its right-most byte. Thus, if two consecutive references are farther than 367_8 words apart, a new chain must be started as shown on the following page.

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

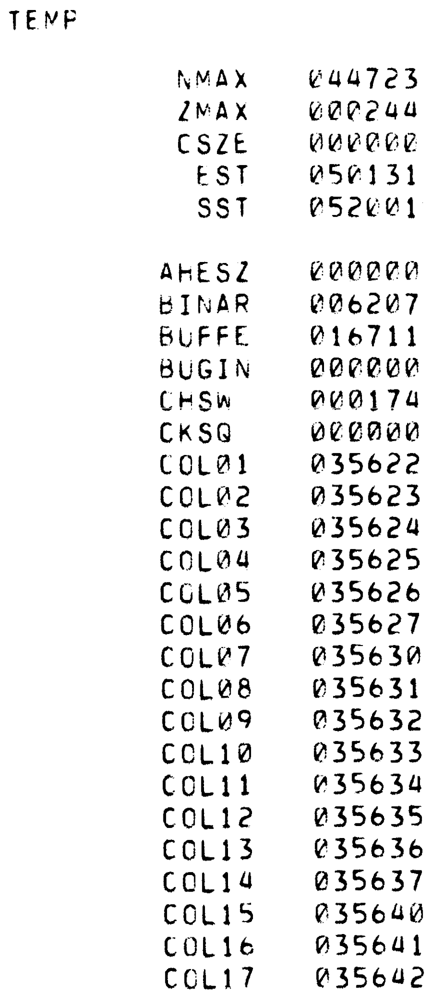

top of memory
SST
Symbol in
Radix 50
00011
. EXTD flags (binary)
1
02300
1
04000
0
05507
Symbol in
Radix 50
00001
. EXTN flags
04500
EST
5507
322
5165
165
5000
377
4500
3000
4000
377
3000
1000
2300
200
2100
100
2000
377
1000
177777

THE SYMBOL TABLE (LOADER MAP) (Continued)

A symbol may be flagged on the loader map with one of two letters. A U appears on the lefthand side of the symbol if the symbol is an external for which no entry has yet been defined, i.e., an unresolved external.

An M on the lefthand side of the symbol means the symbol is defined in two or more entry or .COMM statements.

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



```
TEMP

            NMAX    Ø44723
            ZMAX    ØØØ244
            CSZE    ØØØØØØ
             EST    Ø5Ø131
             SST    Ø52ØØ1

            AHESZ   ØØØØØØ
            BINAR   ØØ62Ø7
            BUFFE   Ø16711
            BUGIN   ØØØØØØ
            CHSW    ØØØ174
            CKSQ    ØØØØØØ
            COLØ1   Ø35622
            COLØ2   Ø35623
            COLØ3   Ø35624
            COLØ4   Ø35625
            COLØ5   Ø35626
            COLØ6   Ø35627
            COLØ7   Ø3563Ø
            COLØ8   Ø35631
            COLØ9   Ø35632
            COL1Ø   Ø35633
            COL11   Ø35634
            COL12   Ø35635
            COL13   Ø35636
            COL14   Ø35637
            COL15   Ø3564Ø
            COL16   Ø35641
            COL17   Ø35642
```

THE SYMBOL TABLE (LOADER MAP) (Continued)

To obtain a copy of the symbol table, respond with the digit

6

to the loader "*" prompt.

To obtain a copy of only the undefined symbols in the symbol table, respond with the digit

9

to the loader "*" prompt.

EXECUTION OF LOADED PROGRAMS

Initiation of Execution

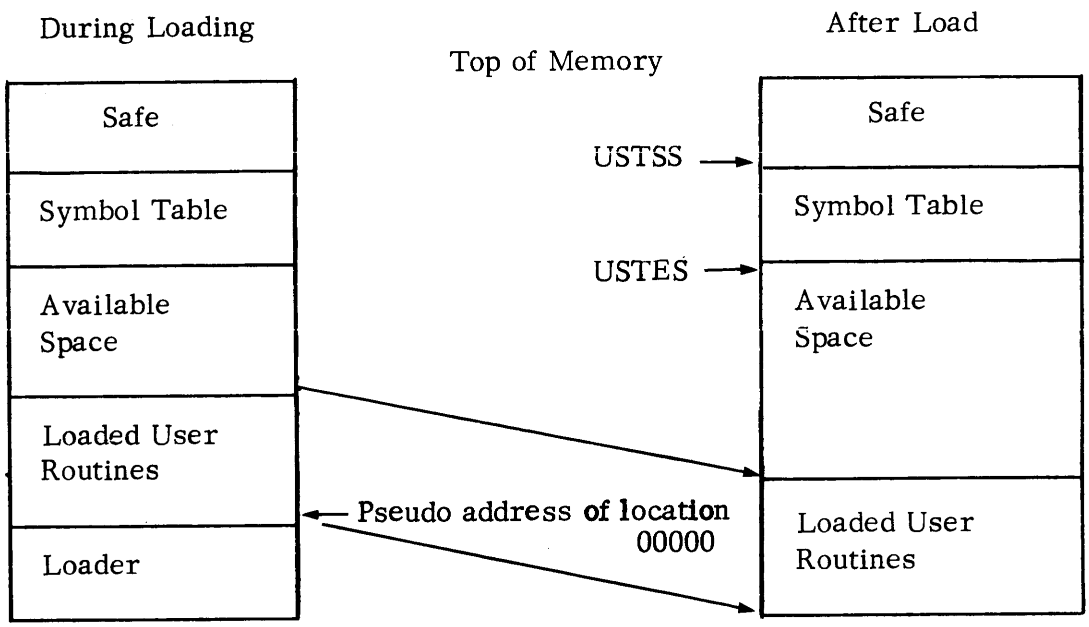
Loading of programs is terminated when the user responds

8

to the loader query "*". The programs previously loaded are then moved to reside at 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 loading is terminated, the following occurs:

> Location 377 is unconditionally initialized to 2406 (JMP @.+6), providing a convenient restart address. (Location 405 of the User Status Table, UST, is set to the starting address of the loaded core image by the loader. See UST layout on page 1-16.)
>
> Memory is shuffled down to reflect the true addresses as shown on the following page.

EXECUTION OF LOADED PROGRAMS (Continued)



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, (See page 1-16).

Starting Address for Execution

After shuffling memory, the relocatable loader will HALT. When the user presses CONTINUE, the loader will HALT 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, the address will receive control regardless of the order in which the tapes were loaded.

If more than one binary tape loaded contained 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. Library tapes are provided by Data General as part of the standard software packages.

Library tapes are loaded in the same way as relocatable binaries. The user mounts

LOADING OF LIBRARY TAPES (Continued)

the. library tape in the appropriate input device and responds to the loader "*" query with 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 calls C, none of those programs will be loaded unless some program loaded before the library tape has called A. If A has an entry corresponding to a previously unresolved external, then all three programs A, B, and C will be loaded.

LOADING LOCAL AND TITLE SYMBOLS

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

The loader maintains a local and title symbols switch which is set by default to inhibit loading of local and title symbols. The user can complement the switch, altering the mode, by responding to the loader "*" query with

```
        4
```

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

REINITIALIZATION OF LOADING

If the loading process is terminated by the fatal error (see ERROR DETECTION section), or if the user wishes to start loading over, the loader must be reinitialized. The user can reinitialize loading by responding to the loader query "*" with

```
                  7
```

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, and

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.

The user can obtain a printout of the current memory limits during loading by giving the response

5

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 the user's loaded system.

All errors are indicated by a two-letter code. The code is printed at the teletype followed by a symbol name, if applicable, 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.

ERROR DETECTION (Continued)

Nonfatal Errors

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

DO Error

```
DO nnnnnn
```

If, while attempting to resolve an external displacement, the loader finds 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

or

the index ≠ 00, and the displacement is outside the range:

$$-200 \leq \text{displacement} < +200.$$

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

IB Error

```
IB nnnnnn
```

ERROR DETECTION (continued)

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 a "*" query after issuing the error code. If an improper tape mounted in the reader caused the error, it should be replaced by a relocatable binary or library tape and loading attempted again.

IN Error

```
IN nnnnnn
```

If the user responds 3 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
```

When an entry or named common (.COMM) symbol having 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 will issue a "*" request when the error occurs.

XE Error

```
XE sssss
```

ERROR DETECTION (continued)

If a .GADD block is encountered that references an as yet undefined symbol, an external undefined in external expression (XE) error occurs. Zero is stored in the memory cell. The undefined symbol sssss is printed out following the error indicator.

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

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 incorrect checksum that was computed.

MO Error

```
MO nnnnnn
```

If the value of NMAX plus the loader size itself conflicts with the bottom of the

ERROR DETECTION (continued)

loader's symbol table, a memory overflow (MO) 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.

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:

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

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

NC Error

```
NC sssss nnnnnn
```

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, a named common (NC) error results. sssss gives the symbol name of the labeled COMMON and nnnnnn indicates the size of the labeled COMMON requested by the present .COMM .

OW Error

```
OW nnnnnn
```

The loader does not permit memory cells to be overwritten by subsequent data once they are loaded. If an attempt to overwrite is made, an overwrite (OW) error occurs. The absolute address where the overwrite was attempted is given by nnnnnn.

ERROR DETECTION (Continued)

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 is origined at location 400: a template is shown below 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
```

USER STATUS TABLE OF LOADING INFORMATION (Continued)

```
USTIN   =   17          ;INITIAL START OF NREL CODE
USTOD   =   20          ;OVERLAY DIRECTORY ADDRESS
USTSV   =   21          ;FORTRAN STATE VARIABLE SAVE ROUTINE
USTEN   =   USTSV       ;LAST ENTRY
```

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.

Location 402 and 403 - USTSS and USTES point to the start and end of the symbol table respectively as shown in the diagram on page 1-4. 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 - USTSA 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 a JMP @.+6, which transfers control to the program starting address. 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 restarted.

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.

Location 411 and 412 - USTIT and USTBR are set to 0 by the loader.

Locations 413-16, 420-21 - These locations are compatible with RDOS.

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

END OF CHAPTER

CHAPTER 2

SOS MAGNETIC TAPE/CASSETTE RELOCATABLE LOADER

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 teletypewriter:

```
                    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.

OPERATION (Continued)

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 teletypewriter 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
```

USER STATUS TABLE (Continued)

```
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
```

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 format 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, printing 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 teletypewriter key, e.g., RETURN.

/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 possible 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 permit both reading and writing.

aaaaa I/O ERROR nn

where: aaaaa is the address associated with the error.

COMMAND LINE (continued)

nn is one of the following RDOS codes:

1 - Illegal file name
7 - Attempt to read a read-protected file.
10 - Attempt to write a write-protected file.
12 - 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 on the teletypewriter.

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.

ERROR MESSAGES (Continued)

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:

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 ≤ 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.

ERROR MESSAGES (continued)

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:

CHECKSUM ERROR nnnnnn

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

ERROR MESSAGES (continued)

NEGATIVE ADDRESS nnnnnn

This error occurs if bit 0 of an address word is set to 1. The assembler restricts addresses to the range: $0 \leq \underline{\text{address}} < 2^{15}$; 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.

END OF CHAPTER

CHAPTER 3

THE RDOS/DOS RELOCATABLE LOADER (RLDR)

This chapter describes the RDOS/DOS Relocatable Loader (RLDR). First, it gives a Loader overview, then describes the following topics:

- RLDR Files
- Command Line
- Global Switches
- Local Switches
- Load Map
- System Tables
- Specifying NREL Addresses
- Symbol Tables
- Command Line Examples
- Example Program with Overlays
- Error Messages

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 compilers which use ASM, 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 wherever 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 (e.g., unmapped NOVA and mapped NOVA), 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 OVERVIEW (continued)

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 via the macroassembler .LMIT pseudo-op, described in Appendix C.

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 in this directory.

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

```
LINK RLDR.SV  %MDIR%:RLDR.SV)
LINK RLDR.OL  %MDIR%:RLDR.OL)
LINK SYS.LB   %MDIR%:SYS.LB)
```

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 is not in the master directory, type the full directory specifier name instead of %MDIR% for this file; e.g.,

```
LINK RLDR.SV DP1:RLDR.SV
```

and so on.

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.

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: a) the module satisfies an unresolved symbol from another module, or b) you specified a force load when you built the library. When a module is loaded because of a), 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 will receive the overlays specified within the square brackets one-by-one when the program executes. The overlays (ovnames) you specify will be placed within the overlay file. The node in the save file will be 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.

LOADING OVERLAYS (continued)

The square brackets establish 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] )
```

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. 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 out with zeroes to the node size and copies the overlays to the overlay file.

In the command line above, if RBs OV1, OV2, OV3, and OV4 are 10, 15, 20 and 40 words respectively, RLDR will use OV4's length (40), and round it up to the next multiple of 256 words; thus 256 words will be the size of the node. If the sizes were 200, 60, 190 and 90 words instead, RLDR would use the length of the first overlay (OV1 with 200 words, plus OV2 with 60 words), then 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 Chapter 4 of your system reference manual for more on overlays.

GLOBAL SWITCHES

Each global switch modifies RLDR's operation globally. You append each global switch to the command word RLDR. Local switches, described next, modify "arguments" in the RLDR command line.

/A Produce 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 (e.g., $LPT/L, MAPFILE/L) with the local /L switch to receive the alpha listing. For example:

```
RLDR/A MYPROG MYPROG.AL/L )
```

GLOBAL SWITCHES (continued)

/B Use short Task Control Blocks (works 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 Create this file for an RTOS, SOS, or stand-alone 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 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 )
```

/D Include a debugger and symbol table in the save file. This switch places an external DEBUG in RLDR's symbol table; by default this copies the symbolic debugger from the system library 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 will place 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 symbol's value.

GLOBAL SWITCHES (continued)

/E Display 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. If you want error messages sent to both the listing file and the console, use this switch.

/G Print labeled common warning messages wherever new labeled common appears in overlays. Normally, RLDR prints each warning message only once for a load. See global /R and Appendix B for more on labeled common in overlays.

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

/I Do not create a UST, TCB, or other system tables, and start NREL code at 445_8 and ZREL code at 50_8. You must also use local /Z if you want ZREL to start at an address other than 50_8. To have the save file begin at 0, use global /Z. Programs created with /I cannot execute under any operating system. This switch is most useful for creating absolute data files.

/K Keep 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, and in Appendix D.

/M Suppress all RLDR output to the console. This speeds loading on slow teletypewriter systems, but you should use it carefully since it also suppresses error messages.

/N Do 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 Omit the program symbol table even though global /D is used.

/P Print 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 will be reported incorrectly (without adjustment for multiple tasks).

/R Place new overlay common in the root program. If an overlay in this command line declares a new named common symbol (.COMM pseudo-op), place 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. See Appendix B for more detail on named common, and global /G to enable warning messages.

Note: RLDR cannot load virtual overlays (local /V) if you use global /R.

/S Place 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 Do not resolve undefined symbols to -1. Normally, the memory locations which reference an undefined symbol are chained to one another (see Chapter 1, 'The Loader Map" illustration). If the symbol is never defined, the address chain remains intact when you include the global /U switch.

/X Allow up to 128 system overlays. The SYSGEN program uses this switch when it instructs RLDR to generate a new operating system.

/Y Allow up to 256 system overlays. See the comment under global /X, above.

/Z Start the save file at location zero. Such a file cannot execute under RDOS or DOS, but can execute in stand-alone mode, or under RTOS or SOS. 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 CLI manual).

LOCAL SWITCHES

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

n/C Allot octal "n" 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 Send error messages to file name when a listing file has been specified (local /L). You must also include global /E. For example:

```
RLDR/E MYPROG $LPT/L $TTO1/E )
```

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. Start 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 )
```

This command line builds a save file name MYPROGFG.SV from the binary MYPROG. MYPROGFG's system table will begin at 20016_8; its page zero code will begin at 200_8.

n/K Allot octal "n" 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).

LOCAL SWITCHES (continued)

name/L

Send the load map and all errors to file name. If name is a disk file name, RLDR will create it and write the map and errors to it. If name exists, RLDR will append to it. Unless you also specify global /E, errors will not go to 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)
```

n/N

Start 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 (i.e., 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)
```

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

Give the save file name. Normally, the CLI gives the save file the name of the first binary in the command line, with the .SV extension; first, however, it deletes any existing save file with the same file name. 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)
```

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 were not specified by a .ENT psuedo-op are not retained. If you specify global /U to the assembler, and local /U to the RLDR, these symbols will be 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 [OVLY0/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 Load overlays as virtual overlays (mapped RDOS only). Virtual overlays occupy mapped extended address space during program execution; they are further described in Chapter 4 of the RDOS Reference Manual. 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.

THE LOAD MAP

RLDR's load map has two parts: a memory map and a list of symbols. (If you specified global /A, it has a third part: an alphabetical list of symbols.) For example, take the command line:

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

The load map produced by this command might look like this (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.                  002070
5.   ROOT1
6.                  002276
7.                  001,000   OVL10   001006
8.                  001,001   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       000463
```

In this load map, lines 0 and 5 contain the program RB titles. If the command line contained a user library, RLDR would also print the titles of modules loaded from the library. It 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 it gave addresses 1470 to 2070 to the first node, and 2276 to 3676 to the second node. Each node is large enough to accept the largest overlay associated with it.

THE LOAD MAP (continued)

Lines 2-3 and 7-8 describe the overlays for node 0 and 1, respectively. You can interpret line 2 as follows:

```
node        overlay          overlay        overlay
number      number           title          length (words)
    \       /                  |              |
     000,000                 OVL00          000004
```

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 had been an undefined or multiply-defined symbol, or named common, the load map would show one of the following codes, followed by the symbol name:

Code	Meaning
XD	External displacement (. EXTD) undefined.
XN	External normal (. EXTN) undefined.
C	Named common symbol.
M	Multipy-defined symbol.

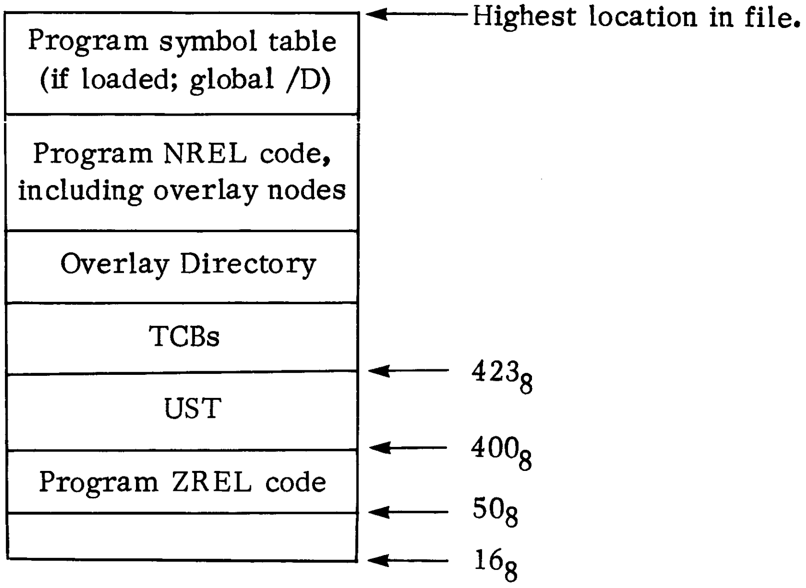
There are no RLDR flags in this load map, but line 12 contains an error message. This message means that neither ROOT0 nor ROOT1 specified a starting address after a . END pseudo-op. Someone will have to correct this in the sources, then reassemble and reload the program, before the program will run.

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 in the next section. Lines 19, 20, and so on name 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; if the command line specified overlays, it also builds an overlay directory into the save file. 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. The following figure shows the arrangement of these tables in the save file:


Highest location in file.
Program symbol table (if loaded; global /D)
Program NREL code, including overlay nodes
Overlay Directory
TCBs
UST
Program ZREL code
423_8
400_8
50_8
16_8

The system uses locations 0 - 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. Chapter 5 of your system reference manual describes the UST further.

Task Control Blocks (TCBs)

Each TCB records runtime information on a program task. RLDR builds one TCB into the program for the number of tasks you specify; you can compute the number of words for the TCB area with the formula:

(number-of-tasks) * TLN

SYSTEM TABLES (continued)

Entry TLN is the number of words for each TCB; generally, this is 21_8. The system parameter file, PARU.SR, describes TLN; see the PARU.SR cross-reference in your system reference manual to find TLN. Chapter 5 of your reference manual further describes TCBs.

Overlay Directory

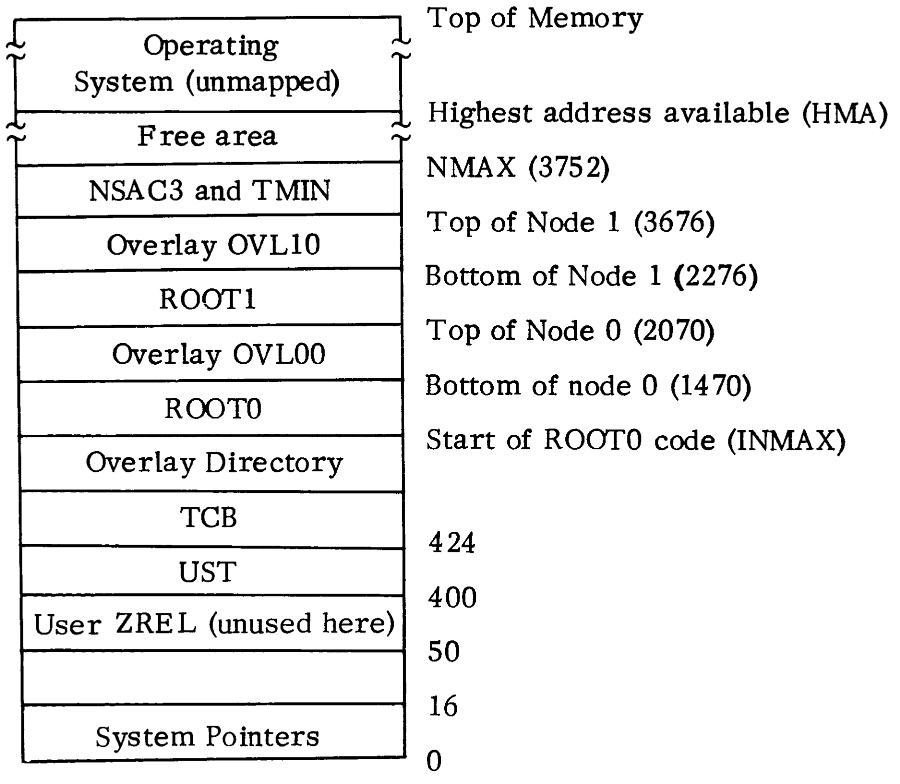
The overlay directory maintains address, load, and use information on each overlay node. RLDR creates the overlay directory only if your command line specifies overlays. The directory length is:

(4*number-of-nodes) +1

For more on the overlay directory, consult the appropriate appendix of your system reference manual.

Memory Map Illustration

The following illustration shows how file ROOT.SV would look in main memory during execution. We showed the command line and load map for this file in the "Load Map" section earlier. Let's assume that the program has loaded (.OVLOD) an overlay into each node.



SYSTEM TABLES (continued)

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.

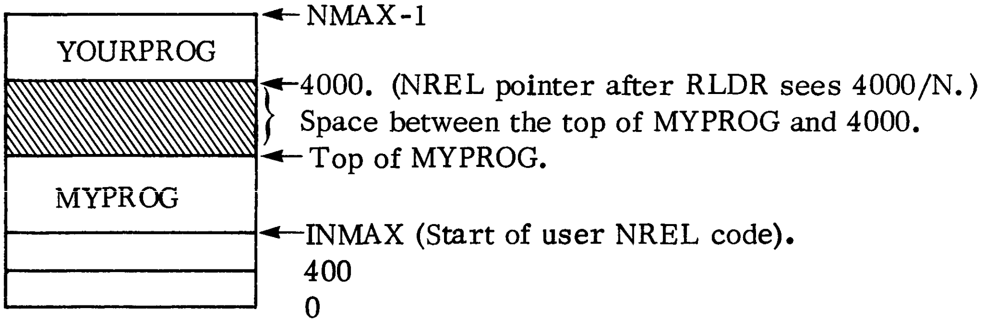
HMA is the highest memory address available to user programs; INMAX is the start of use NREL code. (The illustration is not to scale.)

SPECIFYING NREL ADDRESS

As described under the local /N switch, you can increment 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
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

Any program itself can change its NMAX during execution via the .MEMI system call; this is very important for a swap or a chain.

SYMBOL TABLES

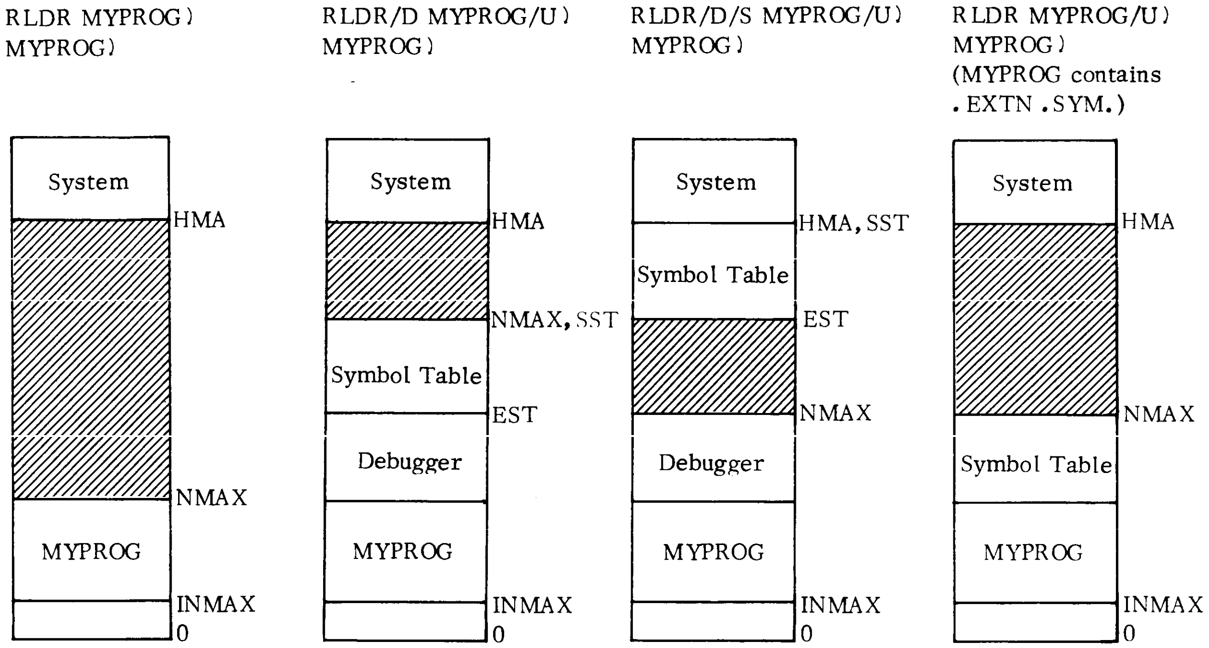
RLDR creates two symbol tables for its own use whenever it builds a .SV file. The first table resides only in memory and contains undefined symbols. The second table, a disk file named savefilename.ST, contains both undefined and defined symbols. As RLDR resolves symbols, it removes them from the memory-resident table and marks them as "defined" in savefilename.ST. 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 have been resolved in savefilename.ST; RLDR then deletes savefilename.ST (unless you used global /K in the command line).

If the memory-resident table grows too large for available memory, RLDR displays a SYMBOL TABLE OVERFLOW message and aborts the command. You can often solve the overflow prdblem by rearranging the order of files in the command line to reduce the number of symbols that are undefined at one time.

There is a third symbol table which differs from the tables RLDR creates for its own use. This is the table which you can have inserted in the save file by appending the global /D switch (which also loads a debugger). For clarity, we'll 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 (assembler global /U, RLDR local /U switches). 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), this may mean that duplicate symbols exist in the program symbol table. This may present debugging problems for users of extended RBs.

The position of the program symbol table (if any) 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.



Aside from the position of the system, these diagrams apply to both mapped systems and background programs in unmapped systems.

For more details on RLDR's symbol tables, see Appendix D.

Checking Save File Size

Because RLDR loads the save file directly onto the disk, you can load a file which is too large to execute in user space. If you try to execute such a file, you'll 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 will build the symbol table down from HMA. If the symbol table would overwrite the program code, RLDR displays the message:

```
SYMBOL TABLE TOO LARGE FOR CORE STORAGE
```

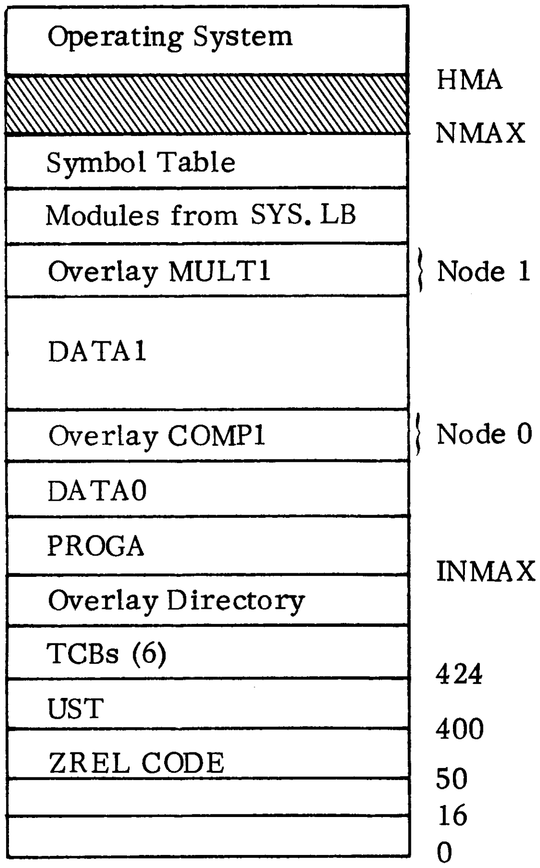
This doesn't necessarily mean that the program won't fit into memory, because the debugger and symbol table were also loaded, and they require significant space. If you don't receive the error message, you can be sure that the program will fit into memory; you can then issue the RLDR command you want for this program.

COMMAND LINE EXAMPLES

This section shows two RLDR command lines, and memory maps of the save files they produce. The first command line loads a multitasking program, with a program symbol table and debugger; the second loads a program for execution in an unmapped foreground.

1.
```
        RLDR/D PROGA DATA0 [COMPA, COMPB, COMPC, COMPD] ↑)
                DATA1 [MULT1, MULT2, MULT4] 14/C 6/K)
```

This builds save file PROGA.SV and corresponding overlay file PROGA.OL. When PROGA executes (after it .TOVLDs COMPA into node 0 and MULTI1 into node 1), memory will look like this:



In a mapped RDOS system, the 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 would not affect execution.

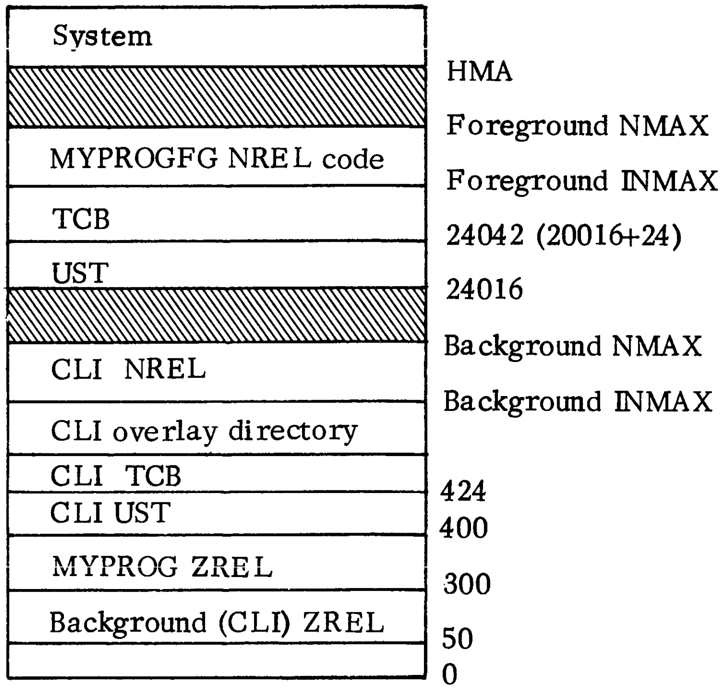
COMMAND LINE EXAMPLES (continued)

2. RLDR MYPROG 300/Z 24000/F MYPROGFG/S)

This builds save file 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 via:

```
EXFG MYPROGFG)
```



EXAMPLE PROGRAM

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)
```

(ASM/L could also have been used with the same results.) This 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.

EXAMPLE PROGRAM (continued)

Although ROOT, OVLY0, and OVLY1 were 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're using a compiler, you'll 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 OVLY0 MACRO REV 06.20              09:01:31 07/25/77
                             .TITL OVLY0
02                           .ENTO OVLY0
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         .SYSTM   ;Print
10 00002'010000         .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 00007'000102 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,C      ;Get the "C".
08 00001'006017         .SYSTM       ;Print
09 00002'010000         .PCHAR       ;it.
10 00003'002403         JMP @.+3     ;Error goes to "ER".
11 00004'002401         JMP @.+1     ;Go to
12 00005'077777         RTURN        ;"RTURN" in ROOT.
13 00006'077777         ER
14 00007'000103 C:      "C
15                      .END

**00000 TOTAL ERRORS, 00000 PASS 1 ERRORS
```

EXAMPLE PROGRAM (continued)

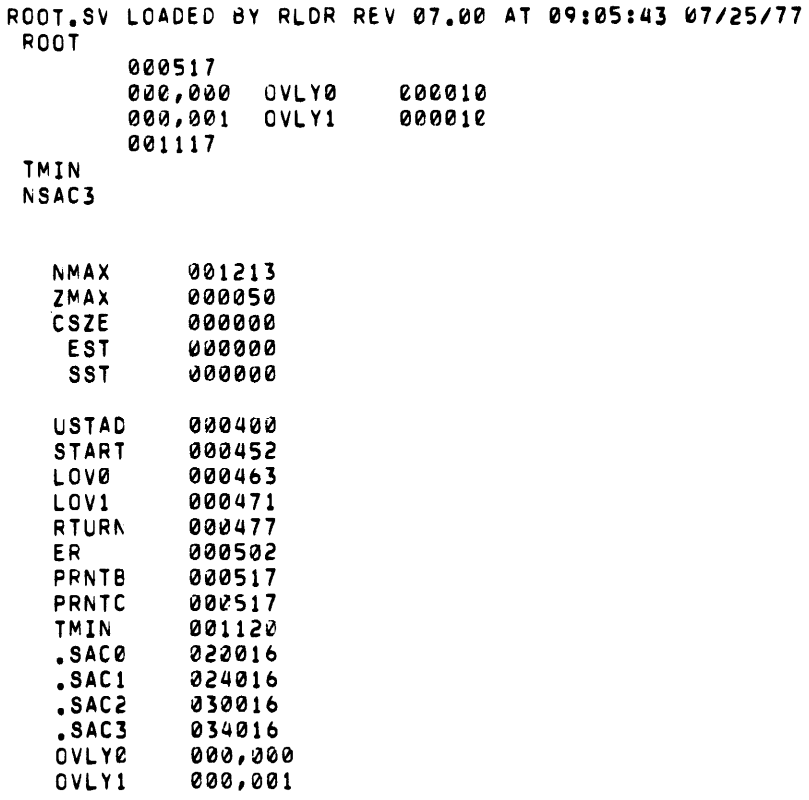
```
0001 ROOT   MACRO REV 6.20              09:01:25 07/25/77
                         .TITL ROOT
                         .ENT START, LOV0, LOV1, RTURN, ER
                         .EXTN OVLY0, PRNTB, OVLY1, PRNTC
          000001         .TXTM 1
                         .NREL        ;NREL code.


   00000'020440 START:   LDA 0, OFILE ;Get overlay filename.
   00001'126400          SUB 1, 1     ;Default mask.
   00002'006017          .SYSTM       ;Open overlay file.
   00003'012000          .OVOPN 0     ;on I/O channel 0.
   00004'002424          JMP ER    ;Error return.
   00005'020426          LDA 0,A   ;GET THE "A".

   00006'006017          .SYSTM    ;Print
   00007'010000          .PCHAR    ;it.
   00010'000420          JMP ER    ;Error return.
   00011'020423 LOV0:    LDA 0,OV0 ;Get OVLY0 addr.
   00012'126400          SUB 1,1   ;Conditional load.
   00013'006017          .SYSTM    ;Load OVLY0
   00014'020000          .OVLOD 0  ;on channel 0.
   00015'000413          JMP ER    ;Required.
   00016'002420          JMP @.PRB ;To PRNTB code in OVLY0.

   00017'020416 LOV1:    LDA 0, OV1;Get OVLY1 addr.
   00020'126400          SUB 1,1   ;Conditional load.
   00021'006017          .SYSTM    ;Load
   00022'020000          .OVLOD 0  ;OVLY1 on channel 0.
   00023'000405          JMP ER    ;Required.
   00024'002413          JMP @.PRC ;To PRNTC code in OVLY1.

   00025'006017 RTURN:   .SYSTM
   00026'004400          .RTN
   00027'002401          JMP .+1   ;Reserved, never taken.

   00030'006017 ER:      .SYSTM    ;Get CLI to
   00031'006400          .ERTN     ;report problem.
   00032'002401          JMP .    ;Impossible.
   00033'000101 A:       "A
   00034'077777 OV0:     OVLY0
   00035'077777 OV1:     OVLY1
   00036'077777 .PRB:    PRNTB
   00037'077777 .PRC:    PRNTC
   00040'000102"OFILE:   .+1*2
   00041'051117          .TXT "ROOT.OL"
          047524
          027117
          046000
                         .END START
```

EXAMPLE PROGRAM (continued)

The RLDR command line for the program was:

```
RLDR ROOT [OVLY0, OVLY1] ROOT.LM/L)
```

This created save file ROOT.SV and overlay file ROOT.PL; it also created disk file ROOT.LM and sent the load map to it. File ROOT.LM contained the following information after the load:



```
ROOT.SV LOADED BY RLDR REV 07.00 AT 09:05:43 07/25/77
 ROOT
        000517
        000,000  OVLY0     000010
        000,001  OVLY1     000012
        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
```

EXAMPLE PROGRAM (continued)

Lines in the listing have the same meanings described under "LOAD MAP", earlier in the chapter. Reading the listing, we see RLDR's proud announcement; then we see the name ROOT. Below ROOT in the listing, we see 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.)

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 name (OVLY0) instead of number, since OVLY0 used .ENTO. 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.

RLDR ERROR MESSAGES

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:

DISPLACEMENT OVERFLOW <u>nnnnnn</u> [<u>oooooo</u>]

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 outside the range:
$-200 \leq \text{displacement} \leq +200$

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

oooooo is the node number/overlay number if an overflow occurs in an overlay.

RLDR ERROR MESSAGES (continued)

ILLEGAL BLOCK TYPE nnnnnn

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

MULTIPLY DEFINED ENTRY sssss nnnnnn

This error occurs when an entry symbol or named common (.COMM) symbol, ssssss, having the same name as one already defined, is encountered during loading. 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 will result in an error.

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.

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.

SYSTEM LIBRARY NOT FOUND

This error occurs when the system library (SYS.LB) could not be found on the current directory.

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 into the TCBPC word of the TCB pointed to be USTCT. It must be stored as the starting address multiplied by 2.

BINARY WITHOUT END BLOCK

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

RLDR ERROR MESSAGES (continued)

TASKS OR CHANNELS SPECIFIED = 0

This error occurs when there was 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.

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.

NO SCHEDULER STARTING ADDRESS

In a stand-alone load (global /C) this error occurs if no start block contained a starting address. The starting address can be patched into USTSA.

WARNING *** ZERO LENGTH OVERLAY

This error indicates that an attempt has been made to load an overlay that contains nothing.

Fatal Errors

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

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

For example:

```
LOAD OVERWRITE 001700
** FATAL LOAD ERROR **
```

The message is output to the error file, and return is made to the CLI which prints the message:

```
FATAL SYSTEM UTILITY ERROR
```

RLDR ERROR MESSAGES (continued)

The fatal errors are:

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: $0 \leq \text{address} \leq 2^{15}$; however, the error can be caused by a reader error. nnnnnn represents the negative address.

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) 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.

LOAD OVERWRITE nnnnnn

The loader does not permit save or overlay file locations to be overwritten by subsequent data once they are loaded. If an attempt to overwrite is made, this error occurs. The absolute address where the overwrite was attempted is given by nnnnnn.

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 still undefined.

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.

SYMBOL TABLE TOO LARGE FOR CORE IMAGE

This error occurs when a global switch /S has been given in the RLDR command line and the symbol table would overwrite loaded programs in the save file built by the loader.

ILLEGAL LOAD ADDRESS

This error occurs when an attempt is made to load into locations 0 - 15.

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.

RDOS ERROR

This error indicates that the loader issued a system call that could not be completed and that resulted in an exceptional return. See the RDOS User's Manual for system calls and possible error returns.

TASK MONITOR ERROR (USTCH)

This error occurs when a .COMM block with symbol TASK is encountered at a point when NMAX differs from the initial value of NMAX. This occurs if .COMM TASK occurs in some module after the first module is loaded.

OVERLAY DIRECTORY OVERFLOW

This error occurs when the number of nodes exceeds 128 or number of overlays at a given node exceeds 256.

NODE .COMM DEFINITION OUTSIDE FIRST MODULE

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.

TCB OVERLAY TABLE OUTSIDE FIRST MAP PAGE BOUNDARY

This message means that the system tables (TCBs, overlay directory, etc.) extend above 2000_8, thus the program would not run correctly in a mapped system.

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 already defined as labeled common. If so, RLDR checks to see if it 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 doesn't fit correctly, RLDR displays this error message. (Appendix B describes this sequence further.)

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.

END OF CHAPTER

CHAPTER 4

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-descriptior1/N  overlay-list [overlay-descriptor2/N ↑ )
       overlay-list ...]  [filename/L]  [filename/E])
```

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, respectively.

The global switches are:

/A Create an alphabetical/numeric memory map of new symbols. You must also designate a listing file with local /L.

/E Send error messages to the console. (Used only when there is a listing file (/L) that suppresses console error output.)

/H Print 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 try 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 will trigger an error message and abort the OVLDR command.

If you omit global /R, OVLDR will try to place any new named common in the overlay node. See Appendix B for more on labeled common in overlays.

The local switches are:

name/E Send error message and other messages to file name. Preceding file is designated to receive error and information messages.

name/L Send memory map to file name. This includes new symbols (those declared in replacement overlay nodes). The map will be numeric unless you also include the global /A switch.

old-overlay/N Must follow an overlay-descriptor. Create replacement for old-overlay. The old-overlay is an overlay-descriptor.

OVLDR works only if the following conditions exist:

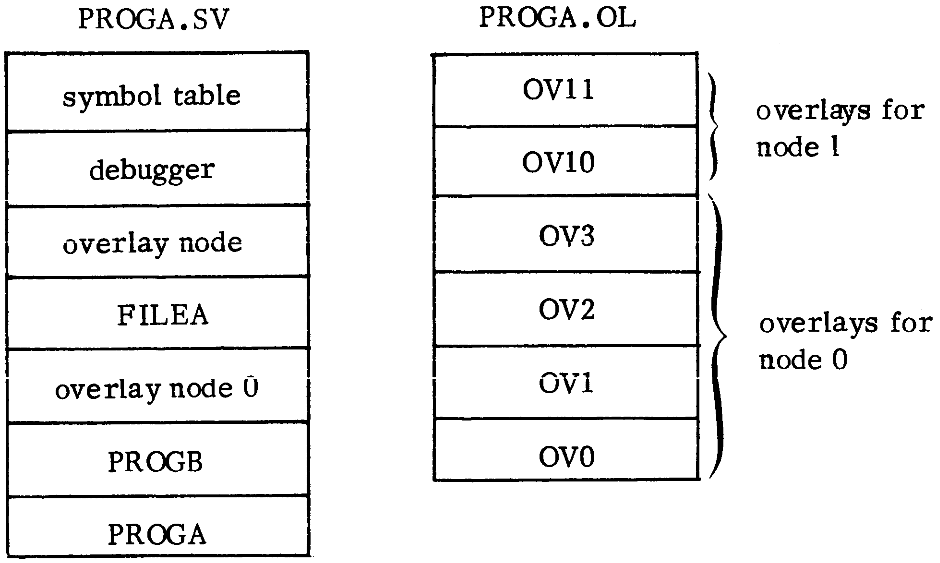
1. There must be an overlay file, filename.OL, created by the RLDR command.

2. The save file, filename.SV, must contain 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])
```

The following save and overlay files were created:



The /D global switch loaded the symbol table, but 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 (e.g., in PROGB or FILEA):

```
.EXTN .SYM.
```

Now we'll create an overlay replacement file via OVLDR. Assume that we want to replace OV3 (node 0, number 2), and OV10 (node 1, number 0). We want to substitute the binaries NEW3 and NEW4 for OV3, and binary NEW10 for OV10. If we identified OV3 and OV10 with .ENTO, and declared OV3 and OV10 in a .EXTN in the save file, we can simply type:

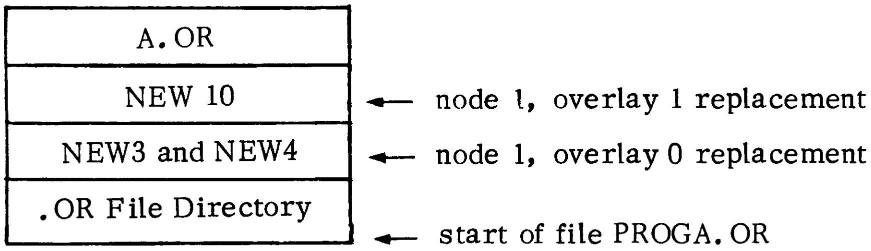
```
OVLDR/A/E PROGA OV3/N NEW3 NEW4 OV10/N NEW10 PROGA.RM/L )
```

If we omitted .ENTO, we 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 would be:

```
OVLDR/A/E PROGA 000002/N NEW3 NEW4 00400/N NEW10 PROGA.RM/L )
```

(We could omit the leading zeroes in the overlay descriptor.)

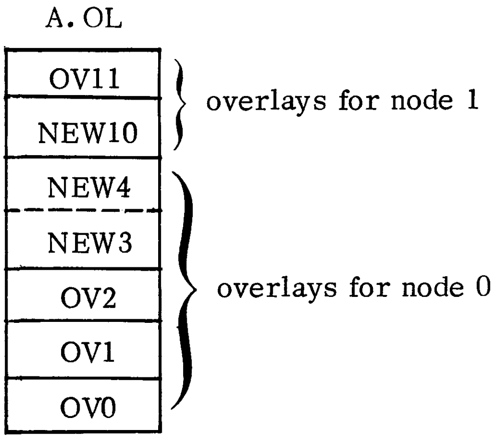
Either command creates the overlay replacement file PROGA.OR, and listing file PROGA.RM (the .RM stands for Replacement Map). The listing file contains a normal and alphabetical map of new symbols. Error messages go to both the console and the listing file. The replacement file PROGA.OR looks like this:



Now, assuming we received no errors, we execute the REPLACE step:

```
REPLACE PROGA)
```

After this command executes, PROGA.OL appears as follows:



The error messages that may occur in executing an OVLDR command are given below. They are all fatal.

ILLEGAL LOAD ADDRESS

Any attempt to load into an address outside the overlay node.

NO SYMBOL TABLE

The symbol table was not created when the save and overlay files were loaded.

NO OVERLAY DIRECTORY

The save file does not contain the overlay directory, OLDIR.

INSUFFICIENT MEMORY

OVLDR cannot execute in available memory.

COMMON SIZE ERROR

An overlay defines blank COMMON 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.

NEW .COMM CANNOT BE DECLARED IN NEW OVERLAY

This message occurs when you used global /R, and a replacement overlay declared new labeled common. OVLDR cannot change the node size in the root program, thus it cannot assign new labeled common outside the node.

.SV FILE SYMBOL IS EXTD/N FOR LABELED COMMON

OVLDR sends this message if 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, thus cannot be redefined.

END OF CHAPTER

APPENDIX A

RELOCATABLE BINARY BLOCK FORMATS

This appendix begins by describing radix 50 representation, which MAC, ASM, and compilers which use ASM, use to condense five-character symbols. It then describes the block types which 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:

$a_4a_3a_2a_1a_0$

where: Each a may be one of the following characters:

A - Z (one of 26 characters)
0 - 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

Licensed Material - Property of Data General Corporation

RADIX 50 REPRESENTATION (continued)

If any a is translated to b, the bits required to represent the original a can be computed as follows:

$$N_1 = (b_4 *50+b_3) *50)+b_2$$

$N_{1maximum} = (50)^3 - 1 = 174777$, which can be represented in 16 bits (one word)

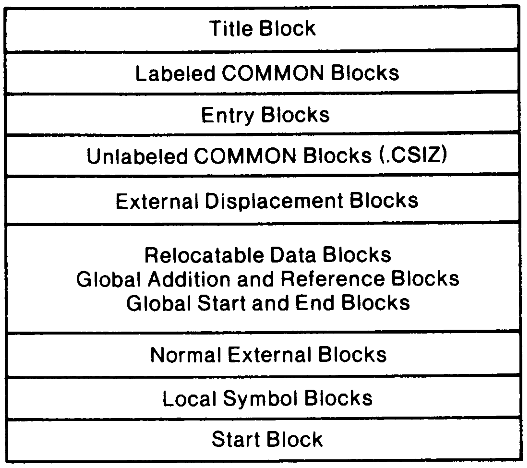
$$N_2 = (b_1 *50) +b_0$$

$N_{2maximum} = (50)^2 - 1 = 3077$, 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

SD-00646

RELOCATABLE BINARY BLOCK TYPES (continued)

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 above.

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 - 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 3-5 are reserved for relocation flags. Some block types contain these flags, other don't. The relocation property of each address, datum, or symbol is defined in three bits. For example, for a Relocatable Data Block, bits 0-2 of word 3 apply to the address, bits 3-5 apply to the first data word, bits 6-8 apply to the second data word, etc. The meaning of the bit settings is given in the table following.

Bits	Meaning
000	Illegal
001	Absolute
010	Normal Relocatable
011	Normal Byte Relocatable
100	Page Zero Relocatable
101	Page Zero Byte Relocatable
110	Data Reference External Displacement
111	Illegal

All other blocks use bits of word 3 only and set words 4 and 5 to zero.

Word 6 contains a checksum, such that the sum of all words in the block is 0.

For those blocks containing user symbols, each symbol entry is three words long.

RELOCATABLE BINARY BLOCK TYPES (continued)

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

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)

		Word
7		1
word count (-3)		2
0		3
0		4
0		5
checksum		6
title in radix 50		7
	flags	8
0		9

SD-00647

The third word of the user symbol entry for a title is set to 0.

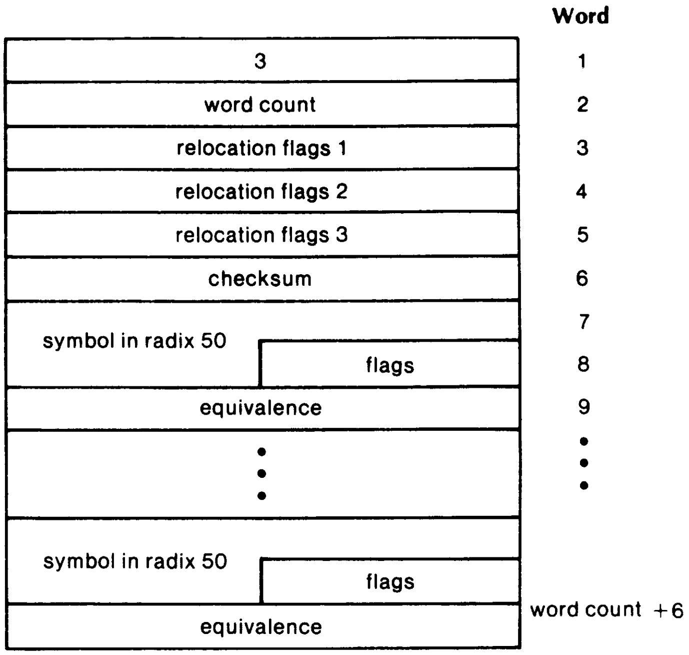
LABELED COMMON BLOCK (.COMM)

	Word
13	1
word count (-4)	2
relocation flags 1	3
0	4
0	5
checksum	6
symbol in radix 50	7
flags	8
0	9
expression value	10

SD-00648

Bits 0 - 2 of the relocation flags (word 3) apply to the expression (expr following . COMM). All other bits of the word are zeroed.

ENTRY BLOCK (.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)

	Word
15	1
word count (-1)	2
relocation flags 1	3
0	4
0	5
checksum	6
expression value	7

SD-00650

Bits 0-2 of the relocation flags (word 3) apply to expression (expr following .CSIZ). All other bits of word 3 are zeroed.

EXTERNAL DISPLACEMENT BLOCK (.EXTD)

	Word
4	1
word count	2
0	3
0	4
0	5
checksum	6
symbol in radix 50	7
flags	8
077777	9
⋮	⋮
symbol in radix 50	
flags	
077777	word count +6

SD-00651

The third word of each user symbol entry in the External Displacement Block is set to 077777.

RELOCATABLE DATA BLOCK

	Word
2	1
word count	2
relocation flags 1	3
relocation flags 2	4
relocation flags 3	5
checksum	6
address	7
data	8
data	9
⋮	⋮
data	word count +6

SD-00652

Contents of the relocation flag words (words 3-5) 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)

		Word
14 for .GADD; 20 for .GREF		1
word count (-5)		2
relocation flags 1		3
0		4
0		5
checksum		6
address		7
symbol in radix 50		8
	00000	9
0		10
expression value		11

SD-00653

Bits 0-2 of the relocation flags (word 3) apply to the address and bits 3-5 apply to the expression. All other bits of word 3 are zeroed.

GLOBAL LOCATION START AND END BLOCKS (.GLOC)

Start Block	Word	End Block
16	1	17
-3	2	0
0	3	0
0	4	0
0	5	0
checksum	6	-17
symbol in radix 50	7	
00000	8	
0	9	

SD-00654

NORMAL EXTERNAL BLOCK (.EXTN)

	Word
5	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
address of last reference	9
⋮	⋮
symbol in radix 50	
flags	
address of last reference	word count +6

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	
equivalence	word count + 6

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.

START BLOCK

	Word
6	1
word count (-1)	2
relocation flags 1	3
0	4
0	5
checksum	6
address	7

SD-00657

Bits 0 - 2 of the first relocation flag word are used for address relocatability; other bits of word 3 are 0.

LIBRARY START AND END BLOCKS

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.

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

SD-00658

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've repeated the listing, without comments, below.

```
                             .TITL ROOT
                             .ENT START, LOV0, LOV1, RTURN, ER
                             .EXTN OVLY0, 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 LOV0:           LDA 0,OV0
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, OV1
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 OV0:            OVLY0
00035'077777 OV1:            OVLY1
00036'077777 .PRB:           PRNTB
00037'077777 .PRC:           PRNTC
00040'000102"OFILE:          .+1*2
00041'051117                 .TXT "ROOT.OL"
       047524
       027117
       046000
                             .END START
```

RB FILE ILLUSTRATION (continued)

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 10-word series (if any) are shown in the right column. (For the rest of this appendix, all number 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've 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 007414 000015 000401 040400 ..............A.
110 001000 173377 111104 111144 000000 114072 016000 177577 .....D.....:....
120 177577 177577 177577 041000 047522 052117 047456 000114] ......B.ORTCC..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] ----   ----   ----   ----   ----   ----   ----   ................
```

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 bits 0-11 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 3 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.

RB FILE ILLUSTRATION (continued)

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 — Contain 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 GREF	=	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.).

END OF APPENDIX

APPENDIX B

OVERLAYS AND NAMED COMMON (RDOS/DOS)

Named common is code you 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.

You need to read the rest of this appendix only if you want overlays to declare named common in your programs.

NAMED COMMON AND OVERLAYS

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 within 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 upwards. 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. The map will contain the warning message LABELED COMMON IS ROOTBOUND to indicate the incorrect node start address.

SUMMARY

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

```
RLDR PROGA [A,B,C])
```

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 COMM's assigned memory location 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 alloted 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 COMM's assigned memory location is greater than the node start, and this location plus SIZE is less than the next available location, RLDR continues.

- If COMM's assigned memory location is greater than node start, and this location plus SIZE is greater than the next available location, RLDR displays an ALIGNMENT ERROR message and aborts the command.

SUMMARY (continued)

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.

END OF APPENDIX

APPENDIX C

THE .LMIT FEATURE

You can instruct RLDR to partially load a module with the Macroassembler's .LMIT pseudo-op. To use .LMIT, declare a .LMIT symbol in a module, then use the symbol in a different module which you want partially loaded. The module to be partially loaded must .ENTer the symbol. Only the first module which .ENTers the symbol and uses it will be partially loaded. If you place the .LMIT symbol in ZREL, RLDR will load all of this module's NREL code and its ZREL code up to the symbol. If the symbol is in NREL, RLDR loads all ZREL code and NREL code up to the symbol. In the RLDR command line, you must insert the module which establishes the .LMIT symbol before the module you want partially loaded. You can limit-load only one module with a single .LMIT symbol; if the same .LMIT symbol occurs as a .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, you type RLDR A B C). The limit symbol is LIM1, and it instructs RLDR to partially load module B. RLDR finds that ROUT1, which A declares external, is defined in B after the limit symbol; therefore, RLDR stores the value -1 for ROUT1 in its symbol table. It then proceeds to load all of A and C, and B -- up to symbol LIM1. If module C mentions ROUT1, RLDR inserts -1 wherever each ROUT1 symbol occurs; if C mentions ALPHA, RLDR returns an UNDEFINED SYMBOL error.

For .LMIT 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 modules you want partially loaded. 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.

END OF APPENDIX

APPENDIX D

RLDR 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 each defined and undefined symbol. The second table remains in memory and contains only undefined symbols. Formats for these RLDR tables are shown below.

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 head: "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--Word n symbol name in ASCII, packed in ASCII,
2 characters 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 A.
Bit 13 is flag for extended format.

Word 2-n Chain pointers.

END OF APPENDIX

INDEX

Within this index, the letter "f" following a page number means "and the following page"; "ff" means "and the following pages".

Data General

Software Documentation Remarks Form

How Do You Like This Manual?

Title ______________________________ No. __________

We wrote the book for you, and naturally we had to make certain assumptions about who you are and how you would use it. Your comments will help us correct our assumptions and improve our manuals. Please take a few minutes to respond.

If you have any comments on the software itself, please contact your Data General representative. If you wish to order manuals, consult the Publications Catalog (012-330).

Who Are You?

□ EDP Manager
□ Senior System Analyst
□ Analyst/Programmer
□ Operator
□ Other ______________________

What programming language(s) do you use? __________
____

How Do You Use This Manual?

(List in order: 1 = Primary use)

_____ Introduction to the product
_____ Reference
_____ Tutorial Text
_____ Operating Guide
_____ ______________________

Do You Like The Manual?

Yes	Somewhat	No	
□	□	□	Is the manual easy to read?
□	□	□	Is it easy to understand?
□	□	□	Is the topic order easy to follow?
□	□	□	Is the technical information accurate?
□	□	□	Can you easily find what you want?
□	□	□	Do the illustrations help you?
□	□	□	Does the manual tell you everything you need to know?

Comments?

(Please note page number and paragraph where applicable.)

From:

Name ______________________ Title ______________________ Company ______________________
Address ____ Date ______________________

FOLD DOWN FIRST FOLD DOWN

FIRST
CLASS
PERMIT
No. 26
Southboro
Mass. 01772

BUSINESS REPLY MAIL

No Postage Necessary if Mailed in the United States

Postage will be paid by:

Data General Corporation

Southboro, Massachusetts 01772

ATTENTION: Software Documentation

FOLD UP SECOND FOLD UP

STAPLE